Formal synthesis of VLSI layouts from algorithmic specifications

Sadiq M. Sait, Khaled M. Elleithy and Masud-ul-Hasan
Department of Computer Engineering, King Fahd University of Petroleum and Minerals, Dhahran-31261, Saudi Arabia

Due to advances in VLSI technology, it is possible to implement complex digital systems on a single chip. However, modeling such large and complex systems at the structural level is tedious and error-prone. This fact has motivated the development of several high-level synthesis systems. The process consists of translating the abstract behavioral representation generally at the algorithmic level into a structural realizable representation. In this paper, we present a formal approach for high-level synthesis. This formal high-level synthesis system uses \( \mu \)-recursive algorithms to model the behavior to be synthesized. These algorithms can be mathematically verified for correctness before being subjected to the task of translation. As a case study, the modeling and synthesis of VLSI layouts for matrix-matrix multipliers is discussed.

Keywords: formal synthesis, VLSI layouts, algorithmic specifications, high-level synthesis

1. INTRODUCTION

Due to advances in VLSI technology, it is possible to implement complex digital systems on a single chip. However, modeling such large and complex systems at the structural level is tedious and error-prone. This fact has motivated the development of several high-level synthesis systems. High-level synthesis (HLS) refers to the process of synthesizing the hardware of a digital system from an abstract high-level description. The process consists of translating the abstract behavioral representation, generally at the algorithmic level, into a structural realizable representation.

The high-level synthesis problem has two main subproblems: one the extraction of structure from the behavioral description of digital systems, and two, the optimization of data flow and control flow, subject to a set of constraints (resources, timing, space, etc.). In order to alleviate the complexity of the synthesis problem restrictions are imposed. These restrictions may be on (1) the constructs used in front-end specification, (2) the structure of the hardware obtained, or (3) the class of digital circuits to be synthesized. For example, a system for a specific class of applications such as protocol processors, or systolic arrays, is simpler to design compared to one that must translate any general purpose algorithmic description to efficient hardware.

The first goal of any HLS system must be to produce a functionally correct circuit which needs no further verification after the design is completed. This requirement can be accommodated by a proper choice/design of the constructs of the algorithmic input specification language. Most existing HLS systems are based on programming languages and do not emphasize the need for formal synthesis and verification. In order to support HLS, the specification language must be simple and semantically well characterized. It must facilitate writing specifications using as few predefined objects and concepts as possible. Current hardware specification languages are far from this. In this paper, we present a framework for formal HLS of digital systems. The input algorithmic specification language (ASL) is based on \( \mu \)-recursive descriptions to
model the behaviour to be synthesized. These algorithmic models can be mathematically verified for correctness before being translated to hardware.

The ASL constructs include (iteration, loops, recursion, etc). Therefore it is easy to express repetitive and iterative digital systems, especially those for DSP applications such as convolution, FFT, etc. Numerical processors and algorithms involving arithmetic computation can also be modelled conveniently.

Behavioural ASL models are translated into structural level models in an intermediate form called the Realization Specification Language (RSL). In this paper we introduce a high-level synthesis system whose front-end specifications can be verified for correctness before implementation in hardware. Emphasis in this paper is on the logic/layout level cell library (backend) which is a mapping of RSL (structural intermediate form) constructs to hardware, and the translation of RSL descriptions to VLSI layouts. To illustrate modelling and synthesis, as an example we present the results of synthesizing VLSI chips of various recursive matrix-matrix multiplication algorithms.

2. PREVIOUS WORK

Up to the late 1970s, research work in the CAD area has been focused toward problems related to the lower levels of the design process. These problems are known as physical synthesis problems. Examples of tools developed for such tasks include timing analysers, circuit extractors, placers, routers, design rule checkers, etc.1. The last two decades have also seen the development of an extremely large number of tools and techniques that translate both physical and structural descriptions to silicon. Serious work on the automation of higher levels of the design process did not start until about a decade ago. In this section we briefly look at some of the reported works in the area of structural and behavioral synthesis.

2.1 RTL synthesizers

Timid efforts to describe large digital systems at the register transfer level began in the early 1960s. This was the time when CAD started being recognized as indispensable. One of the earliest reported works is that of Falkoff2 who used APL to describe IBM System/360. Following this, special RTL languages, called Computer Hardware Description Languages (CHDL), were developed for the structural representation of digital systems. Well known amongst these are AHPL,3 CDL,4 DDL5 and RTS6. The prime objective of CHDLs was to provide a set of notations which could be used to develop a well structured, unambiguous, and concise description of complex digital systems in order to improve communication among designers. There are some excellent textbooks that used these languages as a vehicle to illustrate practical design approaches.7-9. Soon it was realized that a description written in a CHDL can be translated automatically into a logic network.7-9. CHDLs and associated simulators and compilers demonstrated the viability of implementing complex digital systems directly from their RTL descriptions. AHP, DASYs is one such synthesis system which generates logic from structural descriptions at the register transfer level (RTL), and then converts the produced logic descriptions to IC layouts.

