High Level Synthesis from *Sim-nML* Processor Models [*]

Souvik Basu
souvik@gdatech.co.in
GDA Technologies Limited, Bangalore, India

Rajat Moona
moona@iitk.ac.in
Dept. of CSE, IIT Kanpur, India

Abstract

The design of modern complex embedded systems require a high level of abstraction of the design. The Sim-nML[1] is a specification language to model processors for such designs. Several software generation tools have been developed that take ISA specifications in Sim-nML as input.

In this paper we present a tool Sim-HS that implements high level behavioral and structural synthesis of processors from their ISA specifications in Sim-nML. Behavioral Sim-HS transforms Sim-nML specifications of a processor to the corresponding behavioral Verilog model that is suitable for fast functional simulation. Structural Sim-HS generates structural synthesizable Verilog processor model from its Sim-nML specifications.

1 Introduction

Today VLSI technology is providing multiple million gates per chip to support the design of several programmable and non-programmable processor cores. High level synthesis (HLS) technology enables the fast error free design of such complex circuits. Among all active researches in this area of HLS, one important research direction is to synthesize programmable processors from their ISA specifications. In this work, we have developed a HLS system, *Sim-HS*, that generates behavioral and structural Verilog descriptions of the processors from the *Sim-nML* [1] processor ISA specifications. The generated structural descriptions are accepted by various logic synthesis tools.

2 Sim-nML, A brief overview

Sim-nML [1] is a behavioral specification language that is used to specify the instruction set architecture of a programmable processor. The language is an extension of nML [11] machine description formalism, for the purpose of efficient simulation. The hierarchical processor models in *Sim-nML* are specified using *attribute grammar*, where the semantic actions of the instructions are distributed over the whole specification tree. Thus, the common behavior of a class of instructions is captured at the top level and the specialized behaviors are captured at the subsequent lower levels. Root node of the specification tree is named *instruction*. The specifications have two types of productions, *and* and *or* productions. Each of these productions can be *op-production* or *mode-production*, which specify the processor instructions and addressing modes respectively. There are four pre-defined attributes for the *and-production– syntax, image, action* and *uses*. The *syntax* and *image* attributes captures the textual assembly language syntax and bit image of the instructions. The *action* attribute captures the operations performed by the instructions, including any side effects. The *uses* attribute captures the resource use model for the instruction that can be used to determine the timing and execution relationship with other instructions. The language provides *'reg'*, *'mem'* and *'var'* data types that are used to specify the registers, memories and temporary variables respectively.

An integrated development environment has been developed around *Sim-nML* processor specification language at IIT Kanpur. This includes the tools for generation of assemblers [3], disassemblers [4], compiler back-ends [5, 6], functional simulators [7] [8], cache simulators [9] etc.

An example of a small processor that supports ALU operations, memory load-store operation and branch operation is given below. The processor operations are specified using the *op-productions*. Their are four ALU operations in the example; AND, ADD, SUB and MUL. The processor has four addressing modes specified using the *mode-productions*. These are the register, memory indirect, memory post increment and memory absolute addressing modes.

```
type word = card(16)
type absa = card(9)
type disp = int(4)
type off = int(6)
mem PC[1,word]
```

[*] This work was primarily supported by Cadence Design System (India) Pvt. Ltd. and done at Cadence Research Center at IIT Kanpur, India. http://www.cse.iitk.ac.in/sim-nml

```
mem R[16,word]
mem M[65536,word]
var L1[1,word]
var L2[1,word]
var L3[1,word]
mode register(i:card(4)) = R[i]
   syntax = format(”R%s”, i)
   image = format(”%4b”, i)
mode memory = ind | post | abs
mode ind(r:register, d:disp) = M[r+d]
   update = {}
   syntax = format(”@%s(%d)”, r.syntax, d)
   image = format(”0%4b%4b0”, r.image, d)
mode post(r:register, d:disp) = M[r+d]
   update = {r = r + 1;}
   syntax = format(”@%s++(%d)”, r.syntax, d)
   image = format(”0%4b%4b1”, r.image, d)
mode abs(a : absa) = M[a]
   update = {}
   syntax = format(”%d”, a)
   image = format(”1%9b”, a)
op instruction( i : instr )
   syntax = i.syntax
   image = i.image
   action = {
     PC = PC + 1;
     i.action;
   }
op instr = move | alu | jump
op move(lore:card(1), r:register, m:memory)
   syntax = format(”MOVE%d %s %s”, lore, r.syntax, m.syntax)
   image = format(”0%1b%4b%10b”, lore, r.image, m.image)
   action = {
     if ( lore ) then r = m;
     else m = r;
     endif;
     m.update;
   }
op alu(s1:register, s2:register, d:reg, a:aluop)
   syntax = format(”%s %s %s %s”, a.syntax, s1.syntax, s2.syntax,
d.syntax)
   image = format(”10%4b%4b%4b%2b”, s1.image, s2.image,
d.image, a.image)
   action = {
     L1 = s1; L2 = s2; a.action; d = L3;
   }
op jump(s1:register, s2:register, o:off)
   syntax = format(”JUMP %s %s %d”, s1.syntax, s2.syntax, o)
   image = format(”11%4b%4b%6b”, s1.image, s2.image, o)
   action = {
     if ( s1 >= S2 ) then PC = PC + o;
     endif;
   }
op aluop = and | add | sub | shift
op and() syntax = ”and” image = ”00” action = { L3 = L1 & L2; }
op add() syntax = ”add” image = ”10” action = { L3 = L1 + L2; }
op sub() syntax = ”sub” image = ”01” action = { L3 = L1 − L2; }
op mul() syntax = ”mul” image = ”11” action = { L3 = L1 * L2;
}
```

