The Hgen Hardware Synthesis System

by

Pietro Russo

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 20, 1999

[June 1999]

Author:.....

Department of Electrical Engineering and Computer Science

May 20, 1999

Certified by:.....

Srinivas Devadas

Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by:.....

Arthur C. Smith

Chairman, Department Committee on Graduate Students

The Hgen Hardware Synthesis System

by

Pietro Russo

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 1999

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer [Electrical] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Embedded systems dominate the consumer electronics market. Two factors that greatly affect a product's success are cost and time-to-market. In order to reduce costs a custom design is needed. Creating a custom design requires many interations of design and evaluation which increases the time-to-market and thus reducing the chances for success of the product. A solution to this problem is to *automate* the design and evaluation. The design process can be divided into the design of the software component (which includes the ASIP) and of the hardware component. This work deals with the automation of the hardware model of the ASIP.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Table of Contents

List of Figures

1 Introduction

An embedded system is a system which contains a programmable processor and is used in an application other than general purpose computing. Embedded systems constitute the core of most consumer electronics (e.g., VCRs, cellular phones, and handheld video games). Integrating as much of the system onto one chip has benefits such as reduced manufacturing costs. A system that is built with this methodology is called a System on a Chip (SOAC).

One of the most important parameters that determines the success of a consumer electronic product is its *time-to-market*. Time-to-market is the time it takes a product to start and complete its design and testing phases and be ready for production and shipment to the consumer market.

In order to cut down on manufacturing costs and improve the performance and efficiency of an embedded system, it is important to have a custom architecture with a processor that is well suited to the application, termed an Application Specific Instruction-set Processor (ASIP). Software and hardware are strongly coupled so they should be designed together to achieve a more optimal implementation (this is called Hardware/Software co-synthesis).

Once an initial design is formulated it goes through many iterations, many incremental changes in order to improve certain parameters or trade one parameter off for another. A conflict arises because it takes time to design a custom processor so as the time-to-market increases, the chance of success for the product decreases. In order to get a product with an embedded system onto the market fast the design process and evaluation needs to be *automated*. Automatic generation of the design evaluation tools allows rapid evaluation of target architectures, increasing the coverage of the design space while shortening the design time and, thus, the time-to-market.

An approach to designing embedded systems in an automated way is with the Architecture Exploration System ARIES shown in Figure 1.

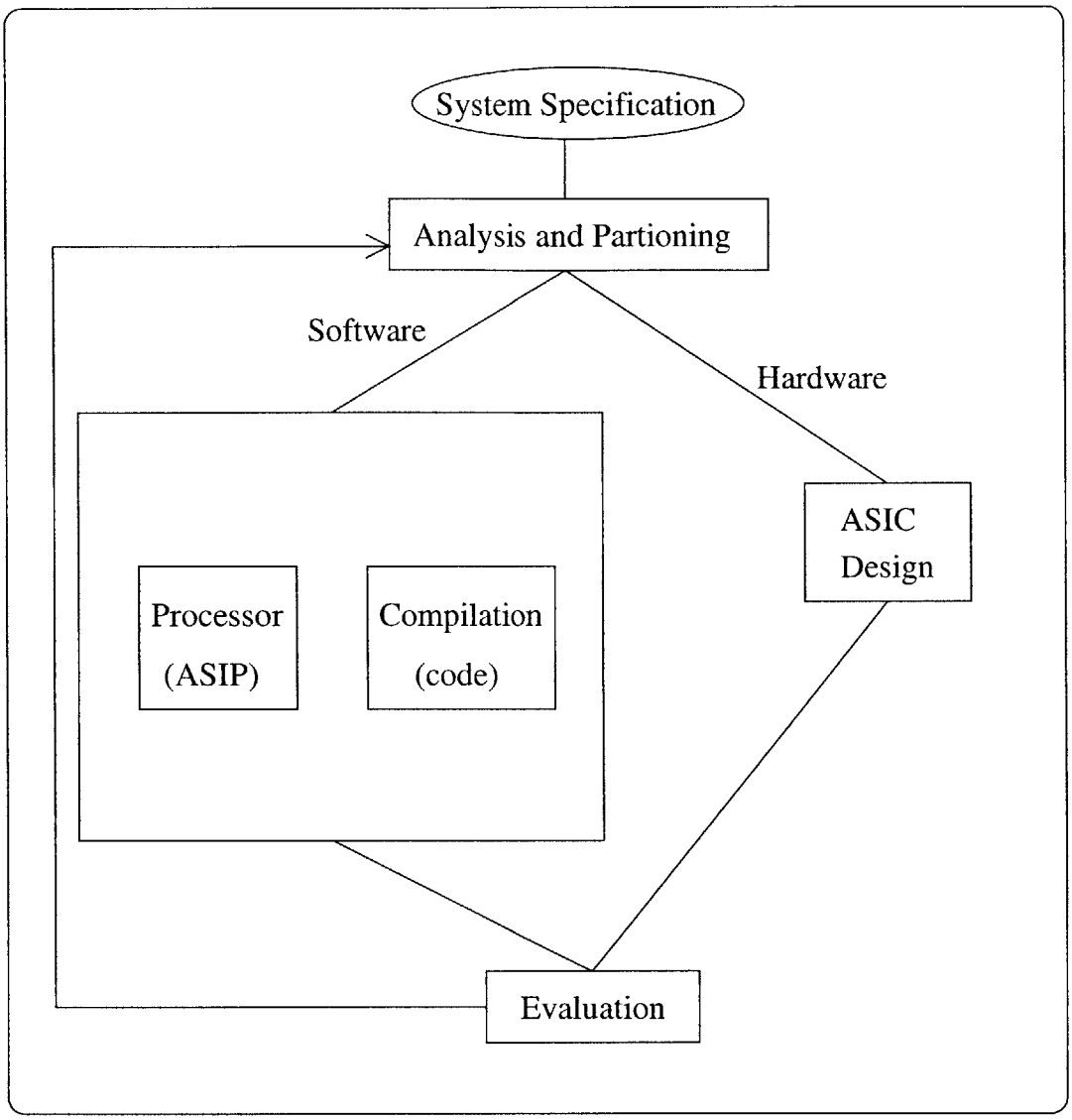

Figure 1 The Architecture Exploration System ARIES

Given a system specification the ARIES system produces the software and hardware implementations for that system. ARIES takes as input an initial system specification which it partitions into software and hardware design portions of the system. The hardware portion is dealt with by an ASIC (Application Specific Integrated Circuit) design tool in the ARIES system. The

software portion is divided into ASIP generation and code generation. This thesis focuses on the automated design of the ASIP.

The Hgen Hardware Synthesis System handles the evaluation of the ASIP. In order to generate the ASIP the Hgen system uses the description of the architecture produced by the Architecture Synthesis System. The Architecture Synthesis system generates the description of the architecture from the feedback information provided by Hgen and the other tools in ARIES. One critical part in determining the ASIP is the binary code that will run on it. The application code is first analyzed and then an initial architecture is generated and described in a machine description language. The application code is compiled for the architecture and then the compiled code is assembled into a binary code. This binary code is then simulated in the Instruction Level Simulator for the given processor. Hgen gives a hardware model of the processor that is used to derive performance measurements. The performance measurements are fed back to the Architecture Synthesis System for the next iteration of the design. This information allows it to decide how to vary the parameters for the next iteration of the design. Figure 2 shows the Hgen system and the design through iterative improvement ideology

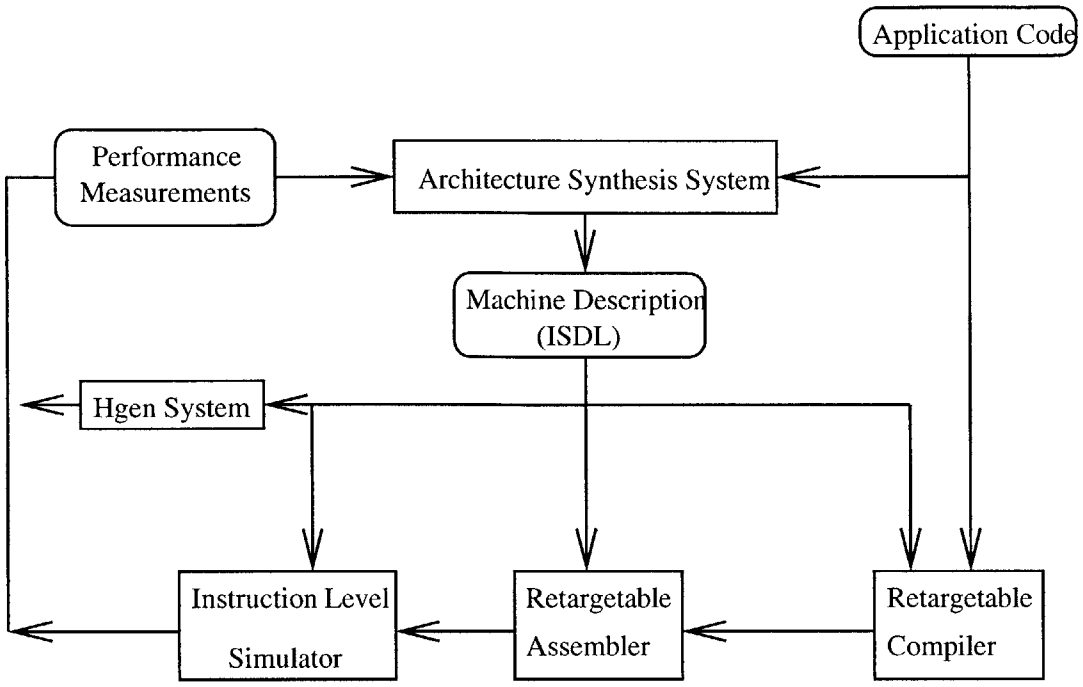

Figure 2 Design Through Iterative Improvement

1.1 Purpose

In order to evaluate any design it is necessary to have accurate measurements of its performance. To evaluate the programmable portion of the system (i.e., the ASIP and its application code) the following information is needed:

- Number of cycles to run the application code on the target architecture
- Length of the clock cycle (clock period)
- Die Size
- Power consumption

The Hgen Hardware Synthesis System provides *three* of the four parameters needed for performance evaluation. The ILS provides the number of clock cycles while the Hgen system provides the physical costs of the design: the clock period, die size, and power consumption. The physical costs are *absolutely necessary* to evaluate the design accurately.