2.2 High-level synthesizers

Notation to describe digital systems at even higher levels of abstraction was also developed. Most well known among these are ISPS, PMS10 and VHDL.11. Systems to produce RT-level hardware designs or structural representations from high-level description languages (or models) have been reported. These include the CMU-DA7 and Emerald12 systems, which produce data paths from ISPS, and the design automation assistant (DAA)13 which generates hardware using a rule-based approach. The ADAM system of the University of Southern California14 takes as input a high-level behavioural description in the form of a control and timing graph and generates RTL designs. All the above systems adopt a unique functional specification language whose grammars require that the target architecture be almost completely defined before using the language.

In the past few years several attempts have been made to build a top-down synthesis system which will generate hardware from high-level models. The YASC silicon compiler automatically synthesizes general cells.15 It accepts a behavioural description as an input and transforms it into a boolean level description, which is synthesized into custom cells that are physically synthesized into layout. MIMOLA is a design system which integrates several tools for the automatic synthesis of digital systems from high-level descriptions (data flow graphs).16 MIMOLA, a maximum parallel schedule is first generated, then global module allocation is performed by modelling it as an integer programming problem.

The Flamel system generates a bit-slice hardware from a Pascal program by using compiler-optimizing techniques.17 Ideas for transforming Ada programs into silicon chips have been suggested.18, 19 HARP is another silicon compilation system which creates an RTL description from a Fortran program.21 It is used in conjunction with a VLSI back-end to produce mask patterns.

The first high-level synthesis system to achieve substantial commercial acceptance was the GENESIS silicon compiler22. GENESIS is a complete system for chip design using high-level structural elements. The heart of GENESIS consists of a set of functional blocks (PLAs, ROMs, RAMs, etc.). These blocks can be selected and tailored by users to their specific needs. The control structure can be specified as a state machine and the logic is entered as a truth table.

2.3 Formal high-level synthesizers

None of the above systems include features and facilities...
to verify the correctness of the design before synthesis. In formal high-level synthesis a mathematical framework is used to assist and verify characteristics at the algorithmic level before the synthesis step. The mathematical framework is the basis for producing correct-by-construction architectures that need no further verification. An approach for transforming a system described in the form of linear recursive equations to digital systems has been presented [23]. Another approach based on temporal logic as a mathematical framework to produce n-dimensional systolic arrays [24, 25], and a formal language (ASL) based on μ-recursive algorithms have also been presented [26, 27] in the literature.

A formal synthesis system that uses a tool called Lambda has been introduced by Bombana et al. [28]. This Lambda tool was integrated into the ItaIel design system using EDIF and VHDL. Another work on formal synthesis for self-timed circuit design based on a compact event model was introduced by Kishnevsky et al. [29]. In this tool, the formal synthesis procedure is considered as a set of equivalent transformations from some initial specifications. This approach, however, can only be used to produce self-timed circuits.

The formal synthesis framework is presented in this paper accepts high-level descriptions in ASL and produces structural descriptions of the intended design in RSL. Figure 1 illustrates all the processing steps of this synthesis procedure. The procedure is not dedicated to any application specific domain or architecture. However, for a class of digital systems such as DSP circuits, efficient VLSI layouts are produced.

3. ASL PRIMITIVES

The ASL is based on μ-recursive functions as a framework. Recursion is a natural way for defining any computable function, for example, the set of natural numbers

\[ N \text{ can be defined starting from zero and adding one recursively. This can be stated formally as follows:} \]

\[
N(0) = 0 \\
N(N+1) = N(N) + 1
\]

The importance of recursion is not limited to specifying functions, but it can be used to prove most properties of algorithms by inference, which is called proof by induction. In ASL, a given algorithm is represented using a limited number of constructs. Although the constructs are primitive, complex constructs can be developed through a cell library. ASL consists of initial functions and operations. The initial functions are very primitive functions. The following three initial functions are used:

1. The zero function \((\xi())\) which returns zero.
2. The projection function \((\eta(\arg_1, \arg_2, \ldots, \arg_n))\) which chooses an argument \(i\) from \(n\) arguments.
3. The successor function \((\lambda(n))\) which increments the input by one.

The three operations that are used to construct larger functions from smaller ones are:

1. Composition.
2. Recursion.
3. μ-recursion.

The complete details of ASL are given elsewhere. Some of the characteristics of the ASL are:

1. It is simple and semantically well characterized.
2. It permits writing specifications using a few pre-defined objects and concepts.
3. It is a suitable tool for formal synthesis.
4. It supports a hierarchical synthesis methodology.
5. It supports formal verification of the liveness and safety properties of a design.
6. It is complete.

