# GEZEL User Manual

**(Version January 18, 2005)**

**UCLA Electrical Engineering Deparment**
**420 Westwood Plaza**
**P.O. Box 951594**
**Los Angeles, CA 90095-1594**

*The only way an EDA tool can improve is by interacting with users.*

# Table of Contents

# Listings

# Roadmap to the User Manual

While this user manual can be read front-to-back, not all chapters are mandatory before you can do something useful with GEZEL. After reading Sections 1 to 4, you will be able to develop and simulate stand-alone GEZEL designs. Section 5 talks about VHDL code generation and is useful when you want to implement your GEZEL designs in hardware. Sections 6 and 7 consider cosimulation of GEZEL with other environments. Section 8 discusses customization of GEZEL by means of adding your own simulation primitives (library blocks).

- Section 1.0, *Overview,* summarize what GEZEL is about, and presents a taste of the GEZEL modeling language.
- Section 2.0, *Creating hardwired datapaths*, explains how to model datapaths, and how cycle-true code is developed using signals and registers.
- Section 3.0, *Creating sequential designs*, explains the various options for the design of datapath controllers.
- Section 4.0, *Simulating standalone GEZEL designs*, goes into the details of GEZEL simulation, and explains the various options for tracing and debugging.
- Section 5.0, *Converting GEZEL designs to VHDL,* explains how GEZEL designs can be converted into VHDL and fed into backend RT-simulation and RT-synthesis tools.
- Section 6.0, *Cosimulating GEZEL with Instruction Set Simulators*, explains how GEZEL is used in cosimulation.
- Section 7.0, *Cosimulating GEZEL with SystemC,* discusses the integration of GEZEL into a SystemC simulation.
- Section 8.0, *GEZEL Library Blocks,* gives an overview of existing GEZEL library blocks (such as RAM cells), and also explains how you can create your own.
- Appendix A, *Installing GEZEL*, explains how to download, configure and compile GEZEL. This includes the GEZEL kernel as well as various cosimulators that are included in the release.
- Appendix B, *Reuse in the GEZEL Kernel*, talks about the object-oriented architecture of the GEZEL kernel, including the implementation mechanism of library blocks.
- Appendix C, *References*, is a publication list for GEZEL and related tools (like the instruction-set simulators used for cosimulation).

The reader should have some familiarity with the following concepts:

1. The reader must be familar with basic hardware design concepts: registers and signals, gates, logic functions,digital arithmetic, and design of combinatorial and sequential logic. The reader must also have familiarity with the concept of logic simulation.
2. In order to use the cosimulator, the reader must be familiar with the C programming language and with C compilation and linking.
3. In order to use the output of the VHDL code generator, the reader must be familiar with VHDL modeling and the use of VHDL for RT-level simulation or synthesis.

4. To customize GEZEL, the reader must be familiar with the C++ programming language. If changes to the syntax must be done, familiarity with flex and/or bison are required.

# Acknowledgements

Much of what GEZEL is today was defined by the users of the tool. We would like to acknowledge the contributions of the following people (alphabetically), for their early adoption of the tool, their feedback on the tool, their contributions to the tool and their comments on the manual.

Sara Bocchio, ST

Herwin Chang, UCLA

David Hwang, UCLA

Bocheng Lai, UCLA

Jan Madsen, DTU

Bjarne Mathiesen, DTU

Yusuke Matsuoka, Renesas Technology Corp

Wei Qin, Boston University

Kazuo Sakiyama, KUL

Jørgen Steensgaard-Madsen, DTU

Students of the Spring 2003 EE201A class at UCLA

Andreas Vad Lorentzen, DTU

Oreste Villa, Politecnico di Milano

Ingrid Verbauwhede, UCLA and KUL

Shenglin Yang, UCLA

# 1.0 Overview

GEZEL is a language and open environment (LGPL) for exploration, simulation and implementation of domain-specific micro-architectures. GEZEL can help with the design of multiprocessor networks and embedded hardware. It has also been used as a teaching tool in class projects on VLSI architecture design. Highlights of the environment are as follows:

- A specialized language, called GEZEL, allows compact representation of the micro-architecture of domain-specific processors. GEZEL uses cycle-true semantics with dedicated modeling of control structures (FSMD).
- The simulation environment is scripted for fast edit-load-simulate cycles. No lengthy compiles. For cycle-true simulation, comparable performance to typical compiled-code environments is achieved at a fraction of the design setup (compile) time.
- The simulation back-end is an open C++ library that enables easy integration of GEZEL into different host environments. Cosimulation interfaces are available to several instruction-set simulators as well as to SystemC.
- GEZEL can be customized with user-supplied custom library blocks in C++.
- A design in the GEZEL language can be automatically translated to synthesizable VHDL. In addition, extra support for stimuli capture is available so that GEZEL simulations can be 'replayed' on the VHDL models.
- As a standalone environment, it works as a hardware exploration environment. When linked with an instruction-set simulator, it becomes a co-design environment.

Figure 1.1 shows an example of what GEZEL can do. In a multi-processor-on-chip (MPSOC), there are several, possibly heterogeneous cores next to dedicated hardware

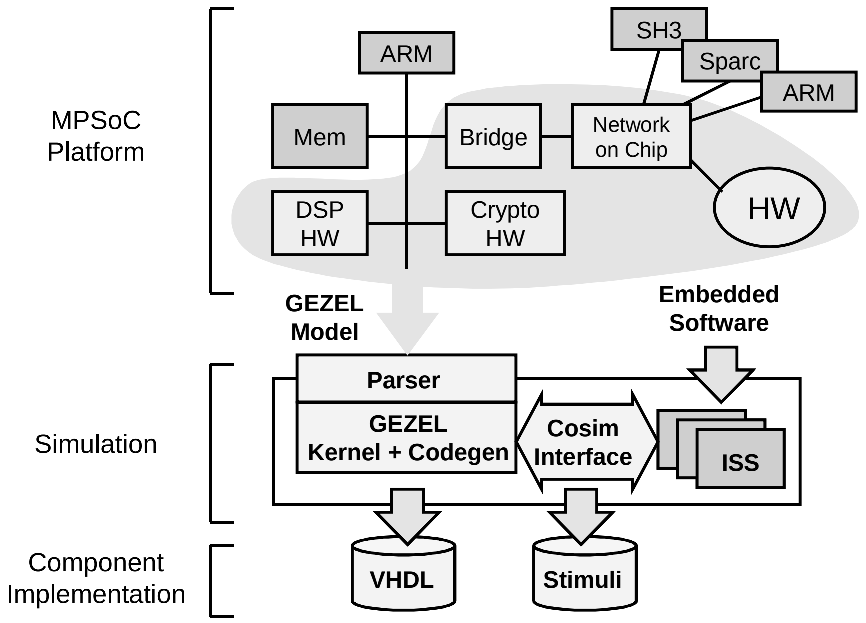


FIGURE 1.1. GEZEL can be used for coprocessor - and network-on-chip design

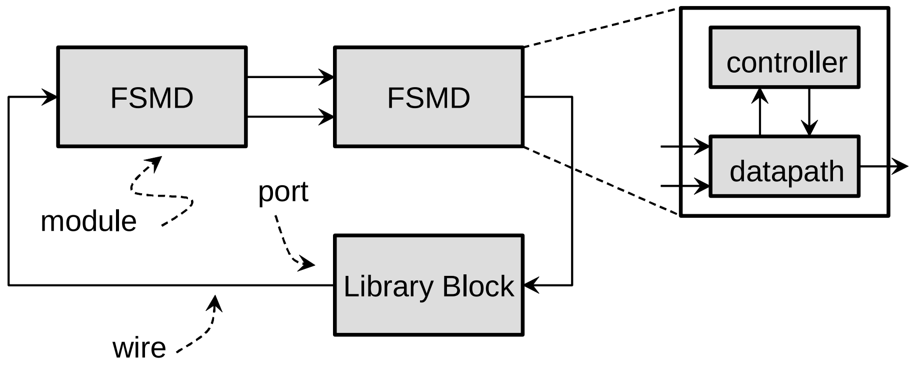


**FIGURE 1.2. A sample GEZEL model**

accelerators and interconnect. The hardware part can be captured in GEZEL language at cycle-true level. The GEZEL simulator can be linked to one or more instruction-set simulators to create an MPSOC platform simulator. This platform simulator reads the embedded software as well as GEZEL code to run a cycle-true simulation of the entire system. After validation, the GEZEL code can be converted into synthesizable VHDL code and handed over to the MPSOC implementation back-end.

In this manual, GEZEL features are discussed from a user-perspective. There is also a Language Reference Manual (LRM) where a more formal treatment of the GEZEL language and semantics is given.

## 1.1 The FSMD model of computation

The GEZEL language models hardware according to the semantics of a finite-state-machine with a datapath (FSMD). This section explains the FSMD model of computation. FSMD modeling will be covered later.

A model of computation helps to support a particular design style, by providing simulation semantics to a program. The model of computation of a C program is that of a procedural, sequentially executed language. The model of computation used for GEZEL is hardware-oriented, and is called FSMD (Finite State Machine with Datapath).

Figure 1.2 illustrates that GEZEL designs contain of a set of modules connected by wires. A module can be an FSMD or else a library block. An FSMD is expressed in the GEZEL language using FSMD semantics. A library block on the other hand is a build-in simulation primitive provided by the GEZEL simulator. Memory cells and cosimulation interfaces are examples of library blocks. An FSMD is a cycle-true model of a datapath with a controller. The datapath contains registers and hardware operators, and the controller sequences operations in the datapath.

Consider first a cycle-true simulation of a hardware module with only registers and operators and no controller, i.e. a fully hardwired datapath. Each register in the module is simulated in terms of two values, one being the next-state value, at the register input, and the other being the state value, at the register output. A cycle-true hardware simulation algorithm takes two simulation phases per clock cycle. During the first phase, the next-state of

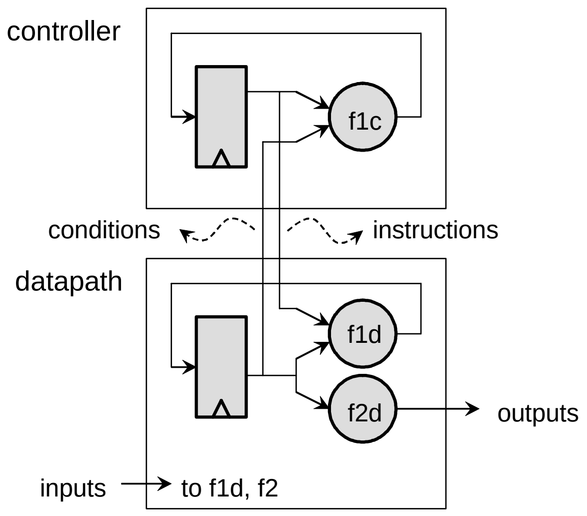


**FIGURE 1.3. The GEZEL FSMD Model consists of two cross-coupled finite state machines.**

the registers as well as the outputs of the datapath are evaluated based on the state of the registers as well as the inputs to the datapath.

```
next_state = f1(state, inputs)
  output = f2(state, inputs)
```

During the second phase, the newly obtained next-state values are copied into the state values so that the simulation of the next clock cycle can begin.

```
state = next_state
```

A digital cycle-true simulator executes these two phases in an alternating fashion. The behavior of the module therefore is completely defined by the functions `f1` and `f2`. They specify a finite state machine (FSM). Depending on the exact form of `f2`, one distinguishes a Moore-type FSM and a Mealy-type FSM. In a Moore FSM, the output value is only dependent on the previous-state, not on the current input.

An FSMD is a refined form of the above model that makes a distinction between two kinds of state in the hardware module. The first is called control-state, and the other is called datapath-state. Control-state represents the storage to work with control steps. Many algorithms, when mapped into digital hardware, decompose in a sequence of control steps. Datapath-state on the other hand holds data values required to evaluate the actual expressions of the algorithm.

The next-state function `f1` can be decomposed into a `f1d` to evaluate datapath state and a `f1c` to evaluate the control state. The datapath state machine uses the control step to implement instructions. The control state machine uses datapath state to implement conditional control steps. Thus, both state machines are cross-coupled. The first phase of the cycle simulation algorithm now becomes:

```
next_data_state = f1d(data_state, control_state, inputs)
  next_control_state = f1c(data_state, control_state)
  data_output = f2(data_state, control_state, inputs)
```

The second phase of the cycle simulation algorithm now becomes:

```
    data_state = next_data_state
 control_state = next_control_state
```

A graphical representation of these equations (Figure 1.3) shows that an FSMD consists of two cross-coupled finite state machines, one playing the role of controller, and the other playing the role of datapath. Information exchange between the two includes conditions (going from the datapath to the controller) and instructions (going from the controller to the datapath).

An FSMD offers important advantages over the basic FSM model when it comes to convenient modeling and mapping of algorithms.

- The explicit distinction of control and datapath state is something that a designer already does naturally. At the highest level, datapath state is naturally present in the state variables of an algorithm. Control state is introduced as a consequence of mapping the algorithm execution onto a time axis of clock cycles.
- A datapath and a controller have different modeling concepts. Datapaths are created by composition of expressions to make calculations. These expressions look like the ones from the C programming language. Controllers on the other hand are created by composition of state transition graphs.

A datapath and a controller have different logic implementation styles. Datapaths are regular, and can be created hierarchically as a composition of smaller elements. Controllers are irregular, and harder to create hierarchically.

An excellent reference on the underlying principles of FSMD modeling is Chapters 10 to 14 of the digital system book by Davio. Unfortunately this reference is out of print. More recently, SpecC has also introduced FSMD modeling.

- Davio, Deschamps, Thayse, “Digital Systems with Algortihm Implementation,” Wiley and Sons, 1983.
- Doemer, Gerstlauer, Gajski, “SpecC Language Reference Manual v 2.0,” 2002, available online from <http://www.cecs.uci.edu/~doemer/publications/SpecC_LRM_20.pdf>.

The relation between controllers and datapaths in GEZEL will be elaborated further in Section 3.0 on page 21. The next subsection presents a small example on the mapping of an algorithm into the FSMD model. The GEZEL syntax is introduced as well.

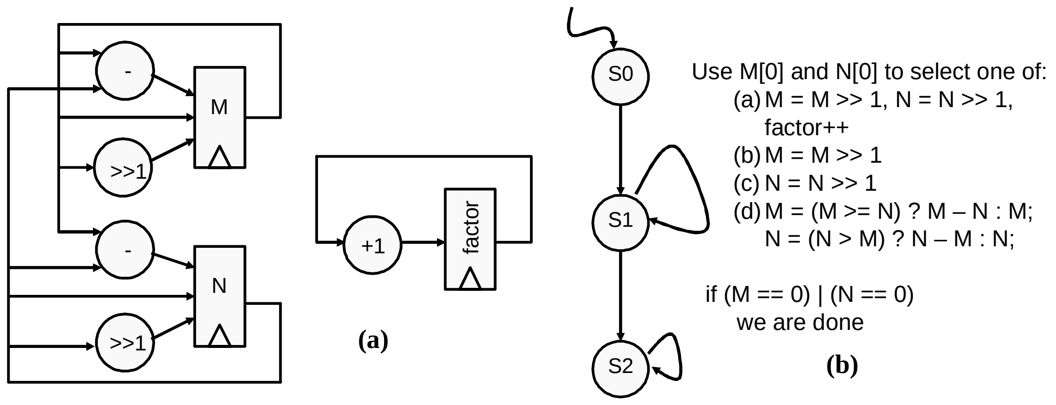


**FIGURE 1.4. The Euclid GCD Algorithm (a) Datapath and (b) Controller**

## 1.2 The Euclid algorithm

In this section, a simple processor that evaluates the greatest common divisor (GCD) using Euclid's algorithm will be modeled into GEZEL modeling and simulation. The particular variant used here is the version defined by Silver and Tersion (1962). This processor determines the GCD of two numbers M and N as follows.