Not only does the Hgen system provide these necessary physical costs but as an added benefit the output can serve as a first draft of the actual hardware implementation. This can cut down on the time-to-market even further because hardware model produced for the performance measurements can also be used to derive the actual hardware implementation. Most of the work needed to get a actual silicon implementation of the design is then done by the silicon compiler.

2 Hgen Overview

The Hgen Hardware Synthesis System takes in an ISDL description of an architecture and outputs a synthesizable Verilog file that is a hardware model for that ISDL description.

The Hgen system needs to meet the following requirements:

- The input must be an ISDL description. ISDL was designed for ARIES to support the automatic generation of all the tools. It is important to have only *one* description that all the tools are based on. If all the tools use the same description language, inconsistencies between the tools are avoided. All the tools in the ARIES system use ISDL, thus Hgen must use ISDL.
- The Hgen system outputs a synthesizable Verilog file. This output can be used by most synthesis tools to create a hardware implementation in any kind of underlying technology. Synthesizable Verilog also provides a clear hardware model that is readable for humans.
- The hardware model produced must be efficient. The synthesizable Verilog that the Hgen system produces must be comparable to a hand-coded version. The goal of the Hgen system is to produce a good design. We will define S_x^H as the silicon area needed for the Hgen design of an architecture x. Similarly S_x^P is the optimal design (e.g., a hand-coded design by the best hardware designer) for architecture x. In order for the automated design process to be useful we need to be able to compare different architectures at least at some qualitative level. This

means the equation $S_A^H > S_B^H \Rightarrow S_A^P > S_B^P$ must hold. We can then define efficiency as $E = S_x^P / S_x^H$. We see that when $E = 1$ we have an optimal design. We see that as the hardware model becomes more efficient, the equation we must satisfy has a greater chance of holding.

- The Hgen system has to take a reasonable amount of time to give its final design. Many iterations are needed to produce a reasonable design so the Hgen system must produce its output, the output for each iteration, in a reasonable amount of time.

3 ISDL

The input to the Hgen system is a description of the candidate architecture in the machine description language ISDL. ISDL is a behavioral language that explicitly lists the instruction set of the target architecture. It is based on an attributed grammar in which the production rules are used to abstract common patterns in operation definitions. ISDL models the processor as a set of state elements and a set of operations that modify the state elements. This section only provides a brief description of the ISDL language. For a complete description of the ISDL language refer to [3].

An ISDL description is divided into six sections:

1. Format Section - This section describes the binary representation of the instruction word. The instruction word is divided into fields and subfields for clarity.

2. Global Definitions Section - This section defines abstractions that will be used in later sections of the ISDL description are defined here. Two important types of definitions are:

1. Tokens: These represent the syntactic elements of the assembly language of the architecture. They can also group together syntactically related entities (such as the register names in a register-file). Tokens are provided with a return value that identifies the different options.

2. Non-terminals: Non-terminals abstract common patterns in operation definitions (e.g., addressing modes). A non-terminal definition consists of the non-terminal name and a list of options. Each option consists of the same six parts that make up an operation definition (see Instruction Set Section below).

3. Storage Section - All the visible state in the system is given a name, size, and type.

4. Instruction Set Section - This section is broken into a list of fields, each with a list of operation definitions. This is to accommodate for VLIW architectures. Each field roughly corresponds to a separate functional unit that runs in parallel with the other fields. An instruction is formed by taking one operation (at most) from each field.

1. Operation Syntax - Operation name and a list of parameters. A parameter is a token or nonterminal.

2. Bitfield Assignments - Describes how to set the instruction word bits to the appropriate values.

3. Operation Action - A set of RTL-type statements that describe the effect of the operation on the processor state.

4. Operation Side Effects - RTL-type statements that describe the side-effects of an operation (such as setting the carry bit).

5. Operation Costs - ISDL pre-defines three costs:

1. Cycle: the number of cycles the operation takes on hardware in the absence of stalls.

2. Stall: the number of additional cycles that may be necessary during a pipeline stall.

3. Size: the number of instruction words required for the operation.

6. Operation timing - Defines the timing of the operation effects. ISDL pre-defines two timing parameters:

1. Latency: Describes when the results of the operation become available.

2. Usage: Describes when the functional unit becomes available.

5. Constraints Section - An instruction word is formed by grouping together operations, one from each field. Not all combinations are valid. The constraints section describes all the valid combinations by listing constraints for each instruction which must be satisfied in order to have a valid instruction. Constraints help provide information about the implementation of the hardware which helps to generate more efficient hardware for a particular description.

6. Optional Architectural Information Section - This section provides additional information that is not necessary for the tools to function but may produce better tools. This section can be used for information such as the presence and structure of caches or describing exceptions and interrupts. The current version of ISDL (Version 1.0) does not predefine any Optional Information.

3.1 The SPAM2 Example Architecture

We illustrate the structure and features of ISDL using the following example:

The SPAM2 architecture is a simple load/store architecture. It has three arithmetic units, each with its own register-file. The register-files are eight bits wide and contain four registers each. This architecture has two buses that move data to/from the register-files from/to the data and instruction memories. For the complete ISDL description of the SPAM2 architecture refer to

Appendix A. For the complete Synthesizable Verilog model of the SPAM2 architecture refer to Appendix B. An overview of the architecture is shown in Figure 3:

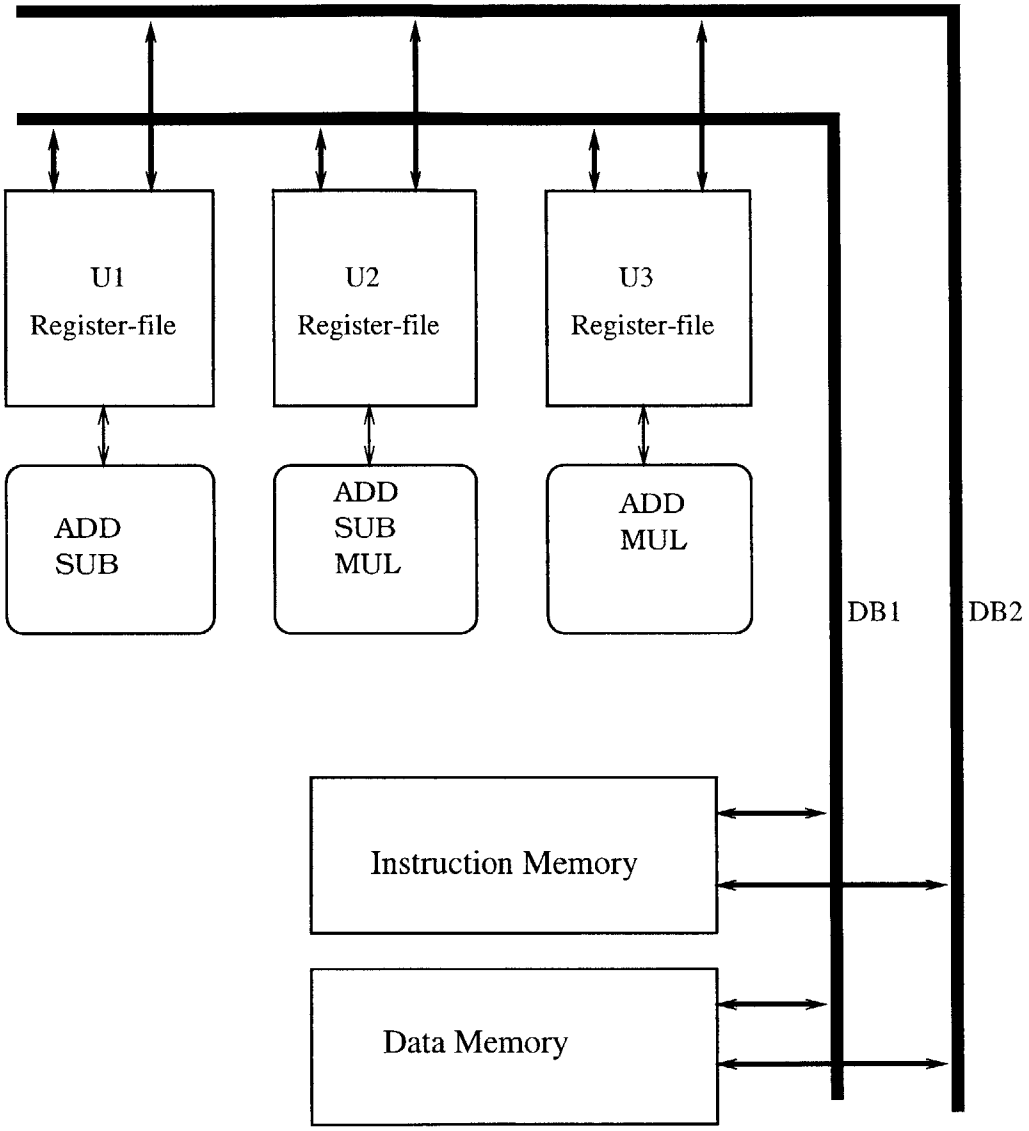

Figure 3 The SPAM2 Architecture

The U1 arithmetic unit has three operations: ADD, SUB, and NOP. ADD and SUB perform integer addition and subraction on the registers in the U1 register-file. Similarly U2 has ADD, SUB, MULT, and NOP. U3 has ADD, MULT, and NOP. DB1 and DB2 are data buses that can

move data between registers in different register-files. The data buses can also be used together to move data from a register to data or instruction memory and vice versa.