3.1 Structural synthesis from ASL

A formal mapping methodology is developed to transform an algorithm represented in ASL to a specific realization language termed RSL. RSL specifies the components and connectivity of a digital architecture that realizes the algorithm. Every construct in ASL has an isomorphic representation in RSL, which is the basis for the automated transformation. Details of the transformation algorithms for mapping from ASL to RSL can be found elsewhere (also see Appendix A). In the following section we give a brief overview of RSL.

4. OVERVIEW OF RSL

The system under consideration accepts ASL as an input and produces an intermediate form RSL which acts as the
specification for the back-end. This back-end produces a VLSI layout from the RSL specification. The specification would have three initial functions, and three operations, which can be applied in a certain sequence to express any computable function or hardware. In order to build such a system, it is required that the intermediate representation RSL be transformed into layouts. To achieve this, hardware of each initial function is synthesized and stored in a cell library. Details of the RSL language and the transformation procedure from ASL to RSL are given in Appendix A.

The approach used for the implementation of RSL is illustrated in Figure 2. It shows that the RSL specifications of a given circuit is translated into logic level models. The formal cell library contains logic/gate level models of all the basic functions of RSL. Modularity and design extendibility to any desired word length are taken into consideration while designing basic cells. This logic level model of the given circuit is then simulated using a gate level simulator. The functionally correct model is then given to the input of the physical design sub-system which generates the final VLSI layout. This physical design system uses a cell library of pre-designed standard cells.

4.1 Formal logic cell library design

As mentioned earlier, any synchronous digital system can be expressed using the basic functions and constructs of ASL, viz. zero, projection, successor, composition, recursion and μ-recursion. Corresponding to each construct in ASL is a construct in the realization specific language (RSL) that must be mapped to hardware equivalent modules. For each RSL primitive, the library contains a logical description (at the gate/transistor level). Similar to the construction of larger functions at the ASL level, larger logic modules are constructed using cells of initial primitives in the library. The synthesized logic circuits are stored in the library for later use. Therefore, for ease of expandability, they are made modular so that they can be easily connected to build cells of larger functions. Also the design accommodates varying word lengths. For example, an n-bit successor function is easily made by cascading n 1-bit successor units. Thus the cell library designed consists of modular building blocks of logic level macros with the above characteristics. Each primitive logic module is carefully synthesized. The approach used for making the formal cell library is shown in Figure 3.

For verification by simulation, the basic functions, which are zero, projection and successor, are modelled at the logic level and expressed using a netlist language.
The logic level netlist is then translated into a transistor-level circuit using a tool called netlist and this transistor-level circuit is then translated into a binary file using a translation tool called precim\textsuperscript{32}. This binary file is required for simulation by rsl, a switch level simulator\textsuperscript{32}. This simulation will verify the correctness of translation. Layouts of these functions are then made using a layout synthesis system such as OASIS and Magic\textsuperscript{33-35}. Circuit from layouts are extracted and simulated to verify the functional correctness of layouts. These stored initial functions are used in the definition and synthesis of larger functions.

Clocking strategy
A two-phase non-overlapping clocking scheme is used in the design of all the cells (or sub-circuits). Input is loaded during $\phi_1$ and the output is obtained during $\phi_2$. The large modules are made by using the primitive functions. Every cell has an input Control and an output Ready. The input signal Control is used to signal the start of operation and the output line Ready is used to indicate that the circuit has finished its operation. The operation of individual elements is data driven. The cells are connected in such a manner that the Ready of one is connected to the Control of the next cell. The cell starts its function when its Control gets some signal from the Ready of the previous cell.

5. LOGIC SYNTHESIS

We now describe the implementation of the three primitive functions which are used to implement larger functions.

5.1 Zero function
The zero function returns the value zero and has no arguments. A register is used to implement the function which is initialized with the value zero ($0^n$).

5.1.1 Projection function
The projection function is used to choose an argument $i$ from $n$ arguments. A multiplexer is used to perform the projection function. $N + 1$ registers are used to store the arguments. An input line Control is used to initiate the multiplexer operation, and an output line Ready is used to indicate that the circuit's operation has been completed.

Figure 4 shows the hardware of projection function. It is a simple 2-to-1 line multiplexer constructed using three transmission gates. There is an input selection line sel to select one of the two inputs i.e. in0 and in1. The output out is obtained by the enable line Control. Larger units are similarly constructed using only transmission gates.

5.1.2 Successor function
The successor function is basically an incrementer that takes an input $n$ and produces the output $n + 1$. An adder and two (dynamic) registers are used to model the successor function in RSL. One of the registers stores the argument $n$ and the other the value one. The operation is executed in one clock cycle. The block diagram of a 1-bit successor cell is shown in Figure 5.