3 Related Works

Several processor ISA specification languages have been designed that can be used for hardware-software co-design.

ISDL [10], developed at MIT LCS is a programmable processor ISA specification language that is used to describe the behavior of a processor using attribute grammar. A special emphasis has been given for VLIW architecture based processor specifications. A synthesis tool HGEN has been developed that generates structural synthesizable Verilog code for the underlying VLIW architecture from the ISDL specifications. nML [11] processor instruction set specification language, developed at TU Berlin is an attribute grammar based language used for processor specification. From a nML based processor description, the hardware elements have been generated [12]. The language used in our work *Sim-nML* is derived from the nML language. MIMOLA [13] hardware specification language, developed at University of Dortmund, Germany is used to write structural specification of a programmable processor at low level, exposing several hardware details. Using MSS [14] synthesis tool, hardware can be synthesized from MIMOLA specifications. LISA [15] processor specification language, developed at Aachen University of Technology, Germany is used to specify programmable processors. VHDL hardware models have been synthesized from LISA for four stage pipelined ICORE architecture. ISPS [18] is an ISA specification language developed at Carnegie-Mellon University. CMUDA [19] and System Architect's Workbench [20] are the HLS systems that take the ISPS specifications as input and generate hardware. TRS [21], developed at MIT LCS is used to describe hardware at the micro-architecture level in functional language form. Hardware synthesis compiler TARC has been developed that takes the concurrent TRS specifications and generates synthesizable Verilog code.

There are several other languages to specify processor ISA, like SLED [16], EXPRESSION [17] etc. Though there is a potential of processor synthesis from these languages, no such work has been reported in the literature.

4 Behavioral *Sim-HS*

The overall design of the *Sim-HS* HLS system consists of two parts, the front-end and the back-end (figure 1). The same front-end is used for both behavioral and structural *Sim-HS*.

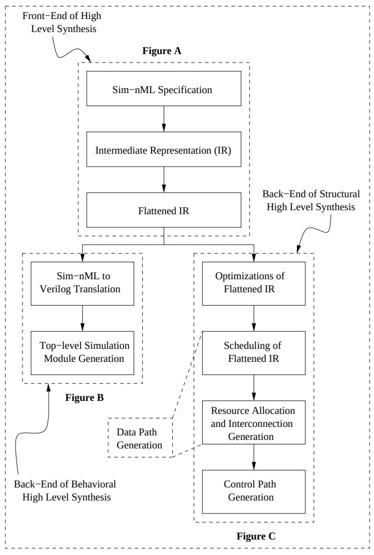
Front−End of High Level Synthesis
Figure A
Sim−nML Specification
Intermediate Representation (IR)
Flattened IR
Back−End of Structural High Level Synthesis
Sim−nML to Verilog Translation
Top−level Simulation Module Generation
Figure B
Optimizations of Flattened IR
Scheduling of Flattened IR
Data Path Generation
Resource Allocation and Interconnection Generation
Control Path Generation
Back−End of Behavioral High Level Synthesis
Figure C

Figure 1. Design of the High Level Synthesis System

4.1 Design of the Front-end of *Sim-HS*

The front-end of *Sim-HS* (figure 1A) takes *Sim-nML* processor specifications as input and produces their flattened representations. First the input specifications are converted to an intermediate representation (IR). Next the information is gathered for all instructions by traversing the the path from the root node to all leaf nodes with proper parameter substitution at all levels.