3.1.1 Format Section

The Format section of the ISDL description shows how to form the instruction word. For this architecture the Format section is as follows:

```
Section Format

U1      = OP[2], RA[2], RB[2], RC[2];
U2      = OP[2], RA[2], RB[2], RC[2];
U3      = OP[2], RA[2], RB[2], RC[2];
DB1     = SRC[5], DEST[5];
DB2     = SRC[5], DEST[5];
```

The terms on the left-hand side of the lines (U1, U2, U3, DB1, DB2) are the major divisions in the instruction word and are called fields. Each field can further be divided into subfields (i.e., for the U1 field, the subfields are OP, RA, RB, RC). Figure 4 shows the binary image of the instruction word in Figure 4. In the ISDL description we can refer to portions of the instruction word by <fieldname.subfieldname>. For example, the first two bits of the instruction word in this architecture can be referred to as U1.OP.

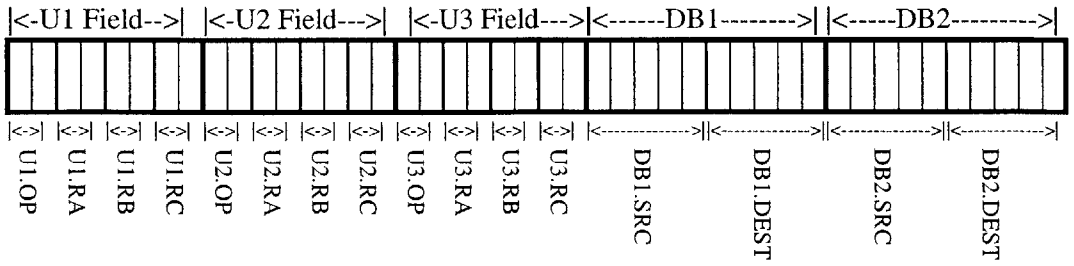

Figure 4 The Binary Image of the SPAM2 Instruction Word

3.1.2 Storage Section

The storage elements used in the architecture are explicitly listed in the Storage section of the ISDL description. The following is the Storage section for the SPAM2 architecture:

```
Section Storage

Instruction Memory INST                 = 0x100 , 0x2C
Memory DM                               =  0x20 , 0x8
RegFile U1                              =   0x4 , 0x8
RegFile U2                              =   0x4 , 0x8
RegFile U3                              =   0x4 , 0x8
ProgramCounter PC                       =         0x8
```

Each line in the storage section gives the type, name, depth (if any), and width of the storage. For example the third line in this section declares U1 as a register-file that has a depth of 4 and a width of 8. Figure 5 shows the complete programmer's model for these memories.

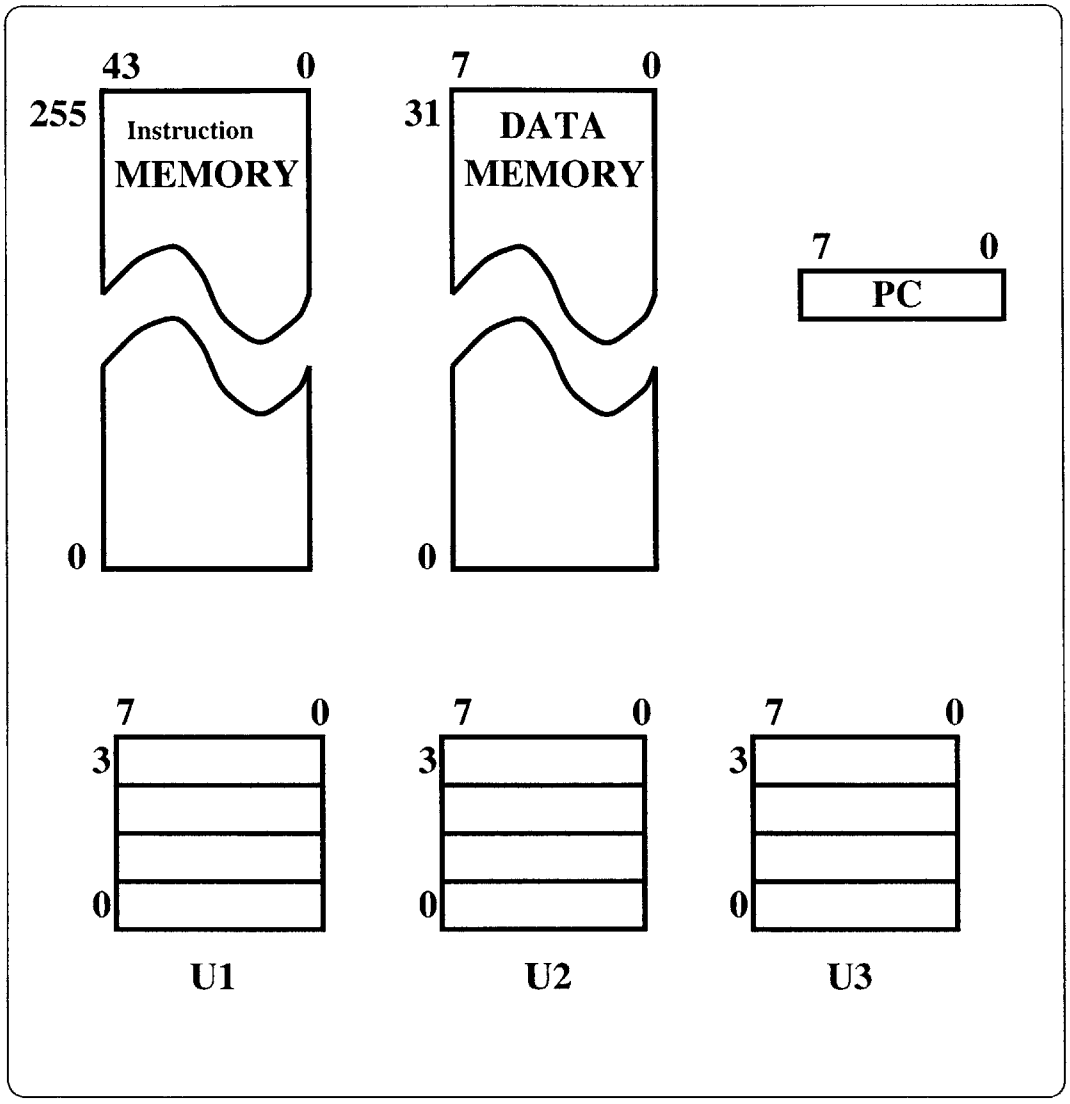

Figure 5 The Programmer's Model for the SPAM2 Architecture

3.1.3 Definitions Section

The Definitions section defines abstractions that will be used in later sections of the ISDL description. An example of a token definition for the SPAM2 architecture is seen below:

```
Token "U1.R"[0..3]     U1_R          { [0..3]; };
```

This token's name is U1_R. It represents any of the values U1.R0, U1.R1, U1.R2, and U1.R3 which represent the registers in the U1 register-file. U2_R and U3_R are defined similarly to represent the registers in the U2 and U3 register-files.

```
Non_Terminal U1_RA: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;
```

This non-terminal's name is U1_RA. It references the appropriate storage element in the U1 reg-file.

```
Non_Terminal SRC:  U1_R { $$ = 0x00 | U1_R; } {U1[U1_R]} {} {} {}|
                   U2_R { $$ = 0x04 | U2_R; } {U2[U2_R]} {} {} {}|
                   U3_R { $$ = 0x08 | U3_R; } {U3[U3_R]} {} {} {} ;
```

This non-terminal SRC has three options : U1_R, U2_R, and U3_R. Each option references the corresponding register-file. This nonterminal references any of the registers in the 3 register-files.