The successor unit in our library is designed as an incremenator and can also be used as an up-counter by feeding back the output to one of its two inputs. Load/Count control inputs are available to accomplish the necessary function. Input is loaded during $\phi_1$ and the output is obtained during $\phi_2$. The cascading of successor units can be done by connecting the andout output of a unit to the andin input of the adjacent unit. The hardware model of the successor function is shown in Figure 6.

6. FORMAL CELLS OF LARGER FUNCTIONS

Larger functions are made by using the basic functions. For example, the successor function can be used to build an adder, and adder can be used to build a multiplier. The adder and multiplier can be used to synthesize an inner product cell, which can then be used to implement say a matrix-matrix multiplier. We will first discuss the details of the Add unit.

6.1 Add unit
The Add unit is used to add two arguments. Addition of $n$ and $m$ is modelled in ASL by incrementing $n$ times the argument $m$. 
is done in a recursive fashion. The successor function which is connected to one of the inputs of comparator is initialized with an input of zero. At the same time the other successor function is loaded with one of the two arguments to be added, say \( n \). The other argument, say \( m \) of the adder, is given to the second input of the comparator. The output \( \text{Ready} \) of the comparator becomes high only when both of its inputs become equal. Both of the successors start incrementing simultaneously and increment \( m \) times. When the output value of the successor connected with the comparator reaches the value equal to \( m \), \( \text{Ready} \) goes high and both the successors stop incrementing as the \( \text{Ready} \) is connected to the \( \text{Control} \) of both the successors. Evidently the successor function loaded with the initial value of \( n \) gives the final result as \( m + n \) (see Appendix B).

### 6.2 Pro unit

In ASL, multiplication of two numbers \( n \) and \( m \) is modelled as addition of \( n \) to itself \( m \) times. The unit \( \text{Pro} \) is used to multiply two arguments.

The direct block diagram representation of the \( \text{Pro} \) function in RSL is shown in Figure 8. It is implemented using one successor function, one comparator, one \( \text{Add} \) unit and some logic gates as control circuitry. The multiplication is done in a recursive manner. Suppose \( m \) is to be multiplied with \( n \). The successor function which is connected to one of the comparator's input is initialized with an input of zero. The argument \( m \) is given to the second input of the comparator. The unit \( \text{Add} \) is loaded with two arguments zero and \( n \). The unit \( \text{Add} \) adds \( n \) to an empty register, \( m \) times, giving the final result \( m \times n \) (see Appendix B).

### 6.3 Inner-Product unit

The inner-product unit is used to multiply two arguments and add the result with the third argument. Suppose \( a, b \) and \( c \) are three inputs to the inner-product unit and \( d \) is the output. The function of the cell is given by:
The direct block diagram representation of inner-product unit in RSL is shown in Figure 9. It is implemented using one Add circuit, and one Pro circuit, and some logic gates. The Pro multiplies two arguments in a recursive manner and the Add unit adds the third argument to its result (see Appendix B).

7. LAYOUT SYNTHESIS

In the previous section, we discussed the development of the logic level formal cell library. This section deals with the layout design methodology of the cells stored in the formal cell layout library, as well as the layouts of the formal cells which use the basic cells in their definition.

Cell layout design methodology

In a cell based VLSI layout synthesis system, the heart of the physical design unit is the cell library. The approach for such a synthesis system is illustrated in Figure 10. The logic level model, using a formal logic cell library, of the given circuit is fed to the physical design system. Two sub-systems of the physical design system are used. The first sub-system is used for placement and routing using a standard cell library. It places the cells and performs the global and detailed routing in a plane to minimize the layout area. The output of the first sub-system is used by the other sub-system. It assembles the final physical layout. The circuit is then extracted and simulated to verify that the hardware description of the primitive function under design performs the intended function.

Physical design sub-system

Layouts are made using the layout design environment Open Architecture Silicon Implementation Software (OASIS)\textsuperscript{33, 34}. A number of design tools are integrated into the OASIS system. It is a cell-based system for IC design. The tools integrated into the OASIS system have been developed to automatically translate high-level descriptions or netlist similar to those designed for RSL, into testable physical layouts, using predesigned standard cells.

One of the premises of the OASIS system is the modularity of the software. New, improved algorithms can be easily substituted in place of the old ones. The entire system is controlled with a single data flow supervisor program to assure data consistency is maintained at all stages of the design. The data flow supervisor is template-driven: the templates used in OASIS can be easily expanded to support additional software tools, thus providing the desired openness of the system.