- If M and N are even, then GCD(M,N) = 2 * (GCD(M/2, N/2))
- If M is even and N is odd, then GCD(M,N) = (GCD(M/2, N))
- If M is odd and N is even, then GCD(M,N) = (GCD(M, N/2))
- If M and N are odd, then, assuming M > N, GCD(M,N) = (GCD(M-N, N))

GEZEL models are written at the register-transfer (RT) level of abstraction. An example of such a model that evaluates the GCD algorithm is shown in Figure 1.4. The datapath holds three registers. Two of them, M and N, hold the values of M and N in the GCD algorithm. Each clock cycle, M and N are subtracted, shifted left, or unchanged. This is determined by the control step of the Euclid algorithm. An FSM controller is used to express conditional sequencing.

GEZEL allows a description close to Figure 1.4. The program in Listing 1 shows a processor that evaluates the GCD with one iteration per cycle. The processor has a data processing part (`dp`) and a control part (`fsm`). It also has a test-bench that generates two test values. The test-bench is connected to the processor in the system interconnect description.

The datapath description is in lines 1—20. This datapath has two 16-bit input ports `m_in` and `n_in`, and one 16-bit output port `gcd`. In contrast to Figure 1.4a, this is not a structural description. The datapath consists of a number of signal flow graphs, indicated with `sfg`. An `sfg` expresses a single clock cycle of behavior on the datapath. You can think of an `sfg` as an instruction that can be executed by the datapath. The signal flow graphs collect expressions that operate on the datapath registers, created in lines 3—6.

The controller is shown in lines 22—32. This is a finite state machine description that has three states, one of which is the initial state. Line 25 shows an unconditional state transition, starting at state `s0` and ending at state `s1`. During this state transition, the datapath will execute `sfg init` and `outidle`. A conditional state transition is expressed using if-then-else logic, such as shown in lines 26—30.

**LISTING 1. A GEZEL Program to evaluate greatest common divisor (GCD)**

```
1. dp euclid(in  m_in, n_in : ns(16);
2.           out gcd        : ns(16)) {
3.     reg m, n   : ns(16);
4.     reg done   : ns(1);
5.     reg factor : ns(4);
6.     sfg init     { m = m_in;
7.                    n = n_in;
8.                    factor = 0;
9.                    done = 0; }
10.    sfg flags    { done = ((m == 0) | (n == 0));  }
11.    sfg shiftm   { m = m >> 1; }
12.    sfg shiftn   { n = n >> 1; }
13.    sfg reduce   { m = (m >= n) ? m - n : m;
14.                   n = (n <  m) ? n - m : n; }
15.    sfg shiftf   { factor = factor + 1; }
16.    sfg outidle  { gcd = 0; }
17.    sfg complete {
18.                  gcd = ((m > n) ? m : n) << factor;
19.                  $display("cycle=", $cycle, " gcd=", gcd); }
20. }
21.
22. fsm euclid_ctl(euclid) {
23.   initial s0;
24.   state s1, s2;
25.   @s0 (init, outidle) -> s1;
26.   @s1 if (done)               then (complete)            -> s2;
27.       else if ( m[0] &  n[0]) then (reduce,outidle,flags)-> s1;
28.       else if ( m[0] & ~n[0]) then (shiftn,outidle,flags)-> s1;
29.       else if (~m[0] &  n[0]) then (shiftm,outidle,flags)-> s1;
30.                     else (shiftn,shiftm,shiftf,outidle,flags)-> s1;
31.   @s2 (init, outidle) -> s2;
32. }
33.
34. dp test_euclid(out m, n : ns(16)) {
35.   sfg run {
36.     m = 2322;
37.     n = 654;
38.   }
39. }
40. hardwired h_test_euclid(test_euclid) {run; }
41.
42. system S {
43.   bresen(x1, y1, x2, y2);
44.   test_bresen(x1, y1, x2, y2);
45. }
```

The test-bench for the GCD processor is shown in lines 34—45. We will apply the constant values 2332 and 654 as test values. This GEZEL description can be simulated with the `fdlsim` simulation tool. To simulate 25 cycles from this description, execute the command line

```
> fdlsim euclid.fdl 25
cycle=23 gcd=6
Activity(%) on 4 registers: 29 (29/100)
FDL Cycles 25, sleeping cycles 0(0%)
```

The simulator reports that the GCD of the two test values is 6, and that this value is obtained at cycle 23 of the simulation. This line is printed using a simulation directive as shown on line 19 of Listing 1

An interesting feature of GEZEL is that it does not require a compilation phase. When the simulator starts, it will parse in the GEZEL description and immediately start the simulation. This way the design and evaluation of hardware models becomes interactive.

The GEZEL parser generates error messages immediately when it encounters an error. For example, when line 13 of Listing 1 contains `'sff reduce'` then the following error message appears:

```
> fdlsim euclid.fdl 25

*** (line 13) Syntax Error

(10)      sfg flags   { done = ((m == 0) | (n == 0));  }
(11)      sfg shiftm  { m = m >> 1; }
(12)      sfg shiftn  { n = n >> 1; }
(13) >>>  sff reduce  { m = (m >= n) ? m - n : m;

Failed to parse euclid.fdl
```

When the Euclid design simulates correctly, the same code can be converted into VHDL. A companion tool for the GEZEL standalone simulator is a GEZEL-to-VHDL code generator called `fdlvhd`. The tool is run from the command line as illustrated next.

```
> fdlvhd euclid.fdl
Pre-processing System ...
Output VHDL source ...
----------------------------
Generate file: euclid.vhd
Generate file: test_euclid.vhd
Generate file: system.vhd
```

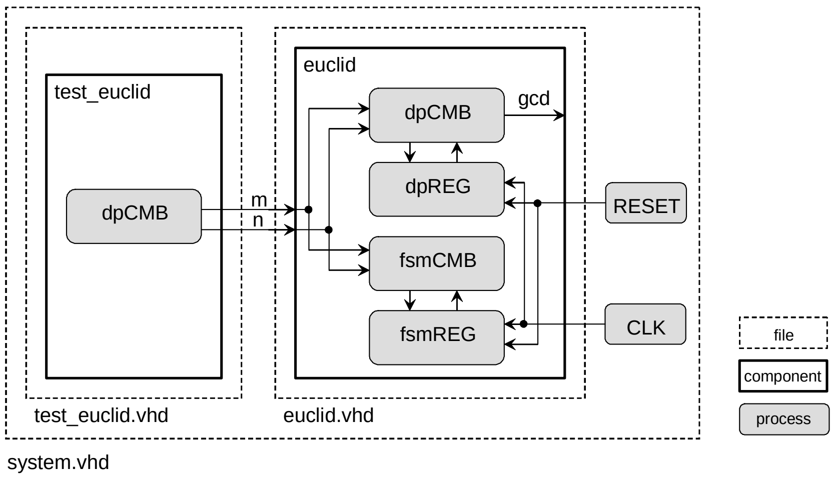


**FIGURE 1.5. Component Hierarchy and Process of the generated VHDL code.**

Three files are generated, and the component/ process hierarchy is illustrated in Figure 1.5. Each datapath module in GEZEL is created in a separate file. A synchronous VHDL modeling strategy creates separate processes for combinatorial logic and for registers. The datapath and controller FSM are each created as separate sets of processes.

# 2.0 Creating hardwired datapaths

Datapaths are the basic building blocks in GEZEL, similar to a *module* in Verilog or an *entity* in VHDL. First, the essential datapath elements are considered: registers and signals, and expressions. Then datapath definitions are introduced that can embed these expressions. Finally, the different methods of datapath composition are discussed, either by creating interconnections between ports, or else by structural hierarchy: encapsulating one datapath into another one.

## 2.1 Registers and signals

GEZEL models synchronous, single-clock designs. Yet, a clock signal is not present in GEZEL language, it is implicit in the design description. By looking at a GEZEL program, you can say precisely how it will behave as a clock-cycle true description. You can do this by looking at the kind of variables it uses in calculations. GEZEL has two kinds of variables: *signals* and *registers*.

A signal can hold a value within a single clock cycle. It has the same meaning as a wire in an actual implementation. A signal also has a name and a type and is created with the `sig` keyword. For example, a signal with name `v12` and type `ns(12)` is created as follows.

```
sig v12 : ns(12);
```

This type `ns(12)` stands for a 12-bit unsigned number. Signal `v12` can hold values from 0 to 4095. When you force this signal to hold values outside of this range, precision loss will occur. This will be discussed in Section 2.2, “Expressions,” on page 10. There is one other type available for values, called `tc(n)`. This type represents arbitrary-length signed numbers with two’s complement representation. For example, to create the equivalent of a C integer on a 32-bit machine, use the following definition.

```
sig aCinteger : tc(32);
```

Registers are used to store values over multiple clock cycles. In contrast to signals, register variables have two values: a current-value and a next-value. The current-value is the value available at the output of a register, so it is the value obtained when reading from the register. The next-value is the value at the input of the register, so it is the value that is being written into the register. A register is created in the same way as a signal but uses the `reg` keyword. A 16-bit unsigned register for example is created as

```
reg r : ns(16);
```

The register lies at the basis of clock-cycle-true behavior. There are implicit simulation semantics tied to the register. At the start of each clock cycle, the next-value (of the previous clock cycle) is copied into the current-value (of the current clock cycle). In between clock edges, the next-value is updated based on the current-value, constants and inputs. This way, it is possible to create clock-cycle true descriptions without mentioning the clock explicitly.

The initial value of a register is zero (0), while the initial value of a signal is undefined.

## 2.2 Expressions

Expressions enable calculations with signals and registers. Expressions are formed using operators that reference the names of signals and registers. For example, an addition of two signals `b` and `c` into signal `a` looks like

```
a = b + c;
```

When `a` has insufficient precision to hold all possible combinations of the sum `b + c`, precision loss can occur. For example, assume the following types for a, b and c:

```
sig a, b, c : ns (8);
```

Clearly, when `b + c` is bigger than 256, the result cannot be stored in `a`. GEZEL will throw out bits at the most-significant side of the result (overflow). If `b + c` is 260, then the resulting value in a will be 4 (260 = 256 + 4).

In some expressions, *intermediate values* will occur. In the above expression, `b + c` is such an intermediate value. A more obvious example is

```
a = ((b+b) + (c+c));
```

Here, brackets are used to indicate the order in which this expression is to be evaluated. First, the sums `b+b` and `c+c` are obtained. These two intermediate values are combined and assigned to `a`. Intermediate values need a type, too.

GEZEL uses a default type rule to choose the type of intermediate results. This is rule consists of two parts: (a) the result of an operation is the maximum wordlength of the operands and (b) if any of the operands is signed, then the result will be signed as well. There are exceptions to this rule which will be indicated later.

Expressions combine signals and registers with operators. Operators have a *precedence*, a preferred order of evaluation. For example, in an expression such as

```
a = b * b + c * c;
```

the multiplications (*) will be performed before the additions (+), because multiplication has a higher precedence than addition. Precedence rules can be modified by using round brackets. The following bullets introduce the different operators that can be used in expressions, starting at the ones with low precedence and going up to high-precedence operations.

- Assignment and Selection

The assignment operation updates the value of a signal or register. The selection operation conditionally extracts the value of a signal or register.

| | |
|---|---|
| `a = expression;` | The assignment operations assigns the value of `epxression` into `a`. At the moment of assignment, the value of `expression` is casted in `a` (cfr the casting operation). |
| `b ? c : d` | The selection operation implements choice. The value of `b` is evaluated. When it is nonzero, the expression evaluates to `c`. When it is zero, the result is `d`. |

- Bitwise Logical Operations

  Bitwise logical operations combine two bitpatterns into a new bitpattern. The bits at corresponding indices are combined using a single-bit logical operations. The logical operations are Inclusive Or, Exclusive Or, and And.

| | |
|---|---|
| `b \| c` | The bit pattern in b is IOR-ed with the bit pattern in c. |
| `b ^ c` | The bit pattern in b is XOR-ed with the bit pattern in c. |
| `b & c` | The bit pattern in b is AND-ed with the bit pattern in c. |
| `~ b` | The bit pattern in b is inverted (This operation has higher precedence than all two-operand operations). |

- Comparison Operations

  Comparison operations compare the value of two expressions and yield a true-or-false result. The value true or false is represented as a 1-bit unsigned number (`ns(1)`), with 1 indicating true, and 0 indicating false.

| | |
|---|---|
| `a == b` | True if the value of `a` is equal to the value of `b`. |
| `a != b` | True if the value of `a` is different from the value of `b`. |
| `a < b` | True if the value of `a` is smaller than the value of `b`. |
| `a > b` | True if the value of `a` is bigger than the value of `b`. |
| `a <= b` | True if the value of `a` is smaller than or equal to the value of `b`. |
| `a >= b` | True if the value of `a` is bigger than or equal to the value of `b`. |

- Arithmetic Operations

  Arithmetic Operations do calculations on all of the bits of a signal or register, treated as an unsigned number or else a two's complement signed number.

| | |
|---|---|
| `a << b` | `a` is shifted left over `b` positions. The wordlength of the result is equal to the wordlength of `a` plus 2-to-the-power (wordlength of `b`). |
| `a >> b` | `a` is shifted right over `b` positions. The wordlength and the sign of the result are equal to that of `a` (arithmetic shift). |
| `a + b` | `a` is added to `b`. |
| `a - b` | `b` is subtracted from `a`. |

| | |
|---|---|
| `a * b` | `a` and `b` are multiplied. |
| `a % b` | modulo: the remainer of the division of `a` by `b`. The sign of the divisor is ignored. The result is always positive. |
| `a # b` | Bit concatenation. Equivalent to `(a << wordlength(b)) \| b)` |
| `- a` | Negate the value in `a` (this operation has higher precedence than all two-operand operations). |

- Cast Operation

  A cast operation converts the value of a signal into one with another type. This way, it is possible to convert for example a 5-bit unsigned number into a 6-bit signed number. When the target type has enough bits, no precision will be lost. For two's complement signed numbers, a concept called *sign extension* is applicable. Sign extension preserves the sign of a two's complement number when the wordlength increases. When the target type has insufficient bits, some precision can be lost. Bits are chopped off at the most-significant side. The resulting bitpattern is interpreted as a signed/unsigned number of the targeted wordlength.

  For example, if `a` is `ns(8)` and holds the value 7, and `b` is `tc(4)`, then

```
b = (tc(3)) a;
```

  will leave the binary pattern `0b1111` in `b`, which is interpreted as `-1`.

| | |
|---|---|
| `(typespec) expr` | Converts the type of `expr` to `typespec`. |

- Unary Operations

  A unary operation has a single operand. There is a bitwise NOT operator and a negation operation, see 'Bitwise Operations' and 'Arithmetic Operations'.