4.2 Design of the Back-end of Behavioral *Sim-HS*

The back-end of the behavioral *Sim-HS* (figure 1B) takes the flattened IR as input and for each machine instruction, generates the behavioral Verilog code to simulate its action. After this generation, a top level simulation module is generated to facilitate the functional simulation process.

4.3 Implementation of Back-end

All *Sim-nML* variables of *'reg'*, *'mem'* and *'var'* data types are converted to Verilog variables. The scalar variables of *Sim-nML* are translated to Verilog *reg* data types and the *Sim-nML* arrays are translated to Verilog *reg* arrays. An example of variable translations is given below.

SIM-NML: reg B[100,card(32)]   VERILOG: reg[0:31] B[0:99]

The language constructs, like control flow statements are similar in the *Sim-nML action* sequences and Verilog language, and therefore the translation is straight forward. In the generated behavioral model instruction decoding is implemented by Verilog *case* statements. To store the instruction being executed, a new register *IR* is added. The width of *IR* is equal to the maximum width among all the instructions. The parameters of the instructions are identified and used as the bit strings of register *IR*. For this the information in the image attribute is used.

An example translation of an instruction from *Sim-nML* specification of Motorola 68HC11 micro-controller is given below. After execution of the instruction, the *IR* is left shifted by the size of the instruction (and later filled by more instruction bytes from the memory).

Sim-nML Action Sequence

```
op LDAA_Imm(Src : Imm8)
image = format("10000110%s", Src.image)
action = {
        R = Src; CCR<3..3> = R<7..7>;
        if R == 0 then CCR<2..2> = 1;
        else CCR<2..2> = 0;
        endif;
        CCR<1..1> = 0; A = R;
}
```

Behavioral Verilog Code

```
always @(posedge clock) begin
case (IR[0:23]) 24'b10000110XXXXXXXXXXXXXXXX :
begin
        R = IR[8:15]; CCR[3:3] = R[7:7];
        if( (R == 0) ) CCR[2:2] = 1;
        else CCR[2:2] = 0;
        end
        CCR[1:1] = 0; A = R;
        IR = IR << 16;
end
```

In the behavioral module, a simulation clock is also added. As instructions execute, the simulation clock is appropriately incremented.

In the last step of the generation of the behavioral Verilog code for the processor, the Verilog simulation monitor module is added. The simulator monitor module continuously probes the various Verilog variables that represent the external signals on the pins, or the internal signals (figure 2).

5 Structural *Sim-HS*

5.1 Design of the Back-end of Structural *Sim-HS*

The back-end design involves four major steps (figure 1C) – optimizations of the flattened intermediate representation; scheduling of the optimized specifications; resource allocation and interconnection of resources; and finally the control path generation. The data path is generated in scheduling and resource allocation steps.

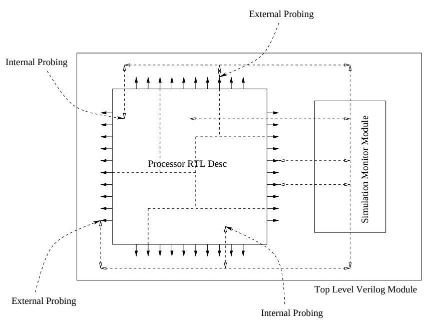

Figure 2. Simulation Monitor Module

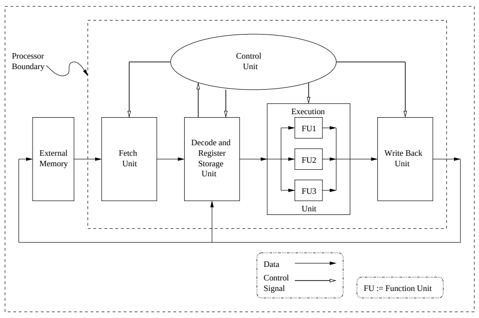

Figure 3. Processor Block Diagram

5.1.1 Architecture of the Synthesized Processor

We synthesize the *Sim-nML* specifications to a non-pipelined processor architecture (figure 3). The different units of the architecture are: fetch unit; decode and register storage unit; execution unit and write-back unit. Execution unit is a collection of several functional units where the input and output ports of the functional units are common to the input and output ports of the execution unit. In our design, the memory is external to the processor core.