Standard cell approach

The layouts produced with the OASIS system utilize standard cells of a uniform height. The cells are placed in an array of horizontal rows and all interconnection of signal nets are made by channel routing in the space between the adjacent rows. All connections of the power nets, VDD and GND, are made by abutting the cell horizontally. Signal nets connecting cells belonging to non-adjacent rows cross the intervening rows of the cells by utilizing feed-through pins (vertical strips of metal running from the top to bottom of a cell) built into some cells, or by inserting feed-through cells (cells consisting solely of a single feed-through) whenever appropriate. A set of scalable CMOS cells compatible with the 2µ SCMOs technology are used. Cell signal input and output ports are made in the second layer of metal, while the power net (VDD and GND) ports are made in the first layer of metal.

The VDD and GND power nets from each row of cells are connected together using power rails running vertically through the entire height of the layout.

Placement

The placement and routing in OASIS is done by a layout
subsystem of OASIS called VPNR (Vanilla Place and Route). The tools comprising VPNR create physical layouts automatically from netlist descriptions of logic circuits, using a library of pre-designed standard cells of uniform height. The goal of VPNR is to place cells and perform global and detailed routing of interconnections in a plane so as to minimize the layout area. VPNR is usually invoked at the end of the entire design process after the design has been simulated and verified to implement the desired functionality. Occasionally, the layout may be generated in the early stages of the design to obtain an estimate of the area taken up by the circuit. VPNR employs placement and global routing algorithms based upon the quadtree paradigm. The combined placement and global routing program receives a netlist of standard cells, partitions it into four quadrants (top-left, top-right, bottom-left and bottom-right), and processes each quadrant in the same manner until each quadrant contains one cell row in the vertical direction. Partitioning is accomplished and directed by approximate global routing.

Routing
After the positions of the cells in rows are determined, the algorithm constructs a minimum spanning tree for each net, finds the exact crossing locations for nets that need to cross the cell rows, inserts feed-through cells if necessary, and assigns sub-nets in channels. The nets are processed sequentially, and the routing of each net takes into account all nets routed previously. After all nets are routed, the global router prepares the data for detailed channel routing. The channel router determines where to put the wires so that the resulting layout occupies the least amount of area. VPNR provides a choice of two channel routers: a greedy router and a left edge based router with channel compaction.

Simulation and verification
The entire purpose of this phase is to verify that the hardware description of the basic cells under design performs the intended function. The layouts made by Magic can be extracted and simulated using a simulation tool called irsim. There is, however, no need to verify the correctness of the entire design, which is supposed to be correct by construction.

Layouts of the different units discussed in the previous sections are synthesized using layout design environment presented above. The VLSI layouts of cells are implemented using SCMOS technology.

7.1 Layouts of formal matrix-matrix multipliers
An example of formal matrix-matrix multipliers is presented as an application of the cell library. Three different types of architectures, i.e. (1) simultaneous recursion, (2) recursion with respect to several variables and (3) recursion with fixed number of nestings, are used. Each architecture accepts two matrices as input, and produces a third matrix as output. The ASL and RSL specifications of all the three types of multiplier are given elsewhere.

In this section we first present the high-level subroutine of a matrix-matrix multiplier which uses simultaneous recursion on inner-product units. The multiplication is done recursively as described below. Suppose A and B are the two input matrices and C is the output matrix.

matrix-multiplication (A, B, C)
begin
for i = 1 to n
for j = 1 to n
begin
C_{i,j,0} = 0
for k = 1 to n
C_{i,j,k} = C_{i,j,k-1} + A_{i,k} * B_{k,j}
next k
next i
end

The above algorithmic description is translated into an ASL description as follows:

C_{1,i}(A_{1,j}, B_{1,j}, 0) = \xi(i)
C_{n,n}(A_{n,k}, B_{n,k}, 0) = \xi(n)
C_{1,i}(A_{1,k}, B_{1,k}, (K) = inner \cdot product(A_{1,k}, B_{1,k}, C_{1,i}(A_{1,k}, B_{1,k}, K - 1))
C_{n,n}(A_{n,k}, B_{n,k}, (K) = inner \cdot product(A_{n,k}, B_{n,k}, C_{n,n}(A_{n,k}, B_{n,k}, K - 1))

The architecture of the formal matrix-matrix multiplier, using simultaneous recursion, consists of n² inner-product units. Figure 11 shows the hardware model. This type of architecture gives output in a serial manner. Figure 12 shows its VLSI layout.

The formal matrix-matrix multiplier using recursion with respect to several variables, is implemented by recursion construct on inner-product units. The architecture consists of n multiplication units and 1 adder unit. Figure 13 shows the hardware model of this type of matrix-matrix multiplier. Figure 14 shows the VLSI layout of a matrix-matrix multiplier using recursion with respect to several variables.