- Bit Selection Operation

  A bit selection operation extracts part of a bitpattern in a word. There is a single-bit format as well as a bitvector format.

| | |
|---|---|
| `a[n]` | Returns bit `n` from `a` as a `ns(1)` number. `n` has to be a positive constant. If `n` is bigger than the wordlength of `a`, `0` is returned. |
| `a[m:n]` | Return bitvector from bit `m` to bit `n` (`n >= m`) from `a` as a `ns(n-m+1)` number. `n` and `m` have to be positive constants. If a bit index goes out of the wordlength range of `a`, `0` is returned for that bit. |

- Lookup Table Operation

  A Lookup Table Operation offers access to a constant array, which is defined earlier in the code. The lookup table content needs to be defined first, after which it can be accessed using a lookup table operation.

  A Lookup Table definition is done by enumerating all the elements in the lookup table in a comma separated list as follows:

```
lookup a : ns(8) = {15, 22, 36, 0x4f};
```

This defines a lookup table `a` which holds elements of type `ns(8)`. The table holds 4 elements. The element at index position 0 is 15 and the element at index position 3 is 0x4f (79).

The lookup table access operation simply access the array using the index in between round brackets. For example, to access the third element of `a`, one would use

```
a(2)
```

## 2.3 Signal flow graphs

The cycle-true execution model of GEZEL expresses concurrency by allowing multiple expressions to be evaluated in the same clock cycle. A set of expressions that execute together in the same clock cycle are grouped together in a *signal flowgraph*. A signal flowgraph creates a symbolic name to refer to these expressions.

Consider the design of a Viterbi Butterfly operation (a well-known operation in convolutional decoding). This operation processes tuples of data according to an operation called add-compare-select

$$y_1 = \min( d_1 + a, d_2 - a ) \qquad \text{(EQ 1)}$$

$$y_2 = \min( d_1 - a, d_2 + a ) \qquad \text{(EQ 2)}$$

Assume the following set of signals and registers.

```
sig a1, s1, a2, s2 : ns(8);  // intermediate signals
reg d1, d2, y1, y2 : ns(8);  // input and output tuple
reg a : ns(8);
```

The signals flowgraph of expressions that implements this equation can be as follows

```
sfg acs {
  a1 = d1 + a;
  s1 = d1 - a;
  a2 = d2 + a;
  s2 = d2 + a;
  y1 = (a1 > s2) ? s2 : a1;
  y2 = (s1 > a2) ? a2 : s1;
}
```

The keyword `sfg` also indicates a name for a group of expressions. In this case, this set is called `acs`.

An `sfg` can hold an arbitrary number of expressions. All expressions within a single `sfg` are concurrent within one clock cycle. The order in which expressions are evaluated is independent of the order in which they appear in the `sfg` definition. Rather, the order is determined by the *data precedences* of signals and registers. A register can always be read, at any moment during a clock cycle. As discussed in Section 2.1 on page 9, a register has both a current value and a next value. For a signal, this is not the case. A signal has only an immediate value, valid within a single clock cycle. Thus, a signal has to be written first before it can be read. It has to be written the first time within a clock cycle based on values in registers and constants. As a consequence of this property of signals and registers, the order of expressions wihtin an `sfg` becomes irrelevant. For example, if you would write:

```
sfg acs2 {
  y1 = (a1 > s2) ? s2 : a1;
  y2 = (s1 > a2) ? a2 : s1;
  a1 = d1 + a;
  s1 = d1 - a;
  a2 = d2 + a;
  s2 = d2 + a;
}
```

then, when evaluating `y1`, the GEZEL simulator will notice that none of the signals `a1`, `a2`, `s1` and `s2` are available yet. Consequently, it would first find a current value for these signals. So, `sfg acs2` behaves exactly the same as `acs`.

## 2.4 Datapath modules

A datapath corresponds to a *module* in Verilog or an *entity* in VHDL. It is a piece of hardware logic that is treated as a single entity by subsequent RT- and logic synthesis tools. A datapath combines a number of `sfg` with a list of input and output signals. An `sfg` can be thought of as an instruction for that datapath.

A datapath is the smallest GEZEL unit that can be simulated. So, subsequent examples will be fully self-contained programs rather than snippets. This requires however the use of a few additional language constructs which, for the time being, will only be explained very briefly.

A special type of datapath is one in which there is only a single `sfg`. For such a datapath a special type of controller is used, a *hardwired controller*. Such a controller will instruct the datapath to execute a single `sfg` inside of the datapath each clock cycle again. The term *hardwired datapath* will be used to indicate a datapath with a single signal flowgraph, under control of a hardwired controller.

Here is an example of a 2-bit counter as a hardwired datapath.

**LISTING 2. A 2-bit counter as a hardwired datapath**

```
dp counter(out value : ns(2)) {
  reg c : ns(2);
  sfg run {
    value = c;
    c = c + 1;
    $display(“Cycle “, $cycle, “: counter = “, value);
  }
8. }
9. hardwired c_counter(counter) {run; }
10.
11. system S {
12.   counter(o);
13. }
```

This datapath has a single output port called `value`. An output port also has a type, indicated after the colon following the port name. The ports define the outline of the datapath. The only way an ‘outsider’ can access the datapath is by reading/writing values on the datapath ports.

On line 2, we create a 2-bit register. This register is local to the datapath `counter`. It can be accessed only from within the datapath.

On line 3—7, we define a signal flowgraph called `run`. It contains, besides expressions, also a *directive* on line 6. A GEZEL directive does affect how the simulator behaves, but it does not affect the simulation outcome. In this case we are using the display directive, which is used to print out values on the datapath. One special variable that is accessed is called `$cycle`. This variable returns the current simulation cycle. Thus, the effect of the display directive will be to print out the current simulation cycle as well as the output value of the counter.

On line 9, a controller for this datapath is created. A datapath cannnot do anything useful without a controller. The primary task of a controller is to select what signal flowgraph should execute in each clock cycle. A `hardwired` controller is a controller that supports only a single signal flowgraph, which is selected in the braces folllowing the controller definition.

Finally, on lines 11—13, the toplevel of the system is expressed. Such a toplevel interconnects datapaths. In this case, there is only a single datapath.

The counter of Listing 2 can be simulated by means of the `fdlsim` standalone GEZEL simulator. To simulate 6 clock cycles, we execute

```
> fdlsim listing2.fdl 6
Cycle 1: counter = 0
Cycle 2: counter = 1
Cycle 3: counter = 2
Cycle 4: counter = 3
Cycle 5: counter = 0
Cycle 6: counter = 1
```

As expected, the counter counts up to three and then wraps around.

A datapath definition thus consists of three elements: An IO definition, a definition of local signals and registers, and a set of signal flowgraphs. The IO definition can create input — as well as output ports. For example, a simple ALU that can add, subtract and accumulate would look as follows.

```
dp alu(in x, y : ns(8); out z : ns(8)) {
  reg acc : ns(8);
  sfg add {
    z = x + y;
  }
  sfg sub {
    z = x - y;
  }
  sfg accumulate {
    acc = acc + x;
    z   = acc + x;
  }
  sfg rst {
    acc = 0;
    z   = 0;
  }
}
```

There are four signal flowgraphs in this example. The datapath has two inputs, `x` and `y`, and one output, `z`. There is an internal accumulator register, `acc`. There is one signal flowgraph call `rst`. This will be used to reset the accumulator register. During this reset operation, we will also drive the output of the datapath to zero.

Not all datapath definitions that one can write down in GEZEL are valid. There are four rules to which a datapath definition must conform. When any of those rules are violated, then either the GEZEL parser will reject your code, or else a runtime error message will be triggered. The four rules are enumerated below.

1. During any clock cycle, all datapath outputs are defined. This means that datapath outputs must always appear at the lefthand-side of an assignment expression inside of any active sfg.
2. During any clock cycle, no combinatorial loop between signals can exist. This happens when there is a circular dependence on signal values, i.e. signal `a` is used to define signal `b`, and signal `b` is used to define signal `a`. This implies that all signal values will eventually only be dependent, during any clock cycle, on datapath inputs, datapath registers and constant values.
3. If an expression uses the value of a signal during a particular clock cycle, then that signal must also appear at the left-hand side of an assignment expression in the same clock cycle.
4. Neither registers, nor signals or datapath outputs can be assigned more than once during a clock cycle. A special case of this is that a datapath input cannot be assigned inside of a datapath, because a datapath input must be driven by the output of another datapath.

Here are a few examples of erroneous signal flowgraphs.

**LISTING 3. A number of erroneous datapaths**

```
1. // WRONG: output v is not always defined
```

```
dp bad1(out v : ns(1)) {
  sfg run {}
}
hardwired hbad1(bad1) {run; }

// WRONG: a combinatorial loop between signals
dp bad2 {
  sig a, b : ns(1);
  sfg run {
   a = b + 1; // a defines b, b defines a
   b = a + 1; // and both are signals (not registers)
  }
}
hardwired hbad2(bad2) {run;}

// WRONG: dangling signal
dp bad3 {
  sig a, b : ns(1);
  sfg run {
   a = b + 1;  // b is unknown
  }
}

// WRONG: a signal is assigned more than once
dp bad4 {
  sig a : ns(1);
  sfg run {
    a = 1;
    a = 5;
  }
}
```

## 2.5 Datapath module interconnections

There can be more than one datapath in a system, and at that moment all datapaths are concurrently active. Such datapaths are interconnected in a top-level netlist, included in a `system` block. Datapath inputs must always be connected to a datapath output. Multiple datapath inputs can be connected to a single datapath output (so-called star-nets). However, any single datapath input can be driven by only a single datapath output.

Nets are defined by using symbolic names as parameters in the IO list of a datapath. For example, assume a pipeline with tree stages. The datapath definitions and system interconnect looks as follows.

```
dp stage1(out o : ns(8)) { .. }
dp stage2(in i : ns(8); out o : ns(8)) { .. }
dp stage3(in i : ns(8); out o : ns(8)) { .. }

system S {
  stage1(a);
  stage2(a, b);
  stage3(b, c);
}
```

There are three nets defined, which are called `a`, `b` and `c`. The matching of a net to a datapath IO port is positional. For example, net `b` matches on port `o` from stage2 and on port `i` from stage3.

The type of a net is defined by the type of the driving port. If there is a mismatch between ports, then no error will occur. However, each input port will perform a cast operation to convert the value on the system net from the type at driving output port to that of the input port.

Finally, it must be mentioned that the simulator will only include a datapath in the simulation cycle if it is included inside of the system block. Referring to the above example, if the system block would only include `stage1`, then `stage2` and `stage3` would be ignored in the simulation *even* if they are defined before as `dp`.

## 2.6 Datapath cloning

Sometimes, multiple copies of one and the same datapath are needed. GEZEL provides a cloning operation to create such an identical copy of a single datapath. The next example shows how three identical AND gates can be created by defining one and then cloning the first AND gate two times.

```
dp andgate(in a, b : ns(1); out q : ns(1)) {
  sfg run {
    q = a & b;
  }
}
hardwired h_andgate(andgate) {run; }

dp andgate2 : andgate
dp andgate3 : andgate
```

Before the cloning operator can be applied, the cloned datapath must have defined a controller as well. The controller of a datapath will be included in the cloning operation. Cloning creates an identical but independent copy. If the parent datapath includes a register, then the cloned datapath will contain its’ own register.

## 2.7 Structural Hierarchy

The system block mentioned before expresses a single level of hierarchy. However, it is also possible to use a more elaborated form of structural hierarchy, by including datapaths within datapaths. For this purpose, GEZEL provides the keyword `use`. Consider the example of a 4-input AND gate.

### LISTING 4. A 4-input AND gate using structural hierarchy and three 2-input AND gates

```
1. dp andgate(in a, b : ns(1); out q : ns(1)) {
2.   sfg run {
3.     q = a & b;
4.   }
5. }
6. hardwired h_andgate(andgate) {run; }
7.
8. dp andgate2 : andgate
9. dp andgate3 : andgate
10.
11. dp fourinputand(in a, b, c, d : ns(1); out q : ns(1)) {
12.   sig s1, s2 : ns(1);
13.   use andgate ( a,  b, s1);
14.   use andgate2( c,  d, s2);
15.   use andgate3(s1, s2,  q);
16.   sfg run {
17.      $display(a," ", b, " ", c, " ", d, " -> ", q);
18.   }
19. }
20. hardwired h_fourinputand(fourinputand) {run; }
21.
22. dp tst(out a, b, c, d : ns(1)) {
23.  reg n : ns(4);
24.  sfg run {
25.    n = n + 1;
26.   a = n[0]; b = n[1]; c = n[2]; d = n[3];
27.  }
28. }
29. hardwired h_tst(tst) {run; }
30.
31. system S {
32.  tst(a, b, c, d);
33.  fourinputand(a, b, c, d, q);
34. }
```

Lines 11—20 create a four-input AND gate using three two-input AND gates. A `use` statement allows to include a two-input AND gate inside of the four-input AND gate. Connections can be made to datapath inputs, outputs or local signals. Of course, the semantic requirements enumerated earlier must be obeyed.

Lines 22—29 define a testbench that enumerates all 4-bit input patterns by decomposing the bits of a counter. Finally, lines 31—34 interconnect the testbench to the four-input AND gate in a system block.

We can now simulate this design for 16 clock cycles, and observe all combinations of the AND gate to verify it works correctly:

```
> ../../devel/build/bin/fdlsim listing4.fdl 16
0 0 0 0 -> 0
1 0 0 0 -> 0
```

```
     0 1 0 0 -> 0
     1 1 0 0 -> 0
     0 0 1 0 -> 0
     1 0 1 0 -> 0
     0 1 1 0 -> 0
     1 1 1 0 -> 0
     0 0 0 1 -> 0
     1 0 0 1 -> 0
     0 1 0 1 -> 0
     1 1 0 1 -> 0
     0 0 1 1 -> 0
     1 0 1 1 -> 0
     0 1 1 1 -> 0
     1 1 1 1 -> 1
```

This completes basic modeling techniques for datapaths. The next section covers the modeling of controllers, that enable the use of datapath with multiple signal flowgraphs.

# 3.0 Creating sequential designs

This section covers the link between a datapath with multiple signal-flowgraphs (instructions), and a controller. Information on how to model datapaths and signal flowgraphs can be found in Section 2.0, “Creating hardwired datapaths,” on page 9. The generic model of control is FSMD. This section covers this model by itself as well as the representation of this model in GEZEL.

## 3.1 FSMD models

The control/datapath model of GEZEL is based on a more generic form of register-transfer level modeling called Finite State Machine and Datapath, or FSMD for short. An FSMD model expresses both datapath operations as well as control operations. It makes a clear distinction however between what is control and what is data processing. Recall from Section 1.1 on page 2 that an FSMD consists of two cross-coupled state machines. One plays the role of the controller, the other plays the role of the datapath. Information exchange between the two includes conditions (going from the datapath to the controller) and instructions (going from the controller to the datapath).

An FSMD provides separate modeling for data processing and for control processing. That is for a good reason, in practice there are many differences between the controller and the datapath. First, the modeling style for the two is different. Datapaths are modeled with expressions on signals and registers. Controllers are modeled with state transition graphs. Secondly, the logic implementation style of the two also shows differences. A datapath with operators typically exhibits a regular logic style. Think for example of a ripple carry-chain adder. A controller on the other hand exhibits an irregular logic style.

The FSMD concepts map as follows to the GEZEL model.