3.1.4 Instruction Set Section

The Instruction Set section is divided into fields. Each field has a list of operations that are mutually exclusive (i.e., cannot operate in parallel). Part of the Instruction set section of the SPAM2 architecture is shown below:

```
Field U1f:
  U1_add U1_RA, U1_RB, U1_RC
     { U1.OP = 0x0; U1.RA = U1_RA; U1.RB = U1_RB; U1.RC = U1_RC; }
     { U1_RC <- ADDm(U1_RA,U1_RB); }
     {}
     { Cycle = 1; Size = 1; Stall = 0; }
     { Latency = 1; Usage = 1; }
```

The name of the first instruction is U1_add. The third line provides the RTL action statement which shows that this is an integer add instruction that takes any two registers from the U1 register-file (given by the nonterminals U1_RA and U1_RB) and writes the result into a register in the U1 register-file (given by the nonterminal U1_RC). The second lines show the bitfield assignments for this operation. If we wanted to do an addition of the second and third registers in the U1

register-file and write the results to the 4th register the instruction word would look like this (underscores are added for clarity to show the field boundaries defined by the Format section and an x denotes a bit that is not affected by this operation):

00011011_xxxxxxxx_xxxxxxxx_xxxxxxxxxx_xxxxxxxxxx

The Operation Side Effect are shown in the fourth line. We see that there are no side-effects for this operation. Since this is an addition operation a possible side-effect could have been setting a carry bit. The costs and timing information in the last two lines show that this instruction completes in one cycle and the results of this instruction are available to the next instruction.

3.1.5 Constraints Section

The Constraints section is a list of boolean equations that must be satisfied by any instruction in order for the instruction to be valid. We have provided one constraint for the SPAM2 architecture below:

```
~( ((DB*_move *,*) | (DB*_nop))& ((DM_* *) | (IM_* *)) )
```

The '*' designates a wildcard. What the boolean equation above says is that the operations in the DB1 and DB2 fields cannot be used in parallel with the operations in the DM and IM fields. It states that if there is a data bus move or nop operation and a data or instruction memory operation at the same, then the operation is not a valid instruction. From this we can infer that the architecture is sharing the buses that perform these two classes of operations.

4 Translation

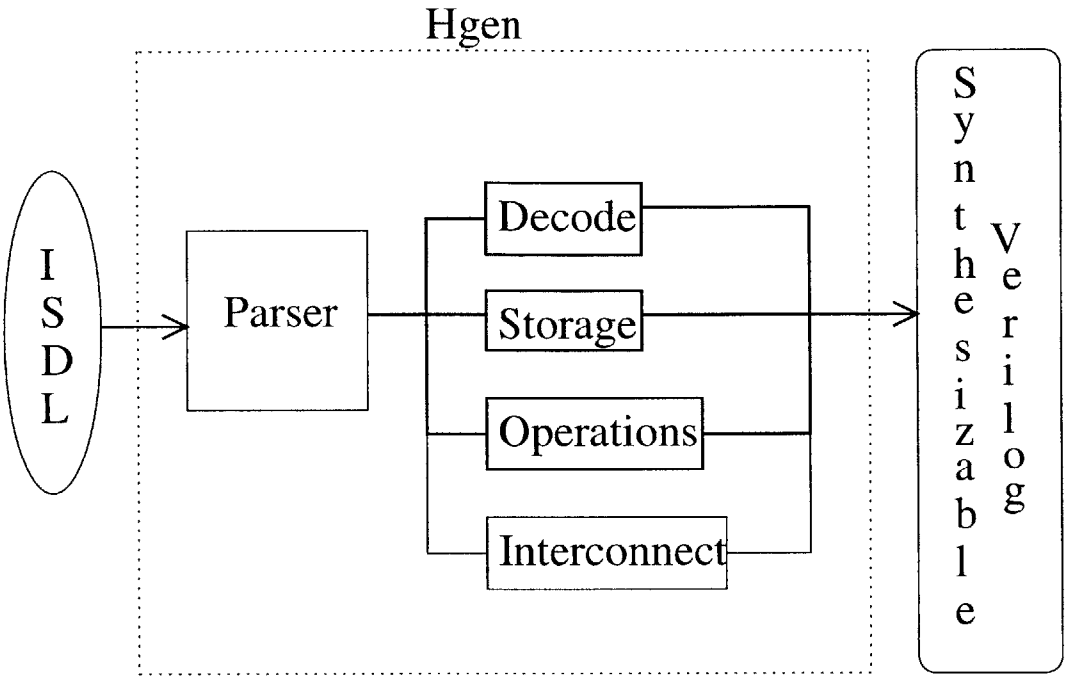

Figure 6 Hgen Methodology

4.1 Methodology

The methodology we use to produce synthesizable Verilog from an ISDL description can be seen in Figure 6. The Hgen system takes in the ISDL description and parses it using an already existing ISDL parser. The parser outputs a C++ data structure of the parsed ISDL in addition to performing some sanity checks to ensure the validity of the ISDL file. The data structure is examined by a Hgen system which then translates the ISDL code.

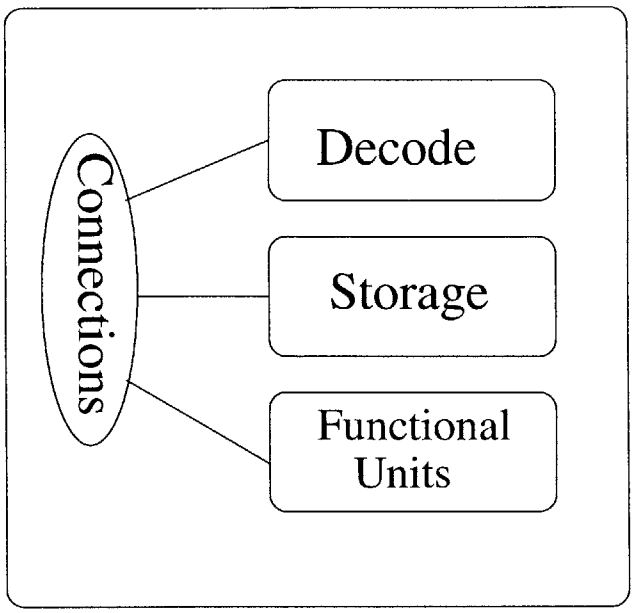

Figure 7 Synthesizable Verilog Code Structure

The translation occurs in parts. This division is seen in Figure 7. It is divided as follows:

- Decode Logic : This takes in the instruction word and output signals that will control which operations will be active for that instruction.
- Storage: Handles creating all the storage for the architecture
- Operations: Takes in the control signals from the decode logic and performs the necessary operations on the storage in the system.
- Interconnect: Handles the wire connections between the other sections.

4.2 Generating Efficient Models

In order to generate efficient hardware we need to make sure that we are sharing as much of the resources between operations as possible. When going from ISDL to synthesizable Verilog it is not easy to know when parts of the hardware can be shared by different operations. This is called the Resource Sharing problem. Consider a move operation that is implemented using a bus and load and store operations that are mutually exclusive with the move. The move operation resides

in a different field than the load and store operations. A naive scheme would generate additional data paths to handle the load and store operations even though it is possible to implement these with the same bus that implements the move (i.e., move, load, and store can share the bus).

4.2.1 Solution to Resource Sharing Problem

There is a systematic way of determining what can be shared. The RTL expressions for all operation definitions are broken up into a number of nodes that can be mapped to a circuit. Each of the n nodes is given a distinct number between 1 and n. Then an $n \times n$ matrix is created with entry $A_{ij}=1$ if the two nodes i and j would never operate in parallel (can be shared) and $A_{ij} = 0$ if the two nodes could possibly operate in parallel (cannot be shared). To determine the entries in the matrix the following criteria are used:

- 1. Nodes that are part of the same RTL statement will have to operate in parallel so they cannot be shared.
- 2. Nodes that perform completely different tasks (for example a shift operation and a bitwise XOR operation) cannot be shared and so are automatically assigned 0. Pairs where one node is a subset of another (e.g., the add and subtract operations) could be shared if the rest of the rules allow it.
- 3. Nodes belonging to operations in the same field in the Instruction Set Section will never run in parallel so they can be shared. This is due to the fact that operations in the same field are mutually exclusive by definition.
- 4. Nodes that belong to operations in different fields will probably have to operate in parallel (since the operations will probably have to operate in parallel) so it is assumed they cannot be shared.

When the constraints section is taken into account then more sharing may be possible because our last assumption (which is necessary if no information is given) can be overturned since information is available that will specify two nodes can (and should) be shared.

Once all the entries are filled in the matrix, maximal cliques of the nodes that can be shared can be created. A clique is a set of nodes such that for any pair of nodes i and j in the clique, Aij = 1. A maximal clique is a clique such that if any node is added to the clique, the resulting set of nodes is no longer a clique. The maximal cliques can be synthesized into circuits. Routing and glue logic completes the implementation. Pseudocode for this algorithm is shown in Figure 7.

```
Label each RTL operation with an integer
    for each i from 0 to n
        for each j from 0 to n
            Aij = 0
            if i and j functionally equivalent
              if i and j operations in same field
              or constraint between i and j
                  Aij = 1
        end
    end
  Generate maximal cliques for A
  Generate hardware for maximal cliques
```

Figure 8 Resource Sharing ALgorithm

4.3 Decode Logic

The decode logic provides the means for decoding the instruction word into signals which will become the control logic for the architecture.

The ISDL bitfield assignments provide the assembly function. This is a function that, for a given operation (or nonterminal option) and a given set of parameters, provides the values of the relevant bits of the instruction word.