The architecture of formal matrix-matrix multiplier using fixed nesting recursion consists of n inner-product units. Figure 15 shows the hardware model of this type of matrix-matrix multiplier and Figure 16 shows its VLSI layout. The ASL and RSL descriptions of the above two matrix-matrix multipliers are given elsewhere.

Table 1 shows the number of devices and layout area of various designs for an 8-bit data bus. Table 2 shows the number of clock cycles required by these units. It can be observed that the area and time are high as compared to the that of designs by non-formal methods. It is the price paid for the functionally correct hardware made by formal techniques.
8. CONCLUSION

In this paper, we present the structure of a formal high-level synthesis system. A formal cell library to support high-level synthesis of digital systems was also presented. The cell library introduced consists of both logic level and layout level cells directly mapped from primitives. It supports a hierarchical design methodology and can be automatically translated to layouts.

Large digital systems can be easily modelled using the formal primitives both at the ASL level, and also at the RSL level. An automated procedure is used to transform ASL representation into a specific realization. The realization format is based on representing architectures using
RSL. Algorithms of linear order complexity are used for transformation from ASL to RSL. Designs expressed using the ASL primitives can be formally verified for correctness before synthesis. The lower level verifications such as correctness of the logic circuits, and correctness at the transistor/layout level rely on simulators.

Extraction of control and data paths of digital systems expressed in ASL is easy. The execution of various functions is based on data flow. Interaction between various units is based on a simple protocol.

A number of digital systems were modelled at the ASL level, translated to RSL and their VLSI layouts synthesized using standard-cell design methodology. As an example the design of matrix-matrix multipliers were pre-
ACKNOWLEDGMENTS

We acknowledge King Fahd University of Petroleum and Minerals for all support provided for this work.
Table 2: Number of clock cycles required by the 8-bit units before the final results are obtained

<table>
<thead>
<tr>
<th>Unit</th>
<th>No. of Arguments</th>
<th>Arguments</th>
<th>Time (clock cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Add</td>
<td>2</td>
<td>m, n</td>
<td>max(m, n)</td>
</tr>
<tr>
<td>Pro</td>
<td>2</td>
<td>m, n</td>
<td>m x n</td>
</tr>
<tr>
<td>inner-product</td>
<td>3</td>
<td>a, b, c</td>
<td>(a x b) + max(a x b, c)</td>
</tr>
<tr>
<td>multiplier1</td>
<td>2(n x n)matrices</td>
<td>A x n x n</td>
<td>max(C1, ..., Cn) where C1 = (a11 x b11 + a12 x b21) +</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>max(a11, b11, a12, b21)</td>
</tr>
<tr>
<td>multiplier2</td>
<td>2(n x n)matrices</td>
<td>A x n x n</td>
<td>(C11 + ... + Cnn) where C11 = 2max(a11, b11, a12, b21)</td>
</tr>
<tr>
<td>multiplier3</td>
<td>2(n x n)matrices</td>
<td>A x n x n</td>
<td>(C11 + ... + Cnn) where C11 = (a11 x b11) + (a12 x b21)</td>
</tr>
</tbody>
</table>

used to represent units that will be further expressed at lower levels and also to represent temporary variables used in the computation. While the identifiers starting with lower case letters are used to represent basic functions as well as constants.

When a register α is initialized to a value β, then this is expressed as \( \rho^\beta_\alpha \). For example, the expression \( \rho^{|suc(I)|}_\alpha \) means that the unit has the register I connected to its input, this register is initialized with the value 2. The INIT statement is an RSL expression that can be used separately to express that register α is initialized to value β as follows:

\[
\text{Init}(\alpha, \beta)
\]

Another version of \( \text{Init} \) can be used to denote the initialization of more than one register at the same time; this is represented as:

\[
\text{Initp}(\alpha_1, \beta_1; \alpha_2, \beta_2; \ldots; \alpha_n \beta_n)
\]

The above statement means that \( n \) registers are initialized in parallel such that register \( \alpha_1 \) is initialized with the value \( \beta_1 \), register \( \alpha_2 \) is initialized with the value \( \beta_2 \) and so on. It is assumed that all registers are synchronized using a global clock.

Constant values are expressed using registers. If it is required to have a constant value \( c \) in the circuit, then this is represented as \( c_\alpha \).

The representation of functional units and the control section is presented in the following section.

Transformation from ASL to RSL

The transformation approach is based on using a one-to-one mapping procedure. The algorithm takes an ASL representation and transforms each ASL construct into an equivalent RSL representation.