- Instructions are created by selecting one or more `sfg` out of a datapath. A single `sfg` can be directly referred to by its name. A set of `sfg` is enumerated as a comma-separated list in between brackets. For example, assume a datapath is defined as follows.

  ```
  dp adp(out a : ns(3)) {
    sig k : ns(2);
    sfg f1 { a = 3; }
    sfg f2 { k = 2;
             a = 2;}
    sfg f3 { k = 1; }
  }
  ```

  Then, the following are valid instructions:

  ```
  f1
  f2
  (f1, f3)
  ```

  Examples of invalid instruction are:

```
f3
(f1, f2)
```

These are invalid because the violate the semantic requirements for datapath models (See Section 2.4, “Datapath modules,” on page 14).

When an instruction is executed during a particular control step of a controller, then that will imply execution of the sfg included in the instruction as well.

- Conditions are created out of logical expressions on registers in the datapath. When conditions are extracted out of datapath inputs or signals, the GEZEL parser will issue a warning. The reason is that GEZEL selects the instruction to execute right at the start of a clock cycle. Before this can be done, any required conditions need to be defined. At the start of a clock cycle however, the only stable values are constants and register outputs. A user can still continue the simulation despite the presence of this warning. However, one must realize at that moment there is a potential risk for anticausal simulation effects (e.g. using the value of a signal before it is available). Therefore, when this warning occurs one must consider if the code can be written such that no warnings appear.
- The connection between a datapath and a controller is established by refering the name of the datapath while creating the controller. Some earlier examples of this could be seen with the `hardwired` controller:

```
dp adp(out a : ns(3)) {
  ..
}

hardwired h_adp(adp) { f1; }
```

In this example, a controller called `h_adp` is created and attached to datapath `adp`.

## 3.2 Sequencer datapath controllers

Besides the trivial `hardwired` controller (See Section 2.4 on page 14), the simplest controller is the `sequencer`. As the name indicates, a `sequencer` will execute a set of instructions sequentially, without taking any conditions into account.

A typical case where sequencers are useful is for static, fixed schedules. Consider for example a 4-tap decimating averaging filter. Such a filter reads four subsequent samples, integrates and dumps the sum of the samples at every fourth sample.

**LISTING 5. A 4-tap decimating averager using a sequencer**

```
dp avg(in i : ns(8); out o : ns(8)) {
  reg acc : ns(9);
  sfg phase0  { acc = i; o = 0; }
  sfg phase12 { acc = acc + i; o = 0;}
  sfg phase3  { o   = (acc + i) >> 2;}
}
sequencer h_avg(avg) { phase0;
                       phase12;
9.                         phase12;
10.                        phase3;}
11.
12. dp tst(in i : ns(8); out o : ns(8)) {
13.    reg a : ns(8);
14.    sfg run {
15.      o = a;
16.      a = a + 2;
17.      $display("C", $cycle, ": i=", i, " o=", o);
18.    }
19. }
20. hardwired h_tst(tst) {run;}
21.
22. system S {
23.   avg(i, o);
24.   tst(o, i);
25. }
```

An averaging filter has four phases. As the datapath in lines 1—6 illustrates, there is an initialization instruction (`phase0`), an accumulation instruction (`phase12`) and a termination instruction (`phase3`). The controller for this datapath is a sequencer with four steps, as shown in lines 7—10. Lines 12—20 show a simple testbench that will feed a string of even numbers to this four-phase averager. Finally, lines 22—25 show the system interconnect function.

This description can be simulated for 10 clock cycles to yield the following output. One can verify that indeed (0+2+4+6)/4 is 3.

```
> fdlsim listing5.fdl 10
C1: i=0 o=0
C2: i=2 o=0
C3: i=4 o=0
C4: i=6 o=3
C5: i=8 o=0
C6: i=a o=0
C7: i=c o=0
C8: i=e o=b
C9: i=10 o=0
C10: i=12 o=0
```

An important motivation for developing FSMD models, instead of plain hardwired datapaths, is that an FSMD allows to express operation sharing in an elegant way. Consider the descriptions in `phase0`, `phase12` and `phase3`. They specify two assignments on an accumulator register and three assignments to an output port *without* the use of a multiplexer. When the same behavior would be represented in a single `sfg`, it would look like this:

```
reg phase : ns(2);
sfg singlephase {
  acc = (phase == 0) ? i : acc + i;
  o   = (phase == 3) ? (acc + i) >> 2 : 0;
```

```
    phase = phase + 1;
  }
```

While there are cases in which this description style is useful, in general it requires modeling overhead and it prevents operation sharing. For example, the addition operation executes in different phases (clock cycles), so the implementation of the averaging filter could reuse the adder. With a writing style with multiple `sfg` such as in Listing 5, the GEZEL VHDL code generator will enable this sharing. It cannot do this with a writing style that uses a single `sfg` such as above.

## 3.3 Finite state machines

A Finite State Machine implements conditional control sequencing on a datapath. The control model is captured by a state transition graph. A Finite State Machine can be in a well-defined number of states. One of these states is the *initial state*, it is the state the FSM is in when it first initializes.

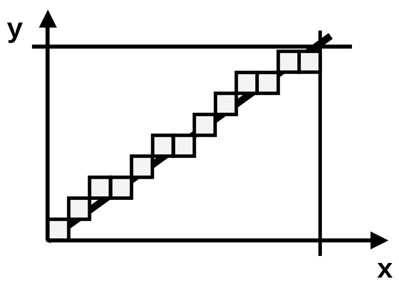
A Finite State Machine will take one state transition per clock cycle. During this state transition, a datapath instruction (one or more `sfg`) can be executed. A state transition can be conditional. In that case, the condition is based on the values of registers in the datapath (or on logical expressions directly derived from it). When state transitions are conditional, then the set of conditions must be complete. This means that, for every *if* (true-branch), there must be a complimentary *else* (false-branch).

Consider the following simple example of FSM modeling. The sequencer of Listing 5 can also be written as an FSM as follows.

```
fsm h_avg(avg) {
  initial s0;
  state s1, s2, s3;
  @s0 phase0  -> s1;
  @s1 phase12 -> s2;
  @s2 phase12 -> s3;
  @s3 phase3  -> s0;
}
```

This description creates four states, called `s0`, `s1`, `s2` and `s3`. `s0` is the initial state, the others are normal states. A state transition indicates the start state with the `@` symbol, and the target state with an arrow (`->`). In between, a datapath instruction is indicated. A single `sfg` can be written as such, a group of `sfg` is specified as a comma-separated list in between round brackets.

Next is an example with slightly more complicated FSM control. The example is a raster line drawing routine, known as the Bresenham Algorithm. The strong point of this algorithm is that it can draw lines of arbitrary slope on a discrete (X,Y) grid, and *without* the use of floating point arithmetic. The complete GEZEL listing illustrates how a slightly more complicated design looks like.



**LISTING 6. The Bresenham line drawing algorithm as an FSMD**

```
// Bresenham line plotter for points in an arbitrary octant
dp bresen(in x1_in, y1_in, x2_in, y2_in : tc(12)) {
   reg x, y             : tc(12);   // current plot position
   reg e                : tc(12);   // accumulated error
   reg eol              : tc(1);    // end-of-loop flag
   reg einc1, einc2     : tc(12);   // increments
   reg xinc1, xinc2     : tc(12);
   reg yinc1, yinc2     : tc(12);
   sig se, sdx, sdy     : tc(12);
   sig asdx, asdy       : tc(12);
   sig stepx, stepy     : tc(12);

   sfg init {
     // evaluate range of pixels and their absolute value
     sdx   = x2_in - x1_in;  asdx = (sdx < 0) ? -sdx : sdx;
     sdy   = y2_in - y1_in;  asdy = (sdy < 0) ? -sdy : sdy;
     // determine direction of x and y increments
     stepx = (sdx < 0) ? -1 : 1;
     stepy = (sdy < 0) ? -1 : 1;
     // initial error
     se    = (asdx > asdy) ? 2 * asdy - asdx : 2 * asdx - asdy;
     // error increment for straight (einc1) and diagonal (einc2) step
     einc1 = (asdx > asdy) ? (asdy - asdx) : (asdx - asdy);
     einc2 = (asdx > asdy) ?  asdy         :  asdx;
     // increment in x direction for straight and diagonal steps
     xinc1 = (asdx > asdy) ? stepx : stepx;
     xinc2 = (asdx > asdy) ? stepx : 0;
     // increment in y direction for straight and diagonal step
     yinc1 = (asdx > asdy) ? stepy : stepy;
     yinc2 = (asdx > asdy) ? 0     : stepy;
     // initialize registers
     x    = x1_in;  y    = y1_in;
     e    = se;
   }

   // end-of-loop test - check if we reach target
   sfg looptest {
     eol   = ((x == x2_in) & (y == y2_in));
   }

   // loop body: adjust x, y and error accumulator
   // use error value to decide straight or diagonal step
43.   sfg loop {
44.     x     = (e >= 0) ? x + xinc1 : x + xinc2;
45.     y     = (e >= 0) ? y + yinc1 : y + yinc2;
46.     e     = (e >= 0) ? e + einc1 : e + einc2;
47.    $display($hex,"Cycle: ",$cycle," Plot point (", x, ",", y, ") ");
48.   }
49. }
50. // controller for bresenham algorithm
51. // initializes, draws one line and then waits in state s3
52. fsm f_bresen(bresen) {
53.   initial s0;
54.   state s1, s2, s3;
55.   @s0 (init)                -> s1;
56.   @s1 (loop, looptest)      -> s2;
57.   @s2 if (eol) then (init)  -> s3;
58.       else (loop, looptest) -> s2;
59.   @s3 (init)                -> s3;
60. }
61.
62. // testbench
63. dp test_bresen(out x1, y1, x2, y2 : tc(12)) {
64.   sig sx : tc(12);
65.   sfg run {
66.     x1 = 5; x2 = 18; y1 = 2; y2 = 8;
67.   }
68. }
69. hardwired h_test_bresen(test_bresen) {run; }
70.
71. system S {
72.   bresen(x1, y1, x2, y2);
73.   test_bresen(x1, y1, x2, y2);
74. }
```

The Bresenham datapath accepts two coordinate tuples, indicating the starting resp. ending points of the vector to be drawn. The bulk of the calculation of the algorithm takes place in an initialization phase, for which a single `sfg` is created (lines 13—34). Basically, the Bresenham algorithm works with three accumulators: one for the x coordinate (register `x`), one for the y coordinate (register `y`), and one error accumulator (register `e`). At runtime, the error accumulator is evaluated to decide on the required increments in the `x` and `y` accumulators.

Not all vectors have the same length, and the Bresenham algorithm only takes a single step (horizontal, vertical or diagonal) per iteration. Because each clock only a single iteration of the Bresenham algortihm is executed, a complete line takes a variable number of clock cycles to generate a vector. Lines 37—39 contain a loop test that decide when to terminate a loop. The actual loop body, which contains the error accumulations, is shown in lines 43—48.

The FSM controller of the Bresenham algorithm is shown in lines 52—60. After initialization, the algorithm takes a first iteration of the loop and evaluates the end-of-loop flag on line 56. From then on, the FSM takes conditional state transitions, which will take it back each time from state `s2` to state `s2` (line 58), or else terminate the loop into state `s3` (line

57). The test `(eol)` checks when the end-of-loop flag becomes true. This test is taken on the value in a register, so it actually checks the end-of-loop condition of the *previous* iteration. For this reason, the instruction of the transition into `s3` is an initialization instruction (line 57). When the output of `eol` is high, the `x` and `y` accumulators are already at there target position, and no more increments should be done.

Finally, lines 63—74 show a simple testbench for the vector generator. The test will evaluate pixels from the vector running from (5,2) to (18,8) (line 66). The output of this simulation with `fdlsim` is shown next. Register values are printed out as tuples. These correspond to *output/input* of a register.

```
> fdlsim bresen.fdl 20
Cycle: 2 Plot point (5/6,2/2)
Cycle: 3 Plot point (6/7,2/3)
Cycle: 4 Plot point (7/8,3/3)
Cycle: 5 Plot point (8/9,3/4)
Cycle: 6 Plot point (9/a,4/4)
Cycle: 7 Plot point (a/b,4/5)
Cycle: 8 Plot point (b/c,5/5)
Cycle: 9 Plot point (c/d,5/6)
Cycle: 10 Plot point (d/e,6/6)
Cycle: 11 Plot point (e/f,6/7)
Cycle: 12 Plot point (f/10,7/7)
Cycle: 13 Plot point (10/11,7/8)
Cycle: 14 Plot point (11/12,8/8)
Cycle: 15 Plot point (12/13,8/8)
```

The algorithm needs 14 cycles to complete the drawing. This corresponds to the largest dimension of the vector, in this case along the X axis.

State transition conditions can also be nested hierarchically. It is possible to write

```
@s0 if (c1) then
      if (c2) then (sfg1) -> s0;
              else (sfg2) -> s0;
    else
      if (c3) then (sfg3) -> s0;
             else (sfg4) -> s0;
```

or, equivalently as a chained else-if condition like

```
@s0 if      ( c1 &  c2) then (sfg1) -> s0;
    else if ( c1 & ~c2) then (sfg2)  -> s0;
    else if (~c1 &  c3) then (sfg3) -> s0;
    else if (~c1 & ~c3) then (sfg4) -> s0;
```

## 3.4 Choosing a controller style

An FSMD consist of two coupled state machines, one playing the role of datapath, and one playing the role of controller. The FSMD model introduces *control steps* in a description, and allows the GEZEL description to move from a *structural* description to a *beha-*

*vioral* description. A GEZEL description is called structural if it uses only a single `sfg` for a datapath that is executed at each clock cycle — cfr. the hardwired datapath defined earlier. A behavioral description is one in which there are multiple `sfg` in a datapath, which are executed over multiple clock cycles.

A structural description will always have only a single assignment per state variable (a register), while a behavioral description can have more. Each control step of a behavioral description, a different assignment can be done. A behavioral description avoids writing multiplexers when multiple assignments are done to the same state variable in multiple `sfg`. When the same functionality needs to be migrated from a behavioral to a structural description, these multiplexers need to be introduced by hand (using the ternary operator ‘`a ? b : c`’).

The absence or presence of the control-step concept also has an important implication on the operation-to-resource binding. Indeed, in a structural description, each operation is executed at each clock cycle. Therefore, each operation will require an individual operator. The word *operator* indicates the resource that implements an *operation*. In a behavioral description, several operations can share the same operator provided that these operations are executed in different control steps. The GEZEL code generator creates VHDL code in such a way that this sharing is possible.

Still there are design cases in which structural descriptions are preferable over behavioral ones. In particular, when creating highly constrained implementations such as very fast or very area-sensitive hardware, it can be necessary to control all aspects of the implementation.

Thus, any design can be created in either design style: structural and behavioral. Which of the two description styles is the better one ? The answer to this question depends on the actual design case, and on the designer. Both have their strengths and weaknesses, and ultimately it is the designer who must select the better option. Here a number of statements that illustrate a few design considerations to select a description style.

| | Structural | Behavioral |
|---|---|---|
| The expressions in a datapath description .. | .. include both scheduling as well as data processing. | .. contain only data processing. |
| The expressions in a datapath description ... | .. are harder to reuse with different schedules. | .. are easier to reuse with different schedules. |
| The state assignment of the controller ... | .. is chosen by the designer. | .. is chosen by the logic synthesis tool. |
| This writing style is useful for ... | .. high-throughput or area-sensitive designs that require full designer control. | .. cycle-true descriptions that put as much work as possible on the logic synthesis tools. |

## 3.5 A Galois Field multiplier

The look and feel of a structural vs behavioral description style is illustrated by implementing a 4-bit, bit-serial Galois-Field Multiplier in each of the description styles.