Assembly Function: *f*(<operation, a, b, ...>) -> Instruction Word

The decode logic implements the *disassembly* function. In order to generate decode logic we need to reverse the assembly function (i.e., given the values of the bits in the instruction word we must first identify the operation and then provide the values of the parameters).

Disassembly Function: *g*(Instruction Word) -> <operation, a, b, ...>

We use the following model to derive the disassembly function from the bitfield assignments of an ISDL description (see Figure 9).

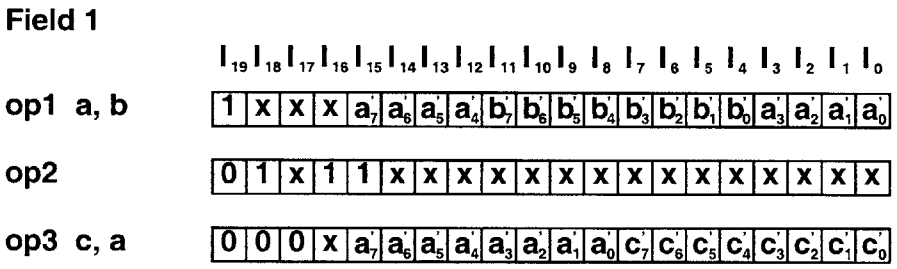

Figure 9 Operation Signatures

We associate a *signature* with each operation in every field of the Instruction Set section. A signature is an instruction word with symbols placed in for each bit. The following symbols are used:

- The constant "0" or "1" it implies that the assembly function for this operation sets the corresponding bit to the given constant.

- A parameter symbol (such as "a'_0") implies that the assembly function for the operation sets the corresponding bit to a function of the value of one of the parameters.
- "Don't care" entries (represented by "x") imply that the assembly function for this operation does not set the corresponding bit.

Our methodology is based on the following axiom:

Axiom 1 *Each parameter symbol in a signature is a function of a* single *parameter only.*

All architectures known to us have this property. It is clear why this property should be satisfied in all architectures: the purpose of bits carrying a parameter symbol is to convey the value of the original parameter to the hardware. Complex encodings involving more than one parameter are unnecessary. This combined with the fact that this encoding has to be reversed by a simple decode mechanism in hardware almost guarantees that such architectures will not occur.

With the signature of each operation and the axiom above we can reverse the assembly function as follows:

We attempt to match the constant part of the signature for each operation against the current instruction word. The match is guaranteed to be unique for a decodeable assembly function. We can then reverse the encoding of each parameter symbol bit in the instruction word to obtain the original parameter value. The axiom above guarantees that the encoding is reversible. Most of the time the encoding can be reversed symbolically (i.e., dealing with multiple bits at the same time). Figure 9 shows this algorithm in pseudo-code. The algorithm shown is recursive. It is presented this way for clarity. When generating the decode logic the number of recursions needed will determine the number of instantiations of each particular circuit (i.e., a recursive circuit is not actually produced, instead copies of the same circuit for each level of iteration needed are produced.).

```
Generate signatures for each operation in each field
Generate signatures for each option in each non-terminal

disassemble(i)
   for each f in description
      disassemble_field(i, f)
   end
disassemble_field(s, f)
   for each operation o in f
      if signature of o matches s
        for each parameter p in o
          case (p)
              token: reverse s to get token value t
              non-terminal: reverse s to get return value r
                 disassemble_ntl(r, p)
        end
      return OK
  end
  return ILLEGAL INSTRUCTION
disassemble_ntl(s, n)
    for each option o in n
       if signature of o matches s
            for each parameter p in o
              case (p)
                  token: reverse s to get token value t
                  non-terminal: reverse s to get return value r
                      disassemble_ntl(r, p)
            end
        return OK
      end
      return ILLEGAL INSTRUCTION
```

Figure 10 Decode Algorithm

There is one more complication that needs to be solved. In VLIW architectures it is possible for an operation op1 in field X to temporarily borrow the bits assigned by operations in field Y. In particular, if these bits belong to the op-code of field Y4, then the field Y must be disabled if op1 is active since it no longer corresponds to a real operation. In order to do this we define a prece-

dence relationship between operations and fields which identify when an operation in one field disables the operations of another. We then construct predicates which disable field Y if operation op1 was present in field X.

Note that neither the assembly nor the disassembly function are complete (valid for all inputs). However, the constraints describe invalid inputs to the assembly function, while invalid inputs to the disassembly function are allowed to result in undefined behavior since they should never occur in a valid program.

4.3.1 Generating Decode Logic

For each operation in a field we define a decode line which will be active if the operation is instantiated in the current instruction. The decode lines are grouped by the field they correspond to in the Instruction Set section of the ISDL description. These lines are collectively referred to as the "identification code" for the corresponding field.

We can derive an equation for each decode line by simply examining the constants in the operation signature. For example, the equation for the operation op2 in Figure 9 is $\overline{I19}\ I18\ I16\ I15$. This results in a very efficient two-level implementation. Similarly, logic can be generated from the decode functions that reverse parameter encodings. Finally a set of multiplexers and glue logic completes the decode circuit.

4.4 Storage

The ISDL description provides a list of all the storage in the architecture but it does not explicitly give the number or type of ports needed for each storage unit. We can derive the number of ports by counting in how many different fields in the Instruction Set section there are accesses to

that storage unit. Finding the number of ports is an instance of the resource sharing problem that we deal with using the method described in section 4.2.1.

To determine the type of port we examine the type of access that the operations in each field could make. If the operations can only do either one read or one write then this is merged into a bi-directional port.

4.5 Operations

The functional units are created from the information provided in the Instruction Set section of the ISDL description. Each field in the Instruction Set section roughly corresponds to a functional unit and so the field boundaries are used to provide modularity in the synthesizable Verilog output.

The modules in the synthesizable Verilog output each have a set of inputs and outputs that connect to the storage units and to the decode logic. One of the inputs to the module is its identification code discussed in section 4.3.1 . This informs the module which operation is active if any. The decode logic for the nonterminals is determined and placed within the module. Determining if different modules can share the decode logic for a nonterminal is subject to the resource sharing problem. The nonterminals used will partly determine the input, output, or inout ports for the memory accesses that the module needs to make. These ports are determined by the memory accesses made by the nonterminals. The other ports are determined through the direct memory accessed made by the module.

4.6 Interconnect

The connections between the storage, decode logic and operations are relatively simple since most of the work is done during the creation of the other modules. Determining which connections are necessary for each module is done during the creation of the module itself. The ports on each module name the inputs and ouputs the module according to a set formula. If there is to be a connection between two ports on different modules then the name generated for the connecting wire will be the same since the same fore each port. This designates a connection in Verilog. The same formula will be used by each module and so the connections are created as the module is created. There are a few extra connections which connect to external signals, such as the clock signal, which are taken care of separately once all the other connections are determined.

4.7 Deriving Structural Information

We have already seen that we can obtain information about the datapaths in an architecture through the Constraints section of an ISDL description. ISDL is a behavioral language and thus structural information needs to be inferred. We can derive information about pipelines and bypass logic from the costs and timing information that is provided in the Instruction Set section.

For simplicity we will discuss ISDL descriptions with all operations having the same cost and timing information. These are some examples of the different types of pipelines:

- A Cycle cost of 1, a Stall cost of 0, and a Latency cost of 4 implies a 5-stage data-path pipeline with no bypass logic.
- A Cycle cost of 1, a Stall cost of 4, and a Latency cost of 1 implies a 5-stage data-path pipeline that is fully bypassed
- A Cycle cost of 1, a Stall cost of 0, and a Latency cost of 1 implies either no pipeline or a pipeline that is fully bypassed with no stalls.

- A Cycle cost of 4, a Stall cost of 0, and a Latency cost of 1 implies a microcoded machine that takes 4 cycles per instruction.

Pattern matching on these parameters in the ISDL description will be our method of determining the existence of a pipeline.

5 Related Work

This section describes previous work related to machine description languages and hardware synthesis. The systems described are compared to our system as appropriate.

5.1 Mimola

The Mimola[6] design system was created as a high-level design environment for hardware, based on the Mimola hardware description language[7]. Later on, the system evolved to a hardware-software co-design environment. The system was designed for development and evaluation of implementations at a much lower level than ISDL. The Mimola language is a structural description at a relatively low level, and thus results in unnecessarily long and complex descriptions, and in slower simulators. On the other hand, the low-level detail makes it much easier to synthesize hardware from the descriptions.

5.2 nML

The nML machine description language[8] is a high-level machine description language that can be used to support automatically generated tools. It was used in the CHESS[9] system for retargetable code-generation as well as a variety of other tools[10]. nML is very similar to ISDL

in that it is a behavioral language based on attributed grammars. The main difference between nML and ISDL is the way constraints are handled. nML can only describe valid instructions. Therefore, it must work around invalid combinations by using additional rules to describe interactions between operations. Thus, nML descriptions are longer and less intuitive than ISDL descriptions. It is also unclear how well suited nML would be for hardware generation, since the constraints provide a lot of structural information used to generate efficient hardware.

5.3 FlexWare

FlexWare[11,12] is a software-firmware system for the development of custom ASIPs and commercial processors. It was developed specifically to support the development of DSP processors and embedded system software. It consists of the code-generator CodeSyn[11] and the simulator Insulin[12]. The FlexWare system can be used to rapidly evaluate architectures.