1. Zero function returns the value and has no arguments. A register is used to realize the function which is initialized with the value zero. The RSL representation is as follows:

\[
\text{Result} = \text{zero}
\]
\[
\text{zero}_\text{Ready} = \text{zero}_\text{Control}
\]

2. Projection function is used to choose an argument \( i \) from \( n \) arguments. A \( (k, \lfloor \log k \rfloor) \) multiplexer is used to perform the projection function. An input line Control is used to determine the start of the multiplexer operation, and an output line Ready is used to indicate that the circuit has finished its operation. This is represented in RSL as follows:

\[
\text{Result} = \max(\text{arg}_1, \ldots, \text{arg}_n, \text{#i})
\]
\[
\text{max}_\text{Ready} = \max_\text{Control}
\]

3. Successor function is used to increment the input argument by one. An adder is used to perform the successor function. A control input signal and a ready output signal are used to determine the start and end of the operation, respectively, as in the projection function. This is represented by the following RSL code:

\[
\text{Result} = \text{suc}(n)
\]
\[
\text{suc}_\text{Ready} = \text{suc}_\text{Control}
\]

4. Composition function representation in RSL is as follows:

\[
\text{Result} = \text{Comp}(x_1, \ldots, x_n, y)
\]
\[
\text{Comp}_\text{Ready} = \text{And}(x_1^\text{Ready}, \ldots, x_n^\text{Ready})
\]

Notice that \( \text{And} \) represents the logical AND, and it is considered as a Basic function and \( \text{Comp} \) is function \( z \) in the recursion construct.

Recursion function is represented in RSL as follows:

\[
\text{computer systems science & engineering}
\]
Result = eq? (arg1, arg2)

The RSL representation of the circuit for recursion is as follows:

\[
Initp(0, m; 1, \text{arg1}, 2, \text{arg2}; \ldots; n, \text{argn})
\]  
\[
suc\text{control} = g(\text{argn})\text{read}
\]  
\[
I = \text{p}^{m+1} \text{suc}(I)
\]  
\[
\text{Ready} = \text{eq}?(I, m)
\]  
\[
\text{Result} = \text{comp}((\text{arg1}, \text{p}^{m+1} \text{Result}))
\]

(A1)  
(A2)  
(A3)  
(A4)  
(A5)

The first equation indicates that \(n+1\) registers are initialized by the arguments, \(n\) of them with the arguments of the \(y\) component and a register for the constant \(m\). The second equation means that the unit \(\text{suc}\), which is a basic function, has its input control connected to the ready output of the unit computing \(g(n)\) to be sure that \(I\) is not incremented until \(g(n)\) is computed. The third equation is used to represent the fact that \(I\) is incremented every clock cycle using the suc unit, and \(I\) is initialized to the value 1 using the register number \(n+1\). The fourth equation determines the end of the operation when \(I\) reaches the value \(m\). The fifth equation means that \(y\) has \(n\) inputs from the \(n\) registers, one output from register number \(n+2\) which is initialized to the value \(g(n)\), and an input \(I\) which is the output of the suc unit.

6. \(\mu\)-Recursion function is represented in RSL as follows:

\[
Initp(0, n; 1, \text{arg1}, 2, \text{arg2}; \ldots; n, \text{argn})
\]  
\[
\text{Result} = \text{p}^{m}\text{succ}(\text{Result})
\]  
\[
\text{Ready} = \text{eq}?(0, g(\text{arg}, \text{Result}))
\]

The first equation is used to initialize the \(n+1\) register with \(y\) arguments and a zero value. The second equation is used to indicate that the unit \(\text{suc}\) has an input from a register initialized with a value 0 and the output of the \(\text{suc}\) is the \(\text{Result}\). The third statement is used for comparing \(\text{suc}\) output with \(m\).

From the previous transformation method we see that each construct in ASL has an isomorphic representation in RSL. The equivalence between ASL and RSL represents the base for the automatic transformation procedure for any algorithm specified using ASL to RSL.

APPENDIX B: FORMAL DESCRIPTION OF DIFFERENT UNITS

ASL Representation

- Multiplication unit code:
  \(\text{pro}(n, 0) = \xi()\)

- Addition unit code:
  \(\text{add}(\text{arg}, \text{arg}, \text{arg})\)
  \(= \text{add}(\text{arg}, \text{arg}, \text{arg})\)
  \(\text{pro}(n, m + 1) = \text{add}(\text{arg}, \text{arg}, \text{arg})\)

- Inner-product cell code:
  \(\text{inner_product}(a, b, c, d)\)
  \(= \text{pro}(\text{arg}, \text{arg}, \text{arg}, \text{arg})\)
  \(\text{add}(\text{arg}, \text{arg}, \text{arg}, \text{arg})\)
  \(= \text{add}(\text{arg}, \text{arg}, \text{arg}, \text{arg})\)
  \(\text{inner_product}(a, b, c + 1)\)
  \(= \text{add}(\text{arg}, \text{arg}, \text{arg}, \text{arg})\)

RSL Representation

The RSL specification is divided into three parts as the ASL specification.