5.1.2 Optimizations of the Flattened IR

The actions of each instruction in the *Sim-nML* specifications can be written in a convenient way using temporary variables across the hierarchies. This makes the specification writing easier and elegant looking. Such temporary variables are declared using *var* keyword in *Sim-nML*.

In order to get the area-optimized hardware, it is necessary to minimize the use of *'var'* type temporary variables. Optimizations are performed on each flattened instruction actions. The instructions are assumed to be independent of each other in a non-pipelined processor. Therefore, we do not perform any inter-instruction optimizations. Temporary variable elimination and dead code elimination are two optimizations performed in structural *Sim-HS* that give reasonably good amount of area reduction for the temporary storage units. The *'reg'* and *'mem'* variables represent the state of the processor and the removal of these variables in the process of optimization can change the semantic meaning of the overall ISA specification. Hence, no optimizations are performed around *'reg'* and *'mem'* type variables.

5.1.3 Scheduling of the Optimized Instructions

Scheduling assigns the optimized instruction actions to control steps according to the following scheduling constraints and goal.

- Architecture of the Processor: The non-pipelined multi-cycle architecture of the processor permits only one instruction to be executed at a time.
- Number of Functional Units: In our design only one functional unit of each type is instantiated and is shared among the operations across and within the instructions.
- Number of Port Resources: The number of data and address port resources in the storage and functional units constrains the scheduling of operations inside an instruction. In our implementation, we have assumed one multiplexed read and write port for the memory. The registers are assumed to have two read and one write ports. All functional units have two input and one output data ports.
- Types of Functional Units: A functional unit can perform only one type of operation. Thus, there are no shared functional units like *'multiplier-adder'* etc. in the design. Thus there is one to one correspondence between the operations and functional units.
- Minimization of Storage Area: In the structural design, as the number of *'reg'* and *'mem'* type storage units are predefined, number of these units can not be reduced. Thus, minimization of storage area is done by minimizing the number of each types of functional unit instantiated. However, because of the minimization of number of functional units, extra multiplexers or multiplexers with large number of inputs may be required in the design.

in order to synthesize, operations inside the instruction actions are converted to a sequence of *three-address* code form. From the *three-address* form, control signals are generated to provide data from the registers to the two input ports of the functional units and to take the output of the functional units to a register.

5.1.4 Resource Allocation and Interconnection

After scheduling, all operations are mapped to functional units and all operands are mapped to storage units to generate the hardware model. As the functional and storage unit resources are shared across the instructions, multiplexers are used with appropriate controls for input selection. Multiple destinations are bussed together.

5.1.5 Control Path Generation

After designing the data path, control path elements are instantiated to design the controller. To control the execution of an instruction a sequence of control signals specific to the scheduled operations of an instruction are generated. An instruction decoding unit decodes the instructions from the image of the instruction. After flattening image attribute for an instruction contains a string of 0's, 1's and unknown bits. The image substring containing 0's and 1's is used to identify the instruction and the unknown image substring typically selects the storage units at the execution time.

5.2 Implementation of Back-end of *Sim-HS*

Our implementation generates the Verilog code, which is compliant with the Synopsys Design Compiler. The generated Verilog code is built upon the Design Ware Library [22] [24] components, thus saving effort in rebuilding our own library.

For each *'reg'* and *'mem'* type variables (except main memory) registers and register files are instantiated. For each types of operation, one functional unit is instantiated inside the execution unit. An example of the generated Verilog structural execution unit module is given below. The execution unit contains one instantiation of adder and multiplier functional units each.

```
module Execution_Unit(ExIn0_Mux_O__Execution_Unit_In0_I,
   ExIn1_Mux_O__Execution_Unit_In1_I, Clk, Sel, Execution_
   Unit_O__Execution_Unit_Dmux_I);
parameter                width=32;
parameter                sel_width=1;
input   [width-1:0]      ExIn0_Mux_O__Execution_Unit_In0_I;
input   [width-1:0]      ExIn1_Mux_O__Execution_Unit_In1_I;
input                    Clk;
input   [sel_width-1:0] Sel;
output [width-1:0]    Execution_Unit_O__Execution_Unit_Dmux_I;
reg [width-1:0]    Execution_Unit_O__Execution_Unit_Dmux_I;
wire    [width-1:0]    Out_1;
wire    [width-1:0]    Out_2;
always @(posedge Clk) begin
DW01_add #(width) Add1(.A(ExIn0_Mux_O__Execution_Unit_In0_I),
           .B(ExIn1_Mux_O__Execution_Unit_In1_I),.CI(),
           .SUM(Out_1),.CO());
DW02_mult #(width,width) Mult1(.A(ExIn0_Mux_O__Execution_Unit
           _In0_I), .B(ExIn1_Mux_O__Execution_Unit_In1_I),
           .TC(), .PRODUCT(Out_2));
case(Sel)
   0 : Execution_Unit_O__Execution_Unit_Dmux_I <= Out_1;
   1 : Execution_Unit_O__Execution_Unit_Dmux_I <= Out_2;
endcase
end
endmodule
```