FlexWare suffers from the fact that it uses two different machine descriptions for the code-generator and the simulator. This raises consistency issues and makes the work of generating tools for a given architecture harder. It is unclear whether the system is well suited to hardware generation since there are no publications describing attempts to implement such a system.

5.4 LISA

The LISA[13] language was developed as a machine description language specifically designed to support the generation of simulators for specific architectures.

LISA contains a lot of structural information and can model most of the complicated timing effects that are likely to be encountered in embedded applications.

LISA is very effective at generating good simulators. Given the structural content in a LISA description, hardware generation should also be possible although we are unaware of any publications describing such a system. However, LISA is not well suited for generating code-generators and assemblers. If it was used in a system such as ours, a separate language would have to be used for code generation, thus resulting in consistency issues as well as making it harder to generate, describe, and evaluate architectures.

5.5 HMDES/Playdoh

HMDES[14] is a machine description language that was developed specifically for the Trimaran compiler system. It is based on a parameterizable architecture called Playdoh[15]. Playdoh represents a very general class of architectures which includes features as complicated as predicated execution and complex instructions.

HMDES is designed around the Playdoh architecture and supports all of its features. It is a mixed language containing both structural and behavioral information. Both the Playdoh architecture and the HMDES language were specifically designed to support VLIW architectures.

While Playdoh is very general and can encompass a wide variety of architectures, it is still a parameterized architecture and thus has a limited scope. Similarly Hmdes supports a parameterizable instruction set and therefore has a more restrictive scope than ISDL. Like nML, Hmdes does not support constraints which may result in longer and less intuitive descriptions. Note, however, that Hmdes, like Lisa, contains a slightly more extensive timing model than ISDL does, and can thus describe architectures that ISDL cannot. However, we believe that architectures that make use of such features will rarely occur in practice.

6 Conclusion and Future Work

Our experimental results show that the Hgen system can generate efficient hardware even for large complex designs.

Architecture	Cycle (nsec)	Lines of Verilog	Die Size (cells)	Synthesis time (sec)
SPAM1	32	1042	31443	827
SPAM2	28	405	4465	100
SCU	20.5	961	35647	728

Table 1 Hardware Synthesis Statistics

In table 1 we see some results produced by Hgen. SPAM1 is a floating-point VLIW architecture that can do 4 operations and 3 parallel moves at the same time. The synthesis was run on a Sun Ultra 30/300 running Solaris2.6. We use the Santa Clara University SCU RTL 98 DSP as a reference point to compare our results. We see that the synthesis time and die size for the SPAM1 architecture is comparable to that of the SCU architecture. This is to be expected from the relative complexities of the designs. The SPAM2 architecture is much smaller and takes less time to synthesize than the other two architectures. This is also expected since the SPAM2 has an 8-bit data path while the other two architectures have 32-bit data paths. The cycle length for SPAM1 is larger than that for SCU because SPAM1 has floating point hardware.

The Hgen system provides the performance measurements that the ARIES system needs to optimize the entire system design. The run time of the Hgen system is reasonable (on the order of minutes for one interation) and is dominated by the time taken by the synthesis tool. Future work on Hgen includes the implementation of pipelines in the hardware model.

References

[1] G. Hadjiyannis, P. Russo, and S.Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. To appear in *Proceedings of the* 36^{th} *Design Automation Conference*, June 1999

[2] G. Hadjiyannis, P. Russo, and S.Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. To appear in *The 6th IEEE International Conference on Electronics, Circuits and Systems*, June 1999

[3] G. Hadjiyannis. *ISDL: Instruction Set Description Language - Version 1.0.* MIT Laboratory for Computer Science, 545 Technology Sq., Cambridge, Massachusetts, July 1998. (http://www.caa.lcs.mit.edu/~ghi/postScript/isdl_manual.ps).

[4] G. Hadjiyannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 299-302, June 1997.

[5] G. Hadjiyannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. Technical Report, Massachusetts Institute of Technology, 1996. (http://www.ee.princeton.edu/spam/pubs/ISDL-TR.html).

[6] P. Marwedel. The MIMOLA Design System: Tools fot the Design of Digital Processors. In *Proceedings of the* 21^{st} *Design Automation Conference*, pages 587-593, 1984.

[7] G. Zimmermann. The MIMOLA Design System: A Computer Aided Digital Processor Design Method. In *Proceedings of the 16th Design Automation Conference*, pages 53-58, 1979.

[8] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Sets Using nML (Extended Version). Technical report, Technische Universitat Berlin and IMEC, Berlin (Germany)/Leuven (Belgium), 1995.

[9] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. In *Code Generation for Embedded Processors*, chapter 5, pages 85-102. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[10] M. A. Hartoog et al. Generation of Software Tools from the Processor Descriptions for Hardware/Software Codesign. In *Proceedings of the 34th Design Automation Conference*, Pages 303-306, June 1997.

[11] P. G. Paulin, C Liem, T.C. May, and S. Sutarwala. CodeSyn: A Retargetable Code Synthesis System. In *Proceedings of the 7th International High-Level Synthesis Workshop*, Spring 1994.

[12] S. Sutarwala, P. G. Paulin, and Y. Kumar. Insulin: An Instruction Set Simulation Environment. In *Proceedings of the 1993 Conference on Hardware Description Languages*, pages 355-362, 1993.

[13] V. Zivojnovic, S. Pees, and M. Meyr. LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design. In *Proceedings of the 1996 IEEE Workshop on VLSI Signal Processing*, 1996.

[14] J. C. Gyllenhall, W. W. Hwu, and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3, University of Illinois, Urbana, 1996.

[15] V. Kathail, M. S Schlansker, and B. R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, 1994.