A Galois Field Multiplier multiplies elements of the field $GF(2^4)$. This finite field consists of 16 elements and is created out of the 2-element field GF(2). The representation of the elements is done using four bits, in terms of a *field basis*. The field basis that will be used is the polynomial basis, in which the individual bits represent coefficients of a polynomial. In this case, the four bits $a_0a_1a_2a_3$ are assumed to be coefficients of a polynomial g(t):

$$g(t) = a_3t_3 + a_2t_2 + a_1t + a_0 \qquad \text{(EQ 3)}$$

The multiplication of two elements out of the field $GF(2^4)$ is defined by the multiplication of two polynomials a(t) and b(t), modulo the irreducible field polynomial d(t). This is a polynomial of degree 4. The simplest irreducible field polynomial for $GF(2^4)$ is

$$d(t) = t^4 + t + 1 \qquad \text{(EQ 4)}$$

As an example, consider the multiplication of a = (1001) with b = (0110). In polynomial format this becomes

$$c(t) = [a(t).b(t)] \bmod d(t) \qquad \text{(EQ 5)}$$

$$c(t) = [(t^3 + 1).(t^2 + 1)] \bmod (t^4 + t + 1) \qquad \text{(EQ 6)}$$

$$c(t) = [t^5 + t^4 + t^2 + 1] \bmod (t^4 + t + 1) \qquad \text{(EQ 7)}$$

The coefficients of this multiplication are elements of the field GF(2), and they are evaluated with modulo-2 arithmetic. Thus, the multiplication result can be simplified to

$$c(t) = [(t + 1)( t^4 + t + 1) + (t + 1)] \bmod (t^4 + t + 1) = (t + 1) \qquad \text{(EQ 8)}$$

The multiplication result corresponds to the bitstring c = (0011).

The next two listings implement this algorithm in a bit-serial fashion. That is, the multiplications of the b operand execute bit-by-bit, and accumulate into the a operand. When the partial results exceeds 4 bits, the resulting polynomial is reduced modulo $(t^4 + t + 1)$. This is done by modulo-2 addition of this polynomial to the partial result.

**LISTING 7. A Galois Field multiplier in behavioral-style description**

```
dp D( in fp, i1, i2 : ns(4); out mul: ns(4);
      in mul_st: ns(1);
      out mul_done : ns(1)) {
  reg acc, sr2, fpr, r1 : ns(4);
  reg mul_st_cmd : ns(1);
  sfg ini { // initialization
```

```
    fpr        = fp;
    r1         = i1;
    sr2        = i2;
    acc        = 0;
    mul_st_cmd = mul_st;
  }
  sfg calc { // calculation
    sr2 = (sr2 << 1);
    acc = (acc << 1) ^ (r1 & (tc(1))  sr2[3])  // add a if b='1'
          ^ (fpr & (tc(1)) acc[3]); // reduction if carry
  }
  sfg omul { // output inactive
    mul      = acc;
    mul_done = 1;
    $display("done. mul=", mul);
  }
  sfg noout { // output active
    mul      = 0;
    mul_done = 0;
  }
}
fsm F(D) {
  state s1, s2, s3, s4, s5;
  initial   s0;
  @s0 (ini,  noout) -> s1;
  @s1 if (mul_st_cmd) then (calc, noout) -> s2;
                      else (ini, noout)  -> s1;
  @s2 (calc, noout) -> s3;
  @s3 (calc, noout) -> s4;
  @s4 (calc, noout) -> s5;
  @s5 (ini,  omul ) -> s1;
}
```

**LISTING 8. A Galois Field multiplier in structural-style description**

```
dp D( in fp, i1, i2 : ns(4); out mul: ns(4);
      in mul_st: ns(1);
      out mul_done : ns(1)) {
  reg ctl : ns(5);
  reg acc, sr2, fpr, r1 : ns(4);

  sfg s1 {
    ctl = mul_st ? 1 : (ctl << 1); // one-hot control
    fpr = fp;
    r1  = i1;
    sr2 = ((ctl == 0) ? i2 : (sr2 << 1));
    acc = (ctl == 0)  ?  0 : (acc << 1)
                           ^ (r1 & (tc(1))  sr2[3])
                           ^ (fpr & (tc(1)) acc[3]);
    mul = acc;
    mul_done = ctl[4];
    $display("mul ", mul, " mul_done ", mul_done);
  }
}
```

```
hardwired H_D(D) {s1; }
```

Both descriptions behave exactly the same, yet they are modeled differently. Listing 7 is a behavioral description and uses an `fsm` to model control of the datapath. Listing 8 shows a hardwired datapath. Listing 8 introduces an extra variable over Listing 7, namely the register `ctl`. This register implements a one-hot controller. At the start of a control cycle, a `‘1’` is injected in this shift register. When it reaches the end, the algorithm is completed. The operations on registers (like `acc` and `sr2`) use the value of the `ctl` register to multiplex two expressions in one assignment. One can verify that in Listing 7, these assignments are located in different `sfg` (`ini` and `calc`). They are integrated by the control steps executed in the control FSM description

# 4.0 Simulating standalone GEZEL designs

This chapter covers GEZEL simulations. Besides the use of the simulation tool, the use of simulation directives is discussed as well as the use of the debug flag.

## 4.1 The simulation algorithm

GEZEL uses a cycle-true simulation algorithm, with an *evaluate* phase and a *register-update* phase. For each simulated clock cycle, the GEZEL kernel takes the following actions in sequence.

1. In each of the controllers in the system (hardwired, sequencer, FSM), select the control step to execute. Selection of the control step also chooses which `sfg` should be executed, and as a result, which expressions should be executed in the evaluate phase of this clock cycle.
2. For each datapath module, evaluate the outputs and the inputs of the registers contained in that datapath. The evaluation process makes use of all expressions which are enabled according to the active control step. Also, the evaluation process obeys data precedences between signals and can trigger evaluation of dependent expressions if needed.
3. Evaluate all *ipblock* (library blocks). The concept of library blocks is treated in Section 8.0, “GEZEL Library Blocks,” on page 75.
4. Evalute all `$display`, `$trace` and `$finish` directives that appear inside of an `sfg` that is currently being executed. Directives are discussed in Section 4.3 on page 34.
5. Update all registers in the simulation by copying the next-value to the current-value.

This simulation algorithm shows the sequence of activities while the GEZEL simulator is in *awake* mode. As this name suggests, there is an alternate mode called *sleep* mode. At runtime, the GEZEL simulator switches automatically between these two modes, based on the activities in your design. In sleep mode, none of the steps 1 to 5 discussed above are executed; the simulator is effectively inactive. Sleep mode is triggered by the occurence of three runtime conditions:

- During clock cycle N, none of the datapath registers has changed state.
- During clock cycle N, none of the controllers has changed state.
- During clock cycle N, none of the library blocks has indicated a change-of-state.

When all of these conditions are simultaneously true, it is easy to see that the simulation result of clock cycle N+1 cannot produce a result that is different from clock cycle N. Consequently, the GEZEL simulator enters sleep mode. If this happens in standalone simulation mode, then the simulation might as well terminate because the simulator cannot leave sleep-mode. However, the-sleep mode will be useful in cosimulations, in which a processor (instruction set simulator) can wakeup the GEZEL simulation dynamically.

## 4.2 The fdlsim tool

The standalone simulator for GEZEL is call `fdlsim`. The command line for `fdlsim` is as follows:

```
fdlsim [-d] [<design.fdl>] <cyclecount>
```

Parameters in between square brackets are optional. When the design filename is not provided, `fdlsim` will read the design from standard input until an end-of-file is encountered. The `-d` is a debug flag. When it is enabled, the simulator provides a more detailed account of the activities during simulation.

When the command line executes, the GEZEL kernel will first parse the design. If any parsing errors are encountered, the simulation will be aborted. If the design is parsed successfully, the simulation will run. It will terminate on one of the following conditions (a) the target cycle count is reached (b) a runtime error occurs or (c) the `$finish` directive is executed.

There are three alternative methods by which the GEZEL simulator can parse and simulate GEZEL designs. Let us consider the following code, which simply counts clock cycles and prints them on the screen:

**LISTING 9. A cycle count printing program**

```
dp mydp {
  sfg go {$display(“Cycle “, $cycle);}
}
hardwired hmydp(mydp) {run;}

system S {
  mydp;
}
```

A first method of simulation is to use the command line. The program can be run as follows to execute 3 clock cycles:

```
> fdlsim listing9.fdl 3
Cycle 1
Cycle 2
Cycle 3
```

A second method is use standard input, to The first is to use the command line, as shown above. The second is to use standard input, and pipe the design into the simulator:

```
> cat listing9.fdl | fdlsim 3
Cycle 1
Cycle 2
Cycle 3
```

The third method is to make use of the scripting feature of the shell. In that case, the location of `fdlsim` must be provided in the code. Assume `fdlsim` is located in `/home/guest/bin/fdlsim`, then the scripting feature (`#!`) is used as:

**LISTING 10. A cycle count printing program, as a script**

```
1. #!/home/guest/bin/fdlsim
2.
3. dp mydp {
4.   sfg go {$display("Cycle ", $cycle);}
5. }
6. hardwired hmydp(mydp) {run;}
7.
8. system S {
9.   mydp;
10. }
```

The GEZEL parser will treat line 1 (starting with `#!`) as a comment. To run this GEZEL program directly from the command line, first turn on the executable flag of `listing10.fdl`:

```
> chmod +x listing10.fdl
```

After that we can run 3 cycles as

```
> listing10.fdl 3
Cycle 1
Cycle 2
Cycle 3
```

A line that starts with '#' is considered as a comment line in GEZEL. This way, it is possible to use the C preprocessor on your GEZEL code before simulation. The C preprocessor enables the use of macro's and include files. To run the simulator together with the C preprocessor, use the command line:

```
> cpp -P listing10.fdl | fdlsim
```

## 4.3 Simulation directives

The simulation output can be modified with the use of simulation directives. These directives are embedded in the GEZEL datapath and controller descriptions.

*Display directives* print out variable values and messages during simulation. A display directive is embedded inside of an `sfg`, and will be executed when the `sfg` executes. The format of the display directive is

```
$display(arg, arg, arg, ...);
```

The arguments of a display directive can be expressions or strings. For example, if the variables `a` and `b` are defined and available in the datapath, we can write

```
    $display(“The value of a + b is “, a + b);
```

The default printing format of `a` and `b` is hexadecimal. This format can be changed using a *modifier directive*. Values can be printed in hexadecimal (`$hex`), decimal (`$dec`) or binary (`$bin`). After a modifier directive is used, it stays in effect until a new one is applied. Thus, for the following three values, the first two will be hex, while the second two will be in binary:

```
    $display(a, b, $bin, a+b, a-b);
```

Apart from strings and expressions, also a number of *meta-variables* are available. Such variables cannot interact with registers or signals, but they can be printed to reveal runtime-dependend information. For this purpose, meta-variables are formatted as directives. `$cycle` returns the current cycle count. `$dp` returns the name of the datapath in which the display directive is used. And `$sfg` returns the name of the `sfg` in which the display directive is used.

*Control directives* affect the course of the simulation. There is one control directive: `$finish`. It can be used inside of an `sfg` definition, and will terminate the simulation when this `sfg` is executed.

*Trace directives* are used record stimuli files. Such a directive is used to create test vector files for future simulations.

The format for a trace directive is

```
    $trace(expression, “filename.txt”);
```

This directive must be placed just after the signal and register definitions inside of a datapath. The default output format for a trace directive is binary. There can be multiple trace directives active at the same time. In that case, each of them should write to a different file. We will illustrate how to use the generated files in Section 5.0, “Converting GEZEL designs to VHDL,” on page 41.

Another format of the trace directive is to use it in the state transition definition of an FSM. In this case, a message will be printed to standard output when the state transition is executed. An example of a trace directive in a state transition is

```
    @s0 (sfg1, sfg2, $trace) -> s1;
```

A summary of all the directives is as follows.

| | |
|---|---|
| `$display(arg, ..)` | Used inside of an `sfg`.<br>Prints strings, expressions and meta-variables. |
| `$cycle` | Used as a `$display` argument.<br>Returns current clock cycle (first cycle = 1). |
| `$sfg` | Used as a `$display` argument.<br>Returns the name of the current sfg. |

| | |
|---|---|
| `$dp` | Used as a `$display` argument.<br>Returns the name of the current datapath. |
| `$finish` | Used inside of an `sfg`.<br>Aborts the simulation. |
| `$trace(expres-sion, “file.txt”);` | Used after register/signal definitions in `dp`.<br>Records value of expression each clock cycle in `file.txt` |
| `$trace` | Used as an instruction in an FSM state transition.<br>Echoes the state transision to the screen. |
| `$option “string”` | Includes optional simulator profiling. string is one of `debug, profile_toggle_upedge_cycles, profile_toggle_alledge_operations, profile_toggle_upedge_operations, profile_toggle_alledge_cycles`<br>(See below for details) |

## 4.4 The debug flag

The main use of the GEZEL standalone simulator is validation of your GEZEL designs. In fact, before taking any GEZEL code into a cosimulation (as will be discussed later), it is a good idea to verify the design first in a small standalone simulation.

`fdlsim` provides a debug mode-of-operation that can be enabled using the `-d` flag on the command line. The effect of the `-d` flag is twofold:

- It will print out the GEZEL symbol table in a file called `fdl.symbols`.
- For each clock cycle, it will print out all register changes in the datapaths and controllers using a value-change format. This means that, if a register is not changing in a particular clock cycle, it will not be printed.

The use of the debug flag will be illustraed on the Galois Field Multiplier from Section 3.5 on page 29. A small testbench was added after line 50 to multiply (1101) with (1001). Also, various directives were added in the simulation, such as on lines 7 (trace), 20 and 25 (display), 26 (finish) and 42 (trace).