- The multiplication unit code:
  \(\text{Result} = \text{zero} | \text{Rule}_{1}\)
  \(\text{Result} = \text{max}((\text{arg}, \text{arg}, \text{arg})^1) | \text{Rule}_{2}\)
  \(\text{Result} = \text{add}(\text{Result}, \text{Result}^1, \text{add}) | \text{Rule}_{3}\)
  \(\text{Initp}(0, m, 1, \text{arg1}, 2, \text{arg2}; \ldots; n, \text{argn})\)
  \(\text{suc}\text{control} = \text{zero}\text{read} | \text{Rule}_{4}\)
  \(I = \text{p}^{m}\text{succ}(I) | \text{Rule}_{5}\)
  \(\text{Ready} = \text{eq}?(I, m) | \text{Rule}_{6}\)
  \(\text{Result} = \text{pro}(n, 1, \text{p}^{m}\text{result}, \text{add}) | \text{Rule}_{7}\)

Comparing the ASL and RSL specifications we see that statement 1 in RSL is corresponding to the first statement in ASL and is obtained using rule 1 in the transformation procedure (explained in Appendix A). Statements 2–4 correspond to the second statement in ASL and are obtained by applying rule 5 twice and rule 4 once, and statements 5–8 correspond to the third ASL statement and are obtained by applying rule 5 once.

- The addition unit code:
  \(\text{Result} = \text{max}((\text{arg}, \text{arg})^1) | \text{Rule}_{1}\)
  \(\text{Result} = \text{max}((\text{arg}, \text{arg}, \text{arg})^1) | \text{Rule}_{2}\)
  \(\text{Result} = \text{Q}(\text{Result}, \text{add}) | \text{Rule}_{3}\)
  \(\text{Initp}(0, m, 1, n)\)
  \(\text{suc}\text{control} = \text{result}\text{read} | \text{Rule}_{4}\)
  \(I = \text{p}^{m}\text{succ}(I) | \text{Rule}_{5}\)
  \(\text{Ready} = \text{eq}?(I, m) | \text{Rule}_{6}\)
  \(\text{Result} = \text{add}(n, I, \text{p}^{m}\text{result}, \text{add}) | \text{Rule}_{7}\)

- The inner-product cell code:
  \(\text{Result} = \text{max}((\text{arg}, \text{arg}, \text{arg}, \text{arg})^1) | \text{Rule}_{1}\)
\[ \text{Result}_2 = \text{mu}_3(\text{arg}_1, \text{arg}_2, \text{arg}_3, \text{arg}_4) \]  \hspace{1cm} \text{Rule}_2 \hspace{1cm} (B19)

\[ \text{result}_3 = \text{temp}(\text{Result}_1, \text{Result}_2) \]  \hspace{1cm} \text{Rule}_4 \hspace{1cm} (B20)

\[ \text{Result}_1 = \text{mu}_4(\text{arg}_1, \text{arg}_2, \text{arg}_3, \text{arg}_4) \]  \hspace{1cm} \text{Rule}_2 \hspace{1cm} (B21)

\[ \text{Result}_3 = \text{add}_1(\text{Result}_1, \text{Result}_2) \]  \hspace{1cm} \text{Rule}_2 \hspace{1cm} (B22)

\[ \text{Init}(0, \text{m}, \text{i}) \]  \hspace{1cm} \text{Rule}_5 \hspace{1cm} (B23)

\[ \text{src\_Control} = \text{Result}_2 \text{\_Ready} \]  \hspace{1cm} \text{Rule}_5 \hspace{1cm} (B24)

\[ \text{I} = \text{p}_i, \text{src}(i) \]  \hspace{1cm} \text{Rule}_5 \hspace{1cm} (B25)

\[ \text{Ready} = \text{eq}(i, \text{m}) \]  \hspace{1cm} \text{Rule}_5 \hspace{1cm} (B26)

\[ \text{Result} = \text{inner\_product}(n, 1, \text{p}_i, \text{res}_0, \text{add}_1) \]  \hspace{1cm} \text{Rule}_5 \hspace{1cm} (B27)

REFERENCES

9. Mesztenyi, K 'Computer design language, simulation and boolean translation'. Technical Report 68-72, Department of Computer Science, University of Maryland (June 1968)
20. Gireze, E F and Knight, J P 'An Ada to standard cell hardware compiler based on graph grammars and scheduling', ICCD '84, New York (October 1984) pp 726-731
34. MCNC's Center for Microelectronics, Open Architecture silicon Implementation Software, release 2.0 edition (December 1992)
Laboratory (September 1990)

Rose, J E 'Greedy Algorithms for Wiring in VLSI',
Master's thesis, Department of Computer Science,
North Carolina State University (1985)