Appendix A

Complete ISDL Description for the SPAM2 Architecture

```
Section Format

U1      = OP[2], RA[2], RB[2], RC[2];
U2      = OP[2], RA[2], RB[2], RC[2];
U3      = OP[2], RA[2], RB[2], RC[2];
DB1     = SRC[5], DEST[5];
DB2     = SRC[5], DEST[5];

// ------------------------------------------------------------

Section Global_Definitions

//    assembly      token       value
Token "U1.R"[0..3]    U1_R        { [0..3]; };
Token "U2.R"[0..3]    U2_R        { [0..3]; };
Token "U3.R"[0..3]    U3_R        { [0..3]; };

Non_Terminal U1_RA: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RB: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RC: U1_R { $$ = U1_R; } {U1[U1_R]} {} {} {} ;

Non_Terminal U2_RA: U2_R { $$ = U2_R; } {U2[U2_R]} {} {} {} ;
Non_Terminal U2_RB: U2_R { $$ = U2_R; } {U2[U2_R]} {} {} {} ;
Non_Terminal U2_RC: U2_R { $$ = U2_R; } {U2[U2_R]} {} {} {} ;

Non_Terminal U3_RA: U3_R { $$ = U3_R; } {U3[U3_R]} {} {} {} ;
Non_Terminal U3_RB: U3_R { $$ = U3_R; } {U3[U3_R]} {} {} {} ;
Non_Terminal U3_RC: U3_R { $$ = U3_R; } {U3[U3_R]} {} {} {} ;

Non_Terminal SRC:  U1_R { $$ = 0x00 | U1_R; } {U1[U1_R]} {} {} {}|
                   U2_R { $$ = 0x04 | U2_R; } {U2[U2_R]} {} {} {}|
                   U3_R { $$ = 0x08 | U3_R; } {U3[U3_R]} {} {} {} ;

Non_Terminal DEST: U1_R { $$ = 0x00 | U1_R; } {U1[U1_R]} {} {} {}|
                   U2_R { $$ = 0x04 | U2_R; } {U2[U2_R]} {} {} {}|
                   U3_R { $$ = 0x08 | U3_R; } {U3[U3_R]} {} {} {} ;

#define REG SRC
#define LOC DEST
```

```
// ------------------------------------------------------------

Section Storage

//                                          = entries , bits_per_entry
Instruction Memory INST                      = 0x100 , 0x2C
Memory DM                                    =  0x20 , 0x8
RegFile U1                                   =   0x4 , 0x8
RegFile U2                                   =   0x4 , 0x8
RegFile U3                                   =   0x4 , 0x8
ProgramCounter PC                            =         0x8

// ------------------------------------------------------------

#define DEFINE_NULL_OP  {} { NULLOP(); } {} {} {}

#define ADDm(x,y)     ADD(x,y,8,"trn")
#define SUBm(x,y)     SUB(x,y,8,"trn")
#define MULm(x,y)     MUL(x,y,8,8,"trn")

Section Instruction_Set

Field U1f:
        U1_NULL DEFINE_NULL_OP
        U1_add U1_RA, U1_RB, U1_RC
                { U1.OP = 0x0; U1.RA = U1_RA; U1.RB = U1_RB; U1.RC
= U1_RC; }
                 { U1_RC <- ADDm(U1_RA,U1_RB); }
                 {}
                 { Cycle = 1; Size = 1; Stall = 0; }
                 { Latency = 1; Usage = 1; }
        U1_sub U1_RA, U1_RB, U1_RC
                { U1.OP = 0x1; U1.RA = U1_RA; U1.RB = U1_RB; U1.RC
= U1_RC; }
                 { U1_RC <- SUBm(U1_RA,U1_RB); }
                 {}
                 { Cycle = 1; Size = 1; Stall = 0; }
                 { Latency = 1; Usage = 1; }
        U1_nop
                 { U1.OP = 0x3; }
                 { NOP(); }
                 {}
                 { Cycle = 1; Size = 1; Stall = 0; }
                 { Latency = 1; Usage = 1; }

Field U2f:
```

```
        U2_NULL DEFINE_NULL_OP
        U2_add U2_RA, U2_RB, U2_RC
               { U2.OP = 0x0; U2.RA = U2_RA; U2.RB = U2_RB; U2.RC
= U2_RC; }
                { U2_RC <- ADDm(U2_RA,U2_RB); }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
        U2_sub U2_RA, U2_RB, U2_RC
               { U2.OP = 0x1; U2.RA = U2_RA; U2.RB = U2_RB; U2.RC
= U2_RC; }
                { U2_RC <- SUBm(U2_RA,U2_RB); }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
        U2_mul U2_RA, U2_RB, U2_RC
               { U2.OP = 0x2; U2.RA = U2_RA; U2.RB = U2_RB; U2.RC
= U2_RC; }
                { U2_RC <- MULm(U2_RA,U2_RB); }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 4; Usage = 1; }
        U2_nop
                { U2.OP = 0x3; }
                { NOP(); }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }

Field U3f:
        U3_NULL DEFINE_NULL_OP
        U3_add U3_RA, U3_RB, U3_RC
               { U3.OP = 0x0; U3.RA = U3_RA; U3.RB = U3_RB; U3.RC
= U3_RC; }
                { U3_RC <- ADDm(U3_RA,U3_RB); }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
        U3_mul U3_RA, U3_RB, U3_RC
               { U3.OP = 0x1; U3.RA = U3_RA; U3.RB = U3_RB; U3.RC
= U3_RC; }
                { U3_RC <- MULm(U3_RA,U3_RB); }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 4; Usage = 1; }
        U3_nop
```

```
                { U3.OP = 0x3; }
                { NOP(); }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }

// DB1 is used for the data
Field DB1:
        DB1_NULL DEFINE_NULL_OP
        DB1_move SRC, DEST
                { DB1.SRC = SRC; DB1.DEST = DEST; }
                { DEST <- SRC; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
        DB1_move_im INT, DEST
                { DB1.SRC = 0x10 | (INT & 0xF); DB1.DEST = DEST; }
                { DEST <- INT; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
        DB1_nop
                { DB1.DEST = 0x1F; }
                { NOP(); }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }

// DB2 is used for the address
Field DB2:
        DB2_NULL DEFINE_NULL_OP
        DB2_move SRC, DEST
                { DB2.SRC = SRC; DB2.DEST = DEST; }
                { DEST <- SRC; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
        DB2_move_im INT, DEST
                { DB2.SRC = 0x10 | (INT & 0xF); DB2.DEST = DEST; }
                { DEST <- INT; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }
        DB2_nop
                { DB2.DEST = 0x1F; }
                { NOP(); }
```

```
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }

#define DMdata 0x0C
#define DMaddr 0x0D

Field DMf:
        DM_NULL DEFINE_NULL_OP
        // DB1.SRC gets code for DM_DATA, DB2.DEST gets code for
DM_ADDR
        DM_ld   REG, LOC
                { DB1.SRC = DMdata; DB1.DEST = REG;
                  DB2.SRC = LOC; DB2.DEST = DMaddr; }
                { REG <- DM[LOC]; }
                {}
                { Cycle = 1; Size = 1; Stall = 1; }
                { Latency = 1; Usage = 1; }
        // DB1.DEST gets code for DM_DATA, DB2.DEST gets code for
DM_ADDR
        DM_st   REG, LOC
                { DB1.SRC = REG; DB1.DEST = DMdata;
                  DB2.SRC = LOC; DB2.DEST = DMaddr; }
                { DM[LOC] <- REG; }
                {}
                { Cycle = 1; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }

#define IMdata 0x0E
#define IMaddr 0x0F

Field IM:
        IM_NULL DEFINE_NULL_OP
        // DB1.SRC gets code for IM_DATA, DB2.DEST gets code for
IM_ADDR
        IM_ld   REG, LOC
                { DB1.SRC = IMdata; DB1.DEST = REG;
                  DB2.SRC = LOC; DB2.DEST = IMaddr; }
                { REG <- INST[LOC]; }
                {}
                { Cycle = 2; Size = 1; Stall = 1; }
                { Latency = 1; Usage = 1; }
        // DB1.DEST gets code for IM_DATA, DB2.DEST gets code for
IM_ADDR
        IM_st   REG, LOC
                { DB1.SRC = REG; DB1.DEST = IMdata;
```

```
                  DB2.SRC = LOC; DB2.DEST = IMaddr; }
                { INST[LOC] <- REG; }
                {}
                { Cycle = 2; Size = 1; Stall = 0; }
                { Latency = 1; Usage = 1; }

// ------------------------------------------------------------

Section Constraints

// SRC and DEST cannot be the same on either bus
~( DB*_move U@[1].R*, U@[1].R* )

// Can not use buses for a move between register files if a memory
// operation is using the buses
~( ((DB*_move *,*) | (DB*_nop))& ((DM_* *) | (IM_* *)) )

// Can not do both a DM and IM operation because they use the same
buses
~( (DM_* *) & (IM_* *) )

// Can not write to same register from two different operations
~( (DB1_move *,@[1]) & (DB2_move *,@[1]) )

// ------------------------------------------------------------

Section Optional
```