**LISTING 11. A Galois Field multiplier testbench**

```
1. dp gfmul( in fp, i1, i2 : ns(4); out mul: ns(4);
2.           in mul_st: ns(1);
3.           out mul_done : ns(1)) {
4.   reg acc, sr2, fpr, r1 : ns(4);
5.   reg mul_st_cmd : ns(1);
6.
7.   $trace(acc, "acc.txt");
8.
9.   sfg ini { // initialization
```

```
    fpr        = fp;
    r1         = i1;
    sr2        = i2;
    acc        = 0;
    mul_st_cmd = mul_st;
  }
  sfg calc { // calculation
    sr2 = (sr2 << 1);
    acc = (acc << 1) ^
          (r1 & (tc(1))  sr2[3]) ^ (fpr & (tc(1)) acc[3]);
    $display("acc ", $bin, acc);
  }
  sfg omul { // output inactive
    mul      = acc;
    mul_done = 1;
    $display("done. mul=", mul);
    $finish;
  }
  sfg noout { // output active
    mul      = 0;
    mul_done = 0;
  }
}
fsm gfmul_ctl(gfmul) {
  state   s1, s2, s3, s4, s5;
  initial s0;
  @s0 (ini,  noout) -> s1;
  @s1 if (mul_st_cmd) then (calc, noout) -> s2;
                      else (ini, noout)  -> s1;
  @s2 (calc, noout) -> s3;
  @s3 (calc, noout) -> s4;
  @s4 (calc, noout) -> s5;
  @s5 (ini,  $trace, omul ) -> s1;
}

dp TB( out fp, i1, i2 : ns(4); out mul_st: ns(1)) {
  reg ctl : ns(4);
  sfg s1 {
    ctl = ctl + 1;
    fp  = 0b0011;
    i1  = 0b1101;
    i2  = 0b1001;
    mul_st = (ctl == 0) ? 1 : 0;
  }
}
hardwired F2(TB) {s1;}

system S {
  gfmul(fp, i1, i2, mul, mul_st, mul_done);
  TB   (fp, i1, i2, mul_st);
}
```

The output of the simulation is:

```
> fdlsim listing11.fdl 10
acc 0000/1101
acc 1101/1001
acc 1001/0001
acc 0001/1111
gfmul_ctl: gfmul_ctl.s5 -> gfmul_ctl.s1
done. mul=f
finish reached !
```

The output of the first four lines was generated by the display directive in line 20. The next line originates from a `$trace` (line 42) telling that the controller gfmul_ctl makes a transition from state `s5` to state `s1`. The result is displayed with another `$display` and finally the simulation is terminated as a result of the `$finish` directive. During the simulation, a tracefile is created for the acc register in acc.txt. The content of this file shows that the simulation ran 6 clock cycles. The file of acc is stored, as binary ASCII numbers. Trace files are useful for use in RT-level VHDL simulations, as will be discussed in Section 5.3 on page 51.

```
> cat acc.txt
0000
0000
1101
1001
0001
1111
```

Now run the simulation again, commenting out all `$display` and `$trace` directives, but enabling the debug mode. The simulation can be monitored clock cycle by clock cycle. Lines indicating register changes include the previous register value, the new register value, and the register name including a pathname that identifies the datapath where the register is located. For example, we can see that in clock cycle 3, the `acc` register in datapath `gfmul` changes value from 0xd to 0x9.  Or, in cycle 5, the FSM of `gfmul_ctl` moves from state `s4` to state `s5`.

```
> fdlsim -d listing11.fdl 10
> Cycle 1
gfmul_ctl: gfmul_ctl.s0 -> gfmul_ctl.s1
                   0                   9 gfmul.sr2
                   0                   3 gfmul.fpr
                   0                   d gfmul.r1
                   0                   1 gfmul.mul_st_cmd
                   0                   1 TB.ctl
> Cycle 2
gfmul_ctl: gfmul_ctl.s1 -> gfmul_ctl.s2
                   0                   d gfmul.acc
                   9                   2 gfmul.sr2
                   1                   2 TB.ctl
> Cycle 3
gfmul_ctl: gfmul_ctl.s2 -> gfmul_ctl.s3
                   d                   9 gfmul.acc
                   2                   4 gfmul.sr2
                   2                   3 TB.ctl
```

```
> Cycle 4
gfmul_ctl: gfmul_ctl.s3 -> gfmul_ctl.s4
                   9                   1 gfmul.acc
                   4                   8 gfmul.sr2
                   3                   4 TB.ctl
> Cycle 5
gfmul_ctl: gfmul_ctl.s4 -> gfmul_ctl.s5
                   1                   f gfmul.acc
                   8                   0 gfmul.sr2
                   4                   5 TB.ctl
> Cycle 6
gfmul_ctl: gfmul_ctl.s5 -> gfmul_ctl.s1
finish reached !
```

## 4.5 Operation profiling and toggle counting

GEZEL provides a simple facility for operation profiling and toggle counting. Operation profiling returns the number of times each operation kind has executed over the course of a simulation. This is useful to estimate the computational complexity of a design. Toggle counting returns an estimate on the number of signal transitions that occur per clock cycle in a particular simulation. This is useful to estimate the immediate dynamic power consumption of a design.

Operation profiling and toggle counting are enabled with the `$option` directive. This directive is given at the top of a GEZEL file, before all datapath definitions. Consider Listing 5 again and insert the directive for operation profiling at line 1:

```
$option “profile_toggle_alledge_operations“
```

This directive produces the following output for 10 clock cycles of simulation:

```
> fdlsim listing05.fdl 10
FDL Cycles 10
           Type          Evals        Toggles
        dpinput             10             16
       dpoutput             20             26
        com_reg             18             34
      assign_op             38             50
         shl_op              2              3
        add_op              17             31
```

The output shows the number of evaluations and the number of net togglecounts per operation type. The `toggle_alledge` part of the option directive enables toggle counting of `0->1` as well as `1->0` transitions. For example, the output shows that 17 additions have been performed over 10 clock cycles, and that these additions result in 31 signal transitions. These signal transitions are measured as hamming distances at the output of the operations in subsequent clock cycles. For example, when in two consecutive clock cycles the pattern `0010` and `0100` would appear, then that would contribute 2 transitions.

It is also possible to obtain the number of signal transitions per clock cycle, by using the directive

```
$option “profile_cycles_alledge_operations“
```

In this case, the output becomes accumulates the operations and toggle counts over each clock cycle.

```
> fdlsim listing05.fdl 10
fdlsim listing05.fdl 10
Profile Cycle      1:      10 evals,            3 toggles
Profile Cycle      2:      11 evals,           12 toggles
Profile Cycle      3:      11 evals,           12 toggles
Profile Cycle      4:      10 evals,           20 toggles
Profile Cycle      5:      10 evals,           18 toggles
Profile Cycle      6:      11 evals,           16 toggles
Profile Cycle      7:      11 evals,           15 toggles
Profile Cycle      8:      10 evals,           21 toggles
Profile Cycle      9:      10 evals,           23 toggles
Profile Cycle     10:     11 evals,          20 toggles
```

Both profile directives have also `_upedge_` counterparts. These restrict the toggle counting to positive toggles only (`0->1` transitions).

# 5.0 Converting GEZEL designs to VHDL

GEZEL designs can be converted automatically into synthesizable VHDL code. The code generation feature shows one of the benefits of working with the restricted semantics of GEZEL: All GEZEL code that you can write and that simulates correctly can be converted into synthesizable VHDL. Apart from the simulation directives, there are no ‘non-synthsizable’ constructs. The library modules (`ipblock` such as discussed in Section 8.0 on page 75) are converted into *black boxes*.

This section covers the use of the code generator tool, `fdlvhd`, as well as the use of the generated code in external tools taking Modelsim as an example. In addition, GEZEL can also record test vector stimuli for use in VHDL simulation.

## 5.1 The fdlvhd tool

The command line of `fdlvhd` is as follows

```
fdlvhd [-d] [-i] [-s] [-c clock reset] [<filename>]
```

Parameters in between square brackets are optional.

The `-d` flag enables debug mode for the VHDL code generator, which creates a statistics report for the generated code.

The `-i` flag creates an active-low reset for the generated VHDL code. The default is an active-high reset.

The `-s` flag creates a synchronous reset for the generated VHDL code. The default is an asynchronous reset.

The `-c` flag allows to specify the names for the clock and reset nets in the generated VHDL code. When specifying for example `-c myclock myreset` then the generated code will use the name `myclock` for the clock net and `myreset` for the reset net.

`<filename>` is the filename of the GEZEL code. This is an optional parameter. When this parameter is not provided, `fdlvhd` will try to read a GEZEL description from standard input. This can be done with redirection or with shell scripting, as discussed in Section 4.2 on page 33.

In the following, the design and VHDL conversion of a small sorting circuit is discussed. The sorter is known as an *odd-even* sorter, which will sort a list of numbers by comparing alternatively adjacent odd and even tuples of a list of numbers. The odd-even sorter can be represented in a flow diagram as illustrated in Figure 5.6a. The squares represent values from the list, the circles represent comparison operations. Each comparison will take a tuple as input and produce a sorted tuple as output. Worst-case, an element has to traverse all positions in the a before it arrives at the correct position. Thus, a list of N elements can be sorted in (N-1) iterations. The entire odd-even algorithm uses N/2.(N-1) comparisons. While the odd-even sorter is not the most efficient sorter, the algorithm is regular, scalable

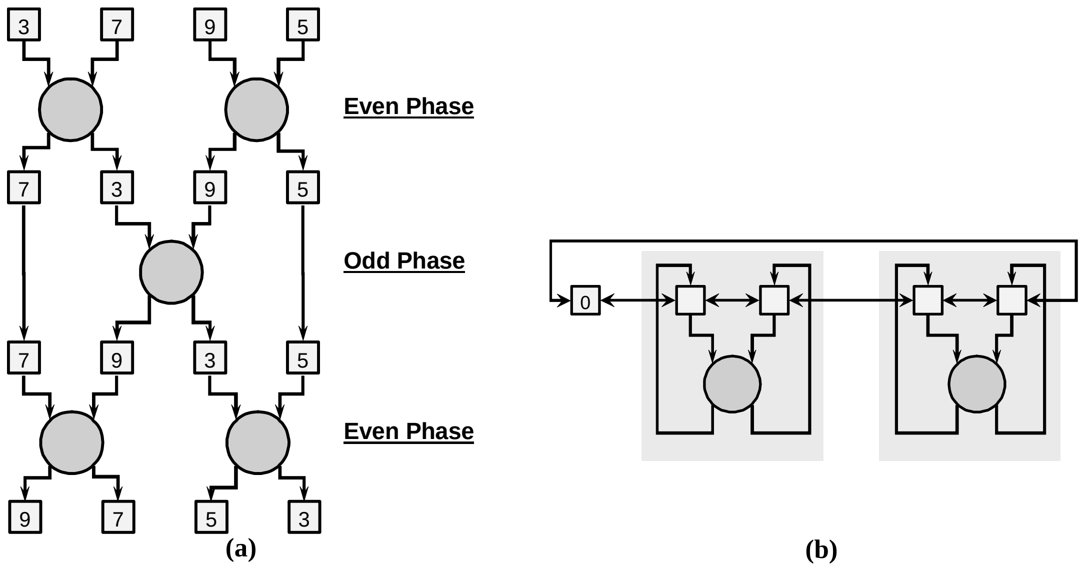


**FIGURE 5.6. Odd-Even Sorting Network (a) flow diagram and (b) architecture**

and performs local comparisons. These are desirable features for a hardware implementation.

One possible architecture for a 4-element odd-even sorter is illustrated in Figure 5.6b. The sorter is implemented using a series of sorter cells, with each cell managing one tuple out of the list. To implement the odd-even exchanges, the sorter cells communicate values with each other. Each sorter cell understands one of four instructions. It can stay idle, it can perform a sort operation, it can communicate list values to the left neighbour, or it can communicate values to the right neighbour.

An 8-element odd-even sorter can be created that executes one iteration (an odd or even phase) per clock cycle, similar to the architecture in Figure 5.6b. The architecture can be implemented using 4 sorter cells, with each sorter cell holding two elements of the list. Structural hierarchy will be used to model the 8-element sorter using 2-element sorter cells. The design will use a testbench to control the resulting sorter, by first loading 8 values (serially), then sorting the list, and next reading out the sorted result. The GEZEL code in Listing 12 shows the 8-element sorter with the testbench.

### LISTING 12. An odd-even sorter program

```
// sorter cell
dp sorter0(in r_in : ns(8); out r_out : ns(8);
           in l_in : ns(8); out l_out : ns(8);
           in ctl  : ns(2)) {
   reg r0, r1 : ns(8);

   sfg run {
     r0 = (ctl == 1) ? ((r0 > r1) ? r0 : r1) :  // sort
          (ctl == 2) ? r_in :                   // to-left
          (ctl == 3) ? r1 :                     // to-right
           r_in;
     r1 = (ctl == 1) ? ((r0 > r1) ? r1 : r0) :
          (ctl == 2) ? r0 :
          (ctl == 3) ? l_in :
          r0;
     l_out = r1;
     r_out = r0;
   }
}
hardwired hsorter0(sorter0) {run; }

dp sorter1 : sorter0
dp sorter2 : sorter0
dp sorter3 : sorter0

dp sorter8(in  d_in  : ns(8);
           out d_out : ns(8);
           in  ctl   : ns(2)) {
   reg r8  : ns(8);
   sig m0_i, m1_i, m2_i, m3_i, m4_i : ns(8);
   sig m0_o, m1_o, m2_o, m3_o, m4_o : ns(8);

   use sorter0 (m0_i, m0_o, m1_o, m1_i, ctl);
   use sorter1 (m1_i, m1_o, m2_o, m2_i, ctl);
   use sorter2 (m2_i, m2_o, m3_o, m3_i, ctl);
   use sorter3 (m3_i, m3_o, m4_o, m4_i, ctl);

   sfg run {
     m4_o  = r8;
     d_out = m4_i;
     r8    = (ctl == 2) ? m4_i :
             (ctl == 3) ? m0_o :
             r8;
     m0_i  = (ctl == 2) ? r8   :
             d_in;
   }
}
hardwired hsorter8(sorter8) {run;}

dp tstsorter(out d : ns(8);
             in  r : ns(8);
             out c : ns(2)) {
   lookup T : ns(8) = {3, 6, 10, 2, 28, 5, 16, 9};
   reg  adr : ns(3);
   reg  sr  : ns(3);

   sfg init {
     adr = 0; sr = 0; d = 0; c = 0;
   }
   sfg load {
     $display("C", $cycle, ": load ", $dec, d);
     d   = T(adr);
     adr = adr + 1;
     c   = 0;
   }
```

```
66.    sfg sort  { d = 0; c = 1; }
67.    sfg left  { d = 0; c = 2; }
68.    sfg right { d = 0; c = 3; sr = sr + 1;}
69.    sfg read {
70.      d = 0; c = 0;
71.      adr = adr + 1;
72.      $display("C", $cycle, ": out[", adr, "] = ", $dec, r);
73.    }
74. }
75.
76. fsm htstsorter(tstsorter) {
77.   initial s0;
78.   state s1, s2, s3, s4, s5, s6, s7, s8;
79.   @s0  (init) -> s1;
80.   @s1  if (adr == 7) then (load) -> s2;
81.                      else (load) -> s1;
82.   @s2  (sort)  -> s3;
83.   @s3  (right) -> s4;
84.   @s4  (sort)  -> s5;
85.   @s5  (left)  -> s6;
86.   @s6 if (sr == 3)  then (read) -> s7;
87.                     else (sort) -> s3;
88.   @s7 if (adr == 7) then (read) -> s8;
89.                     else (read) -> s7;
90.   @s8 (init) -> s1;
91. }
92.
93. system S {
94.  tstsorter(d, r, c);
95.   sorter8 (d, r, c);
96. }
```

Lines 1—20 show the 2-element sorter cell. This cell is duplicated three times in lines 22—24. The 8-element sorter is illustrated in lines 26—48 and includes the four 2-element sorters with a `use` statement. The testbench uses a lookup table and is modeled as an FSMD, as shown in lines 76 —91.

Running the simulation results in the following output.

```
> fdlsim listing12.fdl 30
C1: load 3
C2: load 6
C3: load 10
C4: load 2
C5: load 28
C6: load 5
C7: load 16
C8: load 9
C21: out[0/1] = 2
C22: out[1/2] = 3
C23: out[2/3] = 5
C24: out[3/4] = 6
C25: out[4/5] = 9
C26: out[5/6] = 10
```

```
C27: out[6/7] = 16
C28: out[7/0] = 28
Activity(%) on 11 registers: 50.303 (166/330)
FDL Cycles 30, sleeping cycles 0(0%)
```

The sorting process takes 12 clock cycles (from 9 to 20), and corresponds to 3 iterations through the odd-even sort algorithm.