Finally, after the data path and control path generation, the top-level module is generated, which instantiates the registers, register files, multiplexers and interconnects them.

	68HC11	8085
Lines of Codes in Sim-nML Specifications	2947	1419
Total Number of Machine Instruction	210	231
Lines of Codes in Generated Behavioral Verilog	3708	2143

Table 1. Statistics of the Behavioral Synthesis

	PowerPC	8085	68HC11
Sim-nML Specification	508 lines	1419 lines	2947 lines
Sim-HS Generated Code	656 lines	4679 lines	1752 lines
Synthesized Verilog Code	8478 lines	25783 lines	6698 lines
Synthesis Time (without CT)	220 Sec	910 Sec	300 Sec
Synthesis Time (with CT)	780 Sec	1130 Sec	600 Sec
Chip Area in mm^2	0.525	1.05	0.33

Table 2. Statistics of the Structural Synthesis

6 Results and Conclusions

In this work, we have developed techniques to generate behavioral and structural synthesizable Verilog processor models from the *Sim-nML* processor specifications. This method along with the strength of *Sim-nML* based methodology can be used in an integrated way, to generate ASIP and/or other programmable processor hardware and system level software.

The behavioral *Sim-HS* is tested on the Sim-nML specifications of Motorola 68HC11 micro-controller and Intel 8085 microprocessor. The statistics are shown in the table 1. The size of the behavioral *Sim-HS* generated Verilog code is about the same order as that of the corresponding input *Sim-nML* specifications. The synthesized Verilog code is simulated using Cadence Inc.'s Verilog-XL simulator [23] where it ran without any problem.

The structural *Sim-HS* is tested on specifications of full instruction sets of Motorola 68HC11 micro-controller, Intel 8085 microprocessor and a subset of instruction set of PowerPC 603 processor. The statistics for the structural synthesis are shown in the table 2.

The generated Verilog code was logic synthesized using generic 'class.db' library of Synopsis Design Compiler. Time taken for logic synthesis of the generated code with and without the Clock Tree (CT) are also shown in the table 2. It is observed that there is an increase of about 5% in

	Area before CT insertion	Area after CT insertion
8085 Combinatorial Area	12802.00	13263.00
8085 Non-combinatorial Area	20943.00	21943.00
8085 Total Cell Area	**33745.00**	**35206.00**
68HC11 Combinatorial Area	4230.00	4473.00
68HC11 Non-Combinatorial Area	6700.00	6800.00
68HC11 Total Cell Area	**10930.00**	**11273.00**
PPC603 Combinatorial Area	6102.00	6584.00
PPC603 Non-Combinatorial Area	10878.00	11878.00
PPC603 Total Cell Area	**16980.00**	**17562.00**

Table 3. Total Chip Area for some Synthesized Processors in number of cells

the area for all the processor models due to the clock tree insertion.

The chip areas in terms of unit cells (the area of unit cell depends of the lower level technology libraries) for Intel 8085, Motorola 68HC11 and PowerPC 603 are shown in the table 3 separately for both combinatorial and non-combinatorial logic. Assuming that the size of a unit cell (say an inverter) is about $15\mu\text{m}^2$ and 50% of the chip area is used for the routing, the estimated areas of the processors are also shown in the table 2.

References

[1] Moona, R. "Processor Models For Retargetable Tools". *Proceedings of IEEE Rapid Systems Prototyping 2000* June 2000, pp 34–39.

[2] Gajski, D. D., Dutt, N. D., and Wu, A. C.-H. *"High Level Synthesis Introduction to Chip and Systems Design"*. Kluwer Academic Publishers, 1992.

[3] Kumari, S. "An Automatic Assembler Generator for Sim-nML Description Language". Master's thesis, March 2000. http://www.cse.iitk.ac.in/research/mtech1998/9811119.html.