Appendix B

Synthesizable Verilog Model for the SPAM2 Architecture

```
module SPAM2 (clk,PC);
input clk;
parameter PCW = 'h8;  //program counter width
output [7:0] PC;
reg [PCW-1:0] PC;
reg [43:0] ir;
`define U1 ir[43:36]
`define U1_OP ir[43:42]
`define U1_RA ir[41:40]
`define U1_RB ir[39:38]
`define U1_RC ir[37:36]

`define U2 ir[35:28]
`define U2_OP ir[35:34]
`define U2_RA ir[33:32]
`define U2_RB ir[31:30]
`define U2_RC ir[29:28]

`define U3 ir[27:20]
`define U3_OP ir[27:26]
`define U3_RA ir[25:24]
`define U3_RB ir[23:22]
`define U3_RC ir[21:20]

`define DB1 ir[19:10]
`define DB1_SRC ir[19:15]
`define DB1_DEST ir[14:10]
`define DB1_SRC42 ir[19:17]
`define DB1_SRC10 ir[16:15]
`define DB1_DEST42 ir[14:12]
`define DB1_DEST10 ir[11:10]

`define DB2 ir[9:0]
`define DB2_SRC ir[9:5]
`define DB2_DEST ir [4:0]
`define DB2_SRC42 ir[9:7]
`define DB2_SRC10 ir[6:5]
`define DB2_DEST42 ir[4:2]
`define DB2_DEST10 ir[1:0]
```

```
wire [43:0] IMdata;
wire [7:0] DMaddr;
wire [7:0] b1datau1, b2datau1, b1datau2, b2datau2, b1datau3,
b2datau3, DMdata, IMaddr,
U1RA,U1RB,U1RC,U2RA,U2RB,U2RC,U3RA,U3RB,U3RC;
wire [1:0] b1addru1, b2addru1, b1addru2, b2addru2, b1addru3,
b2addru3;

u1module U1 (enableu1, `U1_OP, U1RA, U1RB, U1RC);
u2module U2 (enableu2, `U2_OP, U2RA, U2RB, U2RC);
u3module U3 (enableu3, `U3_OP, U3RA, U3RB, U3RC);

urfmodule U1RF (clk, `U1_RA, U1RA, `U1_RB, U1RB, `U1_RC, U1RC,
enableu1, b1addru1, b1datau1, ((`DB1_SRC42 == 'h0)||(`DB1_DEST42
== 'h0)),(`DB1_DEST42 == 'h0) , b2addru1, b2datau1, ((`DB2_SRC42
== 'h0)||(`DB2_DEST42 == 'h0)), (`DB2_DEST42 == 'h0));

urfmodule U2RF (clk, `U2_RA, U2RA, `U2_RB, U2RB, `U2_RC, U2RC,
enableu2, b1addru2, b1datau2,((`DB1_SRC42 == 'h1)||(`DB1_DEST42
== 'h1)) ,(`DB1_DEST42 == 'h1) , b2addru2, b2datau2,((`DB2_SRC42
== 'h1)||(`DB2_DEST42 == 'h1)) ,(`DB2_DEST42 == 'h1) );

urfmodule U3RF (clk, `U3_RA, U3RA, `U3_RB, U3RB, `U3_RC, U3RC,
enableu3, b1addru3, b1datau3, ((`DB1_SRC42 == 'b10)||(`DB1_DEST42
== 'b10)),(`DB1_DEST42 == 'b10) , b2addru3, b2datau3,((`DB2_SRC42
== 'b10)||(`DB2_DEST42 == 'b10)) ,(`DB2_DEST42 == 'b10) );


dbmodule DB1 (`DB1_SRC, `DB1_DEST, b1datau1, b1datau2, b1datau3,
DMdata, IMdata[7:0]);

dbmodule DB2 (`DB2_SRC, `DB2_DEST, b2datau1, b2datau2, b2datau3,
DMaddr, IMaddr);


DM dm (clk, DMaddr, DMdata, (`DB1_DEST == 'h0c), (`DB2_DEST ==
'h10));
//IM im (clk, ((`DB1_DEST == 'h14) ? IMaddr : PC), IMdata,
(`DB1_DEST == 'h14), (`DB2_DEST == 'h18)) ;
IM im (clk, PC, IMdata, 1'b0 , 1'b1 ) ;

always @(posedge clk)
begin
   PC <= PC + 1;
   ir <= IMdata;
end
```

```
module u1module (enable, U1OP, U1RA, U1RB, U1RFRC);
output enable;
reg enable;
input [1:0] U1OP;
input [7:0] U1RA, U1RB;
output [7:0] U1RFRC;
reg [7:0] U1RFRC;

always @(U1OP or U1RA or U1RB )
begin
case (U1OP)
2'b00 : begin
  enable <= 1;
  U1RFRC <= add8(U1RA, U1RB); //u1add
                end
2'b01 : begin
  enable <= 1;
  U1RFRC <= sub8(U1RA, U1RB); //u1sub
     end
default : begin
    enable <= 0;
      U1RFRC <= 8'b0;
  end
endcase
end

/***********************************************/
function [7:0] add8;
input [7:0] a, b;
begin
      add8 = a + b;
end
endfunction
/***********************************************/
function [7:0] sub8;
input [7:0] x, y;
begin
      sub8 = x - y;
end
endfunction

endmodule //u1module
```

```
module u2module (enable, U2OP, U2RA, U2RB, U2RFRC);
output enable;
reg enable;
input [1:0] U2OP;
input [7:0] U2RA, U2RB;
output [7:0] U2RFRC;
reg [7:0] U2RFRC;

always @(U2OP or U2RA or U2RB )
begin
case (U2OP)
2'b00 : begin
  enable <= 1;
  U2RFRC <= add8(U2RA, U2RB); //u2add
                end
2'b01 : begin
  enable <= 1;
  U2RFRC <= sub8(U2RA, U2RB); //u2sub
     end
2'b10 : begin
  enable <= 1;
  U2RFRC <= mul8(U2RA, U2RB); //u2mul
end
default : begin
    enable <= 0;
      U2RFRC <= 8'b0;
  end
endcase
end

/***********************************************/
function [7:0] add8;
input [7:0] a, b;
begin
      add8 = a + b;
end
endfunction
/***********************************************/
function [7:0] sub8;
input [7:0] x, y;
begin
      sub8 = x - y;
end
endfunction
/***********************************************/
function [7:0] mul8;
```

```
input [7:0] x, y;
begin
      mul8 = x * y;
end
endfunction

endmodule //u2module


module u3module (enable, U3OP, U3RA, U3RB, U3RFRC);
output enable;
reg enable;
input [1:0] U3OP;
input [7:0] U3RA, U3RB;
output [7:0] U3RFRC;
reg [7:0] U3RFRC;

always @(U3OP or U3RA or U3RB )
begin
case (U3OP)
2'b00 : begin
  enable <= 1;
  U3RFRC <= add8(U3RA, U3RB); //u3add
                end
2'b01 : begin
  enable <= 1;
  U3RFRC <= mul8(U3RA, U3RB); //u3sub
     end
default : begin
    enable <= 0;
      U3RFRC <= 8'b0;
  end
endcase
end

/***********************************************/
function [7:0] add8;
input [7:0] a, b;
begin
      add8 = a + b;
end
endfunction
/***********************************************/
function [7:0] mul8;
input [7:0] x, y;
begin
```

```
      mul8 = x * y;
end
endfunction


endmodule //u3module



module dbmodule(DBSRC, DBDEST, U1R, U2R, U3R, DM, IM);

input [4:0] DBSRC;
input [4:0] DBDEST;
inout [7:0] DM;
inout [7:0] IM;
inout [7:0] U1R;
inout [7:0] U2R;
inout [7:0] U3R;
reg [7:0] DMtmp;
reg [7:0] IMtmp;
reg [7:0] U1Rtmp;
reg [7:0] U2Rtmp;
reg [7:0] U3Rtmp;

reg [7:0] DB;

  always @(DBSRC or DBDEST or DM or IM or U1R or U2R or U3R)
    begin
      case (DBSRC)
        5'h0 : DB = U1R;
        5'h1 : DB = U1R;
        5'h2 : DB = U1R;
        5'h3 : DB = U1R;
        5'h4 : DB = U2R;
        5'h5 : DB = U2R;
        5'h6 : DB = U2R;
        5'h7 : DB = U2R;
        5'h8 : DB = U3R;
        5'h9 : DB = U3R;
        5'ha : DB = U3R;
        5'hb : DB = U3R;
        5'hc : DB = DM;
5'he : DB = IM;
        default : DB = 32'bz;
      endcase
      U1Rtmp = 8'bz;
```

```
      U2Rtmp = 8'bz;
      U3Rtmp = 8'bz;
      DMtmp = 8'bz;
      IMtmp = 8'bz;

      case (DBDEST)
        5'h0 : U1Rtmp = DB;
        5'h1 : U1Rtmp = DB;
        5'h2 : U1Rtmp = DB;
        5'h3 : U1Rtmp = DB;
        5'h4 : U2Rtmp = DB;
        5'h5 : U2Rtmp = DB;
        5'h6 : U2Rtmp = DB;
        5'h7 : U2Rtmp = DB;
        5'h8 : U3Rtmp = DB;
        5'h9 : U3Rtmp = DB;
        5'ha : U3Rtmp = DB;
        5'hb : U3Rtmp = DB;
        5'hd : DMtmp = DB;
5'hf : IMtmp = DB;
      endcase
    end
  assign U1R = U1Rtmp;
  assign U2R = U2Rtmp;
  assign U3R = U3Rtmp;
  assign DM = DMtmp;
  assign IM = IMtmp;

endmodule //dbmodule


module urfmodule(clk, READaddr1, READdata1, READaddr2, READdata2,
WRITEaddr1, WRITEdata1, WRITEenab1, BIWRITEaddr1, BIWRITEdata1,
RWRITEenab1, RWrw1, BIWRITEaddr2, BIWRITEdata2, RWRITEenab2,
RWrw2);
input clk;
input [1:0] READaddr1, READaddr2, BIWRITEaddr1, BIWRITEaddr2; //
read indexes
output [7:0] READdata1, READdata2;
inout [7:0] BIWRITEdata1, BIWRITEdata2;
tri [7:0] BIWRITEdata1, BIWRITEdata2;
reg [7:0] BIWRITEdata1reg, BIWRITEdata2reg;
input WRITEenab1, RWRITEenab1, RWrw1, RWRITEenab2, RWrw2;
input [1:0] WRITEaddr1; //write indexes
input [7:0] WRITEdata1;
parameter RFW = 'h8;
```

```
parameter RFD = 'h4;
reg [RFW-1:0] RF [RFD-1:0];


  always @(posedge clk)
    begin
      if (WRITEenab1 == 1)
        RF[WRITEaddr1] <= WRITEdata1;

      if ((RWRITEenab1 == 1) && (RWrw1 == 1))
        RF[BIWRITEaddr1] <= BIWRITEdata1;
      if ((RWRITEenab2 == 1) && (RWrw2 == 1))
        RF[BIWRITEaddr2] <= BIWRITEdata2;

    end

  assign READdata1 = RF[READaddr1];
  assign READdata2 = RF[READaddr2];

  always @(RWRITEenab1 or RWrw1 or BIWRITEaddr1 or RWRITEenab2 or
RWrw2 or BIWRITEaddr2)
    begin
      if ((RWRITEenab1 == 1) && (RWrw1 == 0))
        BIWRITEdata1reg = RF[BIWRITEaddr1];
      else
        BIWRITEdata1reg = 8'bz;

      if ((RWRITEenab2 == 1) && (RWrw2 == 0))
        BIWRITEdata2reg = RF[BIWRITEaddr2];
      else
        BIWRITEdata2reg = 8'bz;
    end

  assign BIWRITEdata1 = BIWRITEdata1reg;
  assign BIWRITEdata2 = BIWRITEdata2reg;

endmodule //urfmodule


module DM (clk,addr, data, rw, enable);
input clk,rw, enable;
input [4:0] addr;
inout [7:0] data;
reg [31:0] datareg;
parameter DMD = 'h20;
parameter DMW = 'h8;
```

```
reg [DMW-1:0] DM [DMD-1:0];

always @(enable or rw or addr)
  if (enable == 1)
    begin
      if (rw == 0)
        datareg = DM[addr];
      else
datareg = 32'bz;
    end
  else datareg = 32'bz;

always @(posedge clk)
    begin
      if (rw == 1)
        DM[addr] <= data;
    end

  assign data = datareg;


endmodule //DM


module IM (clk,addr, data, rw, enable);
input clk,rw, enable;
input [7:0] addr;
inout [43:0] data;
reg [43:0] datareg;
parameter IMD = 'h100;
parameter IMW = 'h2c;

reg [IMW-1:0] IM [IMD-1:0];

always @(enable or rw or addr)
  if (enable == 1)
    begin
      if (rw == 0)
        datareg = IM[addr];
      else
datareg = 32'bz;
    end
always @(posedge clk)
    begin
      if (rw == 1)
```

```
        IM[addr] <= data;
    end

  assign data = datareg;

endmodule //IM


endmodule //SPAM2
```