Next, VHDL code is created for this description using the following command line.

```
> fdlvhd listing12.fdl
Pre-processing System ...
Output VHDL source ...
----------------------------
Generate file: sorter0.vhd
Generate file: sorter8.vhd
Generate file: tstsorter.vhd
Generate file: system.vhd
```

For each datapath module in the system, a separate VHDL file is created. Some features of the VHDL code generator, including the structure and contents of the generated files, are as follows. Only partial listings will be shown, an ellipsis (...) is used were code was skipped.

- The generated VHDL uses word-level semantics, similar to the GEZEL code. For this purpose, it makes use of the standard IEEE libraries for arithmetic. It also makes use of one extra library, `std_logic_arithext`, which contains a few support functions required for the generated code.

```
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
library work;
use work.std_logic_arithext.all;
```

- For each datapath module, a separate entity is created. The ports on this entity are the same as for the GEZEL module, but also include a clock pin and a reset pin. The sorter cell `sorter0` discussed earlier, for example, looks as follows.

```
entity sorter0 is
   port(
      r_in:in std_logic_vector(7 downto 0);
      r_out:out std_logic_vector(7 downto 0);
      l_in:in std_logic_vector(7 downto 0);
      l_out:out std_logic_vector(7 downto 0);
      ctl:in std_logic_vector(1 downto 0);
      RST : in std_logic;
      CLK : in std_logic
   );
end sorter0;
```

  The generated code uses the same net names as in the GEZEL code. No check is done to verify these names are valid VHDL identifiers, it is up to the user to make sure that no conflicts can arise because of this.

- A hardwired datapath, such as `sorter0` is modeled using two VHDL processes, one to evaluate combinational logic, and one to evaluate sequential logic. The sequential logic can use either low- or high-asserted reset signals, with synchronous or asynchronous reset. The reset style can be chosen in the `fdlvhd` command line. The initial values of the registers are zero, the same as in the GEZEL simulation. For each register, two VHDL signals are created. If `r0` is a GEZEL register, then the VHDL signal `r0` indicates the output of the register, and the VHDL signal `r0_signal` indicates the input of the register. Any other intermediate signals that are created by the VHDL code generator are called `sig_xx` with `xx` a decimal number.

```
architecture RTL of sorter0 is
signal r0:std_logic_vector(7 downto 0);
signal r0_wire:std_logic_vector(7 downto 0);
signal r1:std_logic_vector(7 downto 0);
signal r1_wire:std_logic_vector(7 downto 0);
signal sig_0:std_logic;
signal sig_1:std_logic;

...

begin
   --register updates
   dpREG: process (CLK,RST)
      begin
         if (RST = '1') then
            r0 <= (others=>'0');
            r1 <= (others=>'0');
         elsif CLK' event and CLK = '1' then
            r0 <= r0_wire;
            r1 <= r1_wire;
         end if;
      end process dpREG;

   --combinational logics
   dpCMB: process (...)
      begin

      ...

      end process dpCMB;
end RTL;
```

- The generated VHDL code reflects structural hierarchy in the same way as it appears in the GEZEL code. The `sorter8` module includes 4 submodules and will instantiate the same 4 submodules as components in the VHDL architecture.

```
entity sorter8 is
   port(
      d_in:in std_logic_vector(7 downto 0);
      d_out:out std_logic_vector(7 downto 0);
      ctl:in std_logic_vector(1 downto 0);
      RST : in std_logic;
      CLK : in std_logic
   );
end sorter8;

architecture RTL of sorter8 is

...

component sorter0
   port(
      r_in:in std_logic_vector(7 downto 0);
      r_out:out std_logic_vector(7 downto 0);
      l_in:in std_logic_vector(7 downto 0);
      l_out:out std_logic_vector(7 downto 0);
      ctl:in std_logic_vector(1 downto 0);
      RST : in std_logic;
      CLK : in std_logic
   );
end component;

begin
   --portmap
   label_sorter0 : sorter0 port map ( ... );
   label_sorter1 : sorter0 port map ( ... );
   label_sorter2 : sorter0 port map ( ... );
   label_sorter3 : sorter0 port map ( ... );
   dpREG: process (CLK,RST)
      begin
         if (RST = '1') then
            r8 <= (others=>'0');
         elsif CLK' event and CLK = '1' then
            r8 <= r8_wire;
         end if;
      end process dpREG;

   dpCMB: process (...)
      begin

         ...

      end process dpCMB;
end RTL;
```

- An FSMD module, such as `htstsorter` in Listing 12, is translated to four VHDL processes: a combinational and sequential process for the datapath, and a combinational and sequential process for the controller. In combinational processes, choice is modeled using `case` statements. For the datapath, the entries of this `case` correspond to the different instructions that are possible. For the FSM controller, the entries of the `case` correspond to the different states the FSM can assume.

  State assignment for the controller is symbolic. The generated code uses the same symbolic state names as the GEZEL code. The code generator will also create a symbolic instruction set that the FSM will use to control the datapath. This symbolic instruction set is determined by the combinations of `sfg` that must be executed. For example, suppose that a datapath in GEZEL contains three `sfg` called `sfg0`, `sfg1`, `sfg2`. Considering all possible state transitions in the FSM, the GEZEL code generator finds that either `(sfg0)` executes or else both `(sfg1, sfg2)`. The resulting datapath in VHDL will decode two symbolic instructions, the first called `sfg0` and the second called `sfg1_sfg2`.

The following snippet illustrates the VHDL layout of the testbench `tstsorter` in Listing 12.

```
entity tstsorter is
   port(
      d:out std_logic_vector(7 downto 0);
      r:in std_logic_vector(7 downto 0);
      c:out std_logic_vector(1 downto 0);
      RST : in std_logic;
      CLK : in std_logic
   );
end tstsorter;

architecture RTL of tstsorter is
...
begin
   dpREG: process (CLK,RST)
      ...
      end process dpREG;
   dpCMB: process (...)
         ...
         case cmd is
            when init => ...
            when load => ...
            when sort => ...
            when right => ...
            when left => ...
            when read => ...
            when others=>
         end case;
      end process dpCMB;
   fsmREG: process (CLK,RST)
      ...
      end process fsmREG;
   fsmCMB: process (...)
      begin
      ...
      case STATE is
         when s0 =>
                cmd <= init;
         when s1 =>
            if(sig_4 = '1') then
                   cmd <= load;
            else
                   cmd <= load;
            end if;
         when s2 =>
                cmd <= sort;
         when s3 =>
                cmd <= right;
         when s4 =>
```

```
                    cmd <= sort;
            when s5 =>
                    cmd <= left;
            when s6 =>
               if(sig_5 = '1') then
                       cmd <= read;
               else
                       cmd <= sort;
               end if;
            when s7 =>
               if(sig_6 = '1') then
                       cmd <= read;
               else
                       cmd <= read;
               end if;
            when s8 =>
                    cmd <= init;
            when others=>
            end case;
      end process fsmCMB;
   end RTL;
```

- The system module interconnects datapath modules, and also creates a default testbench that drives the reset and clock signal. If a GEZEL file simulates as a standalone module, then it can also simulate as a standalone VHDL file with the system module as top entity.

```
--system entity
entity system is
end system;
architecture RTL of system is
signal sig_0:std_logic_vector(7 downto 0);
signal sig_1:std_logic_vector(7 downto 0);
signal sig_2:std_logic_vector(1 downto 0);
signal RST : std_logic;
signal CLK : std_logic;

--component declaration
component sorter8
   port(
      d_in:in std_logic_vector(7 downto 0);
      d_out:out std_logic_vector(7 downto 0);
      ctl:in std_logic_vector(1 downto 0);
      RST : in std_logic;
      CLK : in std_logic
   );
end component;
component tstsorter
   port(
      d:out std_logic_vector(7 downto 0);
      r:in std_logic_vector(7 downto 0);
      c:out std_logic_vector(1 downto 0);
      RST : in std_logic;
      CLK : in std_logic
   );
end component;

begin
   --portmap
   label_sorter8 : sorter8 port map (
         d_in => sig_0,
         d_out => sig_1,
         ctl => sig_2,
         RST => RST,
         CLK => CLK
      );
   label_tstsorter : tstsorter port map (
         d => sig_0,
         r => sig_1,
         c => sig_2,
         RST => RST,
         CLK => CLK
      );

   --clk, reset generation
   process
      begin
      RST       <= '1';
      wait for 50 ns;
      RST       <= '0';
      wait;
   end process;

   process(CLK   )
      begin
      if CLK'event and CLK      = '1' then
         CLK          <= '0' after 25 ns;
      else
         CLK          <= '1' after 25 ns;
      end if;
   end process;

end RTL;
```

## 5.2 VHDL Simulation with Modelsim

The VHDL code that is generated by GEZEL can be taken further to synthesis or simulation tools. One example is shown here: the use of the VHDL in conjunction with an the Modelsim VHDL simulator by ModelTech, Inc. The steps to follow in order to simulate your design in Modelsim are as follows.

1. Make sure your design works correctly in the GEZEL simulator (`fdlsim`) before you start using the code generator.
2. Generate VHDL code for the GEZEL design using `fdlvhd`.

3. Copy the generated files, as well as the support library `std_logic_arithext.vhd` to a separate directory, say `mydirectory`. The support library can be found in the source directory of the GEZEL kernel.
4. Start Modelsim, and create a new project that points to this `mydirectory`. (File - New - Project)
5. Assign all VHDL files, including the support library, to this project. (Project - Add File to Project)
6. Compile the VHDL files while respecting the structural hierarchy. (Design - Compile) This means that you have to compile lower-level entities first. Start with the support library, then the leaf modules, then the toplevel modules. For example, in the sorter circuit, a correct compilation order would be: `std_logic_arithext.vhd, sorter0.vhd, sorter8.vhd, tstsorter.vhd, system.vhd`.
7. Load the design and select the toplevel module, `system`. (Design - Load Design)
8. Assign signals to the wave window and simulate.

## 5.3 Stimuli Directives

When developing a big design it is sometimes required simulate only part of the generated VHDL. For example, when developing a cryptographic coprocessor for an ARM, then we might want to simulate the VHDL for the coprocessor without including a VHDL model of an ARM processor, or even without running a cosimulation at the VHDL level. In such a case, you could create a VHDL testbench that only includes the block you are interested in. You would also need to have a set of testvectors or stimuli for the VHDL simulation of this block.

These stimuli can be created using stimuli directives. Stimuli directives allow you to record all I/O of a GEZEL module during GEZEL simulation, and store these into a file. Later, during VHDL simulation, you can replay these I/O activities on a stand-alone VHDL model of the same module. Thus you can run a system cosimulation of an ARM instruction-set simulator and a GEZEL coprocessor (See Section 6.0 on page 54), and later rerun the same simulation on a VHDL model of the GEZEL coprocessor without including the ARM.

The `$trace` stimuli directive can record the value of any signal in the simulation into a file. Each clock cycle, the value of that signal is recorded and stored into a file. We will give an example of the use of `$trace`, by creating a set of testvectors for the `sorter8` module of the sorter circuit presented in Section 5.1 on page 41.

The first step is to extend the GEZEL description with `$trace` directives that record the I/O signals of `sorter8`.

```
dp sorter8(in  d_in  : ns(8);
           out d_out : ns(8);
           in  ctl   : ns(2)) {
```

```
        $trace(d_in,  “d_in.txt”);
        $trace(d_out, “d_out.txt”);
        $trace(ctl,   “ctl.txt”);

        ...
    }
```

When the simulation executes, three extra files will be generated containing the IO signals of `sorter8` per clock cycles. The first few lines of `d_in.txt` for example are as follows:

```
00000000
00000011
00000110
00001010
00000010
00011100
00000101
00010000
00001001
```

These stimuli files can be used in a VHDL testbench. The example process in Listing 13 uses the text IO functions in VHDL to read the files `d_in.txt` and `ctl.txt`, one line at each clock cycle. Such a process can be used in a customized version of the system testbench, discussed at the end of Section 5.2 on page 50.

Note that, while the stimuli directive helps in collecting test vectors from your GEZEL simulation, the design of the VHDL testbench code that uses these files is not automated (yet).

**LISTING 13. A stimuli file reader as VHDL testbench**

```
-- Here is a process that reads out data from d_in.txt
-- and ctl.txt, and can driver the inputs of the
-- sorter8 module.
-- To use the text IO functions in VHDL
--   use IEEE.std_logic_textio.all;
--   use std.textio.all;

   process
     file f_d_in : text is in "d_in.txt";
     file f_ctl  : text is in "ctl.txt";
     variable l_d_in : line;
     variable l_ctl  : line;
     variable v_d_in : std_logic_vector(7 downto 0);
     variable v_ctl  : std_logic_vector(1 downto 0);
     begin
     wait until RST'event and RST = '1';
     loop
       if (not(endfile(f_d_in))) then
         readline(f_d_in, l_d_in);
         read(l_d_in, v_d_in);
       else
         assert false
           report "End of input on din.txt"
           severity warning;
       end if;
       if (not(endfile(f_ctl))) then
         readline(f_ctl, l_ctl);
         read(l_ctl, v_ctl);
       else
         assert false
           report "End of input on ctl.txt"
           severity warning;
       end if;
       sig_0 <= v_d_in;
       sig_2 <= v_ctl;
       wait until CLK'event and CLK = '1';
     end loop;
    end process;
```

# 6.0 Cosimulating GEZEL with Instruction Set Simulators

GEZEL designs can be cosimulated with instruction-set simulators. Such designs can include coprocessors that implement graphics, networking and/or cryptographic functions. This section presents three cosimulators that make part of the GEZEL release.

- `armcosim` cosimulates a single ARM core with GEZEL.
- `armzilla` cosimulates multiple ARM cores with GEZEL.
- `gezel51` cosimulates an 8051 microcontroller with GEZEL.

The general characteristics of the cosimulation design flow are outlined first, followed by a discussion of each of the two cosimulation environments in more detail.

## 6.1 Cosimulation Interfaces and Interface Protocols

The cosimulation of GEZEL with an instruction-set simulator requires, besides a GEZEL program, also an executable program that can run on the instruction-set simulator. These executables can be created using a compiler. When a compiler runs on a different host machine (e.g. a linux PC) than the target execution environment (e.g. an ARM instruction-set simulator), a *cross-compiler* is required. In this discussion, the C programming language and a C cross-compiler will be used to create the executables.

The interactions between the GEZEL program and the executables running on the instruction-set simulators are captured in a *cosimulation* interface, which is an abstracted version of the real hardware/software interface. The cosimulation interfaces of GEZEL are cycle-true models of the real implementations.

There are various forms of cosimulation interfaces, depending on the I/O mechanisms provided by the core (instruction-set simulator). A commonly used type of interface is a memory-mapped interface, in which a set of addresses in the address space of the core is shared between the hardware and the software running on the core. There can also be specialized coprocessor- or I/O-interfaces, which are supported by dedicated instructions on the core. The main advantage of a memory-mapped interface is that it is almost core-independent. Therefore, C code and GEZEL code written for one type of processor can be ported to another processor with only minimal changes. The main advantage of the specialized interface on the other hand, is that it provides a dedicated, non-shared and usually high-bandwidth data channel between the core and the hardware.

A cycle-true cosimulation interface by itself provides only a mechanism to transfer data between a C program and GEZEL. This data transfer proceeds between two concurrent entities (a core and a hardware block). To avoid that data values get lost when one party is unware of the others' activities, synchronization is required. Such synchronization will be provided in a *synchronization protocol*. A synchronization protocol defines a signalling sequence on one or more control signals, in addition to the data transfer channel between C and GEZEL. This signalling sequence ensures that the communicating parties achieve synchronization. Both the control signals and data transfer channel can be implemented using the same cosimulation interfaces. For example, you can use a memory-mapped interface for both of them. Some examples of synchronization strategies will be provided in the examples of `armcosim`, `armzilla` and `gezel51`.