[4] Jain, N. C. "Disassembler Using High-Level Processor Models". Master's thesis, January 1999. http://www.cse.iitk.ac.in/research/mtech1997/9711113.html.

[5] Pogde, P. "Retargettable Code Generation using Sim-nML Machine Description". Master's thesis, May 2000. http://www.cse.iitk.ac.in/research/mtech1998/9811114.html.

[6] Bhattacharya, S. "Generation of GCC Backend from Sim-nML Processor Description". Master's thesis, July 2001. http://www.cse.iitk.ac.in/research/mtech1999/9911149.html.

[7] Chandra, Y. S. "Retargetable Functional Simulator". Master's thesis, June 1999. http://www.cse.iitk.ac.in/research/mtech1997/9711121.html.

[8] Chandra, Y. S., and Moona, R. "Retargetable Functional Simulator Using High Level Processor Models". Thirteenth International Conference on VLSI Design, Calcutta, India, Jan 2000, pp 424–429.

[9] Rajiv, A. R. "Retargetable Profiling Tools and their Application in Cache Simulation and Code Instrumentation". Master's thesis, December 1999. http://www.cse.iitk.ac.in/research/mtech1998/9811117.html.

[10] Hadjiyiannis, G., Hanono, S., and Devadas, S. "ISDL: An Instruction Set Description Language for Retargetability". *In Proceedings of the 34th Design Automation Conference* (June 1997), pp 299–302.

[11] Fauth, A., Praet, J. V., and Freericks, M. "Describing Instruction Sets Using nML (Extended Version).". *Technical report, Technische University at Berlin and IMEC, Berlin (Germany)/Leuven (Belgium)* (1995). ftp://ftp.imec.be/pub/vsdm/reports/retargetable_code_generation/af-edtc95.ps.gz.

[12] Fauth, A., Freericks, M., and Knoll, A. "Generation of Hardware Machine Models from Instruction Set Descriptions". *Proc. IEEE Workshop VLSI Signal Processing, Veldhoven (Netherlands)* (Oct 1993), pp 242–250. http://www.techfak.uni-bielefeld.de/techfak/ags/ti/forschung/publikationen/vlsi-93.ps .

[13] Marwedel, P. "The MIMOLA Design system : Tools for the Design of Digital Processors". *Proc. of the 21th Design Automotion Conference* (1984), pp 53–58.

[14] Marwedel, P. "Matching System and Component Behaviour in MIMOLA Synthesis Tools". *Proc. of the European Design Automation Conference (EDAC)* (1990), pp 146–156.

[15] Zivojnovic, V., Pees, S., and Meyr, H. "LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design". *In Proceedings of 1996 IEEE Workshop on VLSI Signal Processing* (1996). http://www.ert.rwth-aachen.de/Projekte/Tools/LISA/lisa.html.

[16] Raksey, N., and Fernandez. "Specifying Representations of Machine Instructions". *ACM Transaction on Programming Langauges and Systems* (May 1997), pp 492–524. http://www.cs.virginia.edu/ nr/pubs/specifying-abstract.html.

[17] Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., and Nicolau, A. "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability". *Proc. of the Design, Automation and Test in Europe* (1999) pp 485–490.

[18] Barbacci, M. "Instruction Set Processor Specifications (ISPS): The Notion and its Applications". *IEEE Transactions on Computer-Aided Design* (Jan 1981), vol. 30, no. 1, pp 24-40.

[19] Tseng, C. J., and Siewiorek, D. P. "Automated Synthesis of Data Paths in Digital Systems". *IEEE Transactions on Computer Aided Design* (July 1986).

[20] Thomas, D. E., Dirkes, E. M., Walker, R. A., Rajan, J. V., Nestor, J. A., and Blackburn, R. L. "The System Architect's Workbench". Proceedings of the 25th ACM/IEEE Design Automation Conference. (June 1988), pp 337–343.

[21] Hoe, J. C., and Arvind. "Hardware Synthesis from Term Rewriting Systems". *Proc. of VLSI'99"* (December 1999). ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-421a.ps.gz .

[22] Bhatnagar, H. *"Advanced ASIC Chip Synthesis : Using Synopsys Design Compiler and Primetime"*. Kluwer Academic Publishers, 1999.

[23] "OpenBook Reference Manual". *Cadence Inc.*

[24] "DesignWare Library Manual". *Synopsys Inc.*

[25] Despian, M. A., and Huang, I. J. "Synthesis of Application Specific Instruction Sets". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (June 1995), pp 663–675.