## 6.2 The armcosim tool

The `armcosim` tool is a cosimulator for a single ARM core with the GEZEL kernel. The command line for `armcosim` is as follows:

```
armcosim [-v] [-f fpe-path] [-h] [-c cyclecount] \
    -g gezel_description                         \
    -a arm_executable [arm_program_parameters]
```

Parameters in between square brackets are optional.

The `-v` is the verbose flag. When it is enabled, the ARM instruction-set simulator provides more detailed feedback on the activities it is performing.

The `-f` flag allows to select the path to an ARM floating-point emulation library. The ARM instruction-set simulator comes with a floating point emulator as a separate binary file (`nwfpe.bin`). The emulator is required when you want to execute a C program that uses floating point operations. During compilation of `armcosim`, the path to this emulator is hardcoded into the simulator. When, after installing `armcosim`, the emulator is no longer available from the default path, the `-f` flag allows to indicate where to look for this file.

The `-h` flag prints a help message.

The `-c` flag allows to indicate an upperbound for the amount of cycles to simulate. By default, the cosimulation will run until the C program has completed execution, i.e. when the `main` function terminates.

The `-g` flag is used to indicate the path to the GEZEL hardware description.

The `-a` flag is used to indicate the path to the ARM executable that must run on the ISS. This executable must be created as a statically linked ARM-ELF executable.

Here is a small example of a hardware-software cosimulation, consisting of a synchronized data transfer. Below is the hardware description in GEZEL.

**LISTING 14. A GEZEL description of hardware-side of hardware/software handshake**

```
// cosimulation interfaces
ipblock b1(in data : ns(8)) {
  iptype "armsimsink";
  ipparm "address=0x80000000";
}
ipblock b2(out data : ns(8)) {
```

```
7.   iptype "armsimsource";
8.   ipparm "address=0x80000004";
9. }
10. ipblock b3(out data : ns(32)) {
11.   iptype "armsimsource";
12.   ipparm "address=0x80000008";
13. }
14.
15. // hardware ‘receiver’
16. dp D2(in req : ns(8); out ack : ns(8); in data : ns(32)) {
17.   reg reqreg  : ns(8);
18.   reg datareg : ns(32);
19.   sfg sendack {
20.    ack = 1;
21.   }
22.   sfg sendidle {
23.    ack = 0;
24.   }
25.   sfg read {
26.    reqreg  = req;
27.    datareg = data;
28.   }
29.   sfg rcv {
30.     $display("data received ", data, " cycle ", $cycle);
31.   }
32. }
33. fsm F2(D2) {
34.   initial s0;
35.   state   s1, s2;
36.   @s0 (read, sendack) -> s1;
37.   @s1 if (reqreg) then (read, rcv, sendidle) -> s2;
38.                   else (read, sendack)       -> s1;
39.   @s2 if (reqreg) then (read, sendidle)      -> s2;
40.                   else (read, sendack)       -> s1;
41. }
42. // connect hardware to cosimulation interfaces
43. system S {
44.   D2(r,a,d);
45.   b1(a);
46.   b2(r);
47.   b3(d);
48. }
```

Lines 1—13 indicate three cosimulation interfaces between GEZEL and the ARM. These interfaces are unidirectional, memory-mapped interfaces and are represented using `ipblock`. An `ipblock` is a library block with similar semantics as a datapath `dp`. Library blocks are discussed in more detail in Section 8.0 on page 75. In this section, their use as cosimulation interface is important. There are two types of memory-mapped interfaces:

- `armsimsink` blocks, such as in lines 2—5. These are channels from GEZEL to the ARM; they use consider the ARM core to be a data sink. These blocks define an *input* port on the library block where data to be send to the ARM is provided.

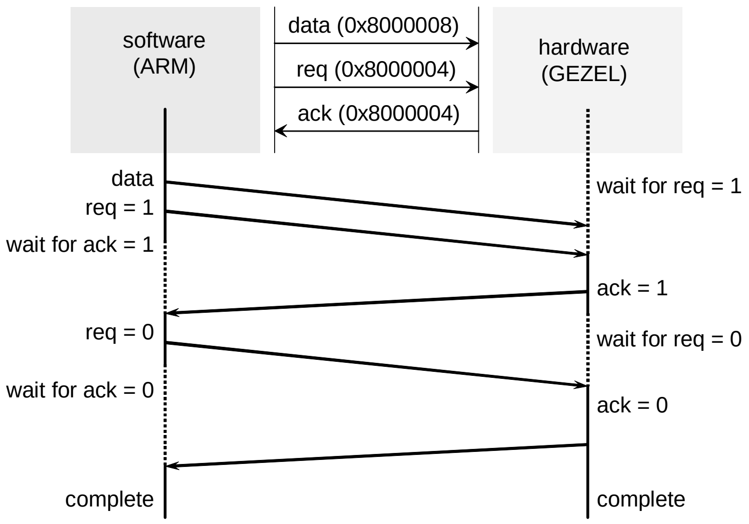


**FIGURE 6.7. Two-phase full handshake protcol between software running on an ARM and hardware described in GEZEL.**

- `armsimsource` blocks, such as in lines 6—9. These are channels from the ARM to GEZEL; they use the ARM core as a source. These blocks define an *output* port on the library block where data that is received from the ARM can be retrieved.

Both `armsimsink` and `armsimsource` define an additional parameter indicated in an `ipparm` field (lines 4, 8, 12). This parameter contains the address value of the ARM memory map that is shared between the GEZEL hardware block and the ARM core.

In lines 15—41, an example hardware module is shown that can accept values from the software running on the ARM. The module executes a two-phase full-handshake protocol, which uses two control lines (an input `req` and an output `ack`). The operations of the protocol are illustrated in Figure 6.7. At the start of the two-phase full-handshake protocol, the hardware module is waiting for the `req` control signal to become high (lines 37—38). Before driving this signal high, the software will first set the data value to a stable value.

At that moment the second phase of the handshake protocol is entered, and an inverse but symmetric handshake sequence is executed. First the software will drive `req` to zero, after which the GEZEL hardware model will respond by driving `ack` to zero (lines 39—40).

A software program that executes this handshake sequence on the ARM is shown next.

**LISTING 15. A C description of software side of hardware/software handshake**

```
int main() {
  volatile unsigned char *reqp, *ackp;
  volatile unsigned int  *datap;
  int data = 0;
  int i;

  reqp  = (volatile unsigned char *) 0x80000004;
  ackp  = (volatile unsigned char *) 0x80000000;
9.   datap = (volatile unsigned int  *) 0x80000008;
10.
11.   for (i=0; i<10; i++) {
12.     *datap = data;
13.     data++;
14.
15.     *reqp = 1;
16.     while (*ackp) { }
17.
18.     *reqp  = 0;
19.     while (! *ackp) { }
20.   }
21.   return 0;
22. }
```

The memory-mapped hardware/software interfaces are included in lines 2—3 as pointers of the `volatile` type. Such pointers are treated with caution by a compiler optimizer. In particular, no assumption is made about the persistence of the memory location that is being pointed at by this pointer. The pointers are initialized in lines 7—9 with values corresponding to the memory addresses used in the GEZEL description.

In lines 11—20, a simple loop is shown that executes the software side of the two-phase full-handshake protocol. Lines 15 and 18 illustrate why the volatile declaration is important. An optimizing C compiler would conclude that `reqp` is simply overwritten in the body of the loop. In addition, the resulting value is loop-invariant and can be hoisted outside of the loop body. The resulting optimized code would write the value 0 once in `reqp` and never change it afterwards. By declaring `reqp` to be a volatile pointer, the compiler will refrain from such optimizations.

Everythin is now ready to run the cosimulation. Start by compiling the ARM program using a cross-compiler. The `-static` flag creates a statically linked executable, a requirement for the ARM ISS.

```
> /usr/local/arm/bin/arm-linux-gcc -static \
           hshakedriver.c -o hshakedriver
```

Next run the cosimulation with `armcosim:`

```
> ../../../build/bin/armcosim -g hshake.fdl -a hshakedriver
Initializing StrongARM processor arm_processor_0
Initializing GEZEL processor hshake.fdl
armsimsink: set address 2147483648
armsimsource: set address 2147483652
armsimsource: set address 2147483656
data received 0 cycle 22887
data received 1 cycle 23040
data received 2 cycle 23079
data received 3 cycle 23118
data received 4 cycle 23157
data received 5 cycle 23196
data received 6 cycle 23235
data received 7 cycle 23274
data received 8 cycle 23313
data received 9 cycle 23352
Total icache reads:       7422
Total icache read misses: 250
icache hit ratio:         96.632%
Total itlb reads:       7447
Total itlb read misses: 17
itlb hit ratio:         99.772%
Total dcache writes:       385
Total dcache write misses: 102
Total dcache reads:        1500
Total dcache read misses:  169
dcache hit ratio:         85.623%
Total dtlb reads:       1885
Total dtlb read misses: 28
dtlb hit ratio:         98.515%
Total biu accesses: 532
biu activity:      67.026%
Total allocated OSMs : 7447
Total retired OSMs   : 7444
Total cycles         : 25808
Equivalent time on 206.4MHz host: 0.0001 sec.
Total memory pages allocated : 370
```

The simulation initializes and then prints a series of ‘data received’ messages, which are generated by the GEZEL program. The round-trip execution time of the protocol takes 39 clock cycles, a rather high value because we are working with an unoptimized C program and an unoptimized handshake protocol. At the end of the simulation `armcosim` prints a series of messages, starting with ‘Total icache reads’. This is output provided by the ARM instruction-set simulator.

## 6.3 The armzilla tool

`armzilla` is a GEZEL+ARM multiprocessor simulator. It is an extended version of the armcosim tool, that allows an arbitrary number of ARM ISS to be used in a simulation. This is useful for developments such as network-on-chip and multiprocessor system-on-chip architectures. The command line for `armzilla` is

```
armzilla [-v] [-f fpe-path] [-d] [-h] <connection file name>
```

The use of the `-v`, `-f` and `-h` flags are the same as for `armcosim`, discussed in Section 6.2 on page 55.

The `-d` flag enables the debug mode of the GEZEL kernel, as discussed in Section 4.4 on page 36.

The connection file name specifies the name of a system topology file. This file defines the number of ARM cores in the simulation, the symbolic names for each executable and the name of the GEZEL file that contains the system hardware description.

An example of a system with two ARM processors follows next, where data is shipped from one ARM to the next using a dedicated communication bus and an optimized two phase single-sided handshake protocol. The example is comparable in functionality to the example of `armcosim` discussed earlier, but uses two ARM processors instead of an ARM processor and a hardware module.

The first part of the design is the system topology file, `system.connect`.

**LISTING 16. An ARMZILLA system topology file for a two-ARM system**

```
// system topology for a sender and a receiver
( core  = "sender",   exec = "sender.exe")
( core  = "receiver", exec = "receiver.exe")
( gezel = "network.fdl" )
```

The system topology file has a number of entities, each one included in round brackets. The entities starting with `core` indicate ARM processors, the entity starting with `gezel` indicates the hardware GEZEL description. Only a single GEZEL file can be used.

A core entitiy specifies a symbolic name for the core, as well as the path and filename of the executable that should be run on that core. The need for symbolic core names originates from the fact that, at the hardware GEZEL level, it must be possible to uniquely distinguish the different cores in the system. We will illustrate this in the GEZEL description shown next.

**LISTING 17. GEZEL interconnect description for a two-ARM system**

```
// network architecture
dp network_link(in  fromsender : ns(32);
                out toreceiver : ns(32)) {
  reg buf : ns(32);
  sfg run {
      buf = fromsender;
      toreceiver = buf;
      $display("Cycle: ", $cycle, " channel = ", toreceiver);
  }
}
hardwired ctl_network_link(network_link) {run; }

// interfaces and system interconnect
ipblock sender_itf(out data : ns(32)) {
  iptype "armzillasource";
  ipparm "processor=sender";
  ipparm "address=0x80000008";
}
ipblock receiver_itf(in data : ns(32)) {
  iptype "armzillasink";
  ipparm "processor=receiver";
  ipparm "address=0x80000008";
}

```

```
25. system S {
26.   sender_itf(S);
27.   network_link(S,R);
28.   receiver_itf(R);
29. }
```

The interconnection network described in Listing 17 is a very simple point-to-point connection with a single register as storage (Lines 1—11).

Two memory-mapped hardware-software interfaces are defined in lines 14—23. Each interface is connected to a different ARM core. The GEZEL library blocks that are used are `armzillasink` and `armzillasource`. The source and sink concept as well the specification of the shared memory address, is similar to the approach of `armcosim` as discussed in Section 6.2 on page 55. The library blocks also indicate an extra parameter called `processor`, and that indicates the symbolic name of the processor to which this memory-mapped interface is attached. This symbolic name must corresponds to one of the names used in the system topology file.

The software running on each of the cores is shown in Listing 18 and Listing 19 The handshaking protocol ilustrated here is an optimized version of the two-phase full-handshake protocol described in Section 6.2 on page 55. The optimizations include the following.

- Convert the two-way request-acknowledge handshake with a one-way request-only handshake, going from the sender to the receiver. This optimization is possible when the receiver is faster than the sender, because the sender can not verify the status of the receiver and thus must assume it is always safe to send data.
- Trigger the handshaking on signal level *changes* rather than signal levels. This effectively doubles the communication bandwith with respect to a level-triggered case.
- Merge the request and data signals into a single shared memory address. This looses one bit of the useful data bandwidth, but at the same time reduces the number of memory accesses by the ARM. In a RISC processor, memory bus badnwidth is a very scarce resource. In the examples below, the most-significant bit of the data word is used as the request bit for the single-side handshake protocol.

**LISTING 18. A Sender C program of the two-ARM multiprocessor**

```
#include <stdio.h>

int main() {
  volatile unsigned int  *datap;
  int data = 0;
  int i;
  datap = (unsigned int  *) 0x80000008;

  for (i=0; i<5; i++) {
     *datap = data | 0x80000000;
     printf("Sender sends %d\n", data);
     data++;

```

```
    data &= 0x7FFFFFFF;
    *datap = data;
    printf("Sender sends %d\n", data);
    data++;
  }
  return 0;
}
```

**LISTING 19. A Receiver C program of the two-ARM multiprocessor**

```
#include <stdio.h>

int main() {
  volatile unsigned int  *datap;
  int data = 0;
  int i;
  datap = (unsigned int  *) 0x80000008;

  for (i=0; i<5; i++) {
     do
       data = *datap;
     while (!(data & 0x80000000));

     do
       data = *datap;
     while ( (data & 0x80000000));
  }
  printf("Receiver complete - last data = %d\n", data);
  return 0;
}
```

The simulation of this multiprocessor proceeds as follows. First, compile each of the sender and receiver programs into statically linked ARM-ELF executables.

```
> /usr/local/arm/bin/arm-linux-gcc -static  \
                                  sender.c -o sender.exe

> /usr/local/arm/bin/arm-linux-gcc -static \
                                  receiver.c -o receiver.exe
```

Next, run `armzilla` with the system topology file as command line argument. `armzilla` will then load the ARM executables, the GEZEL description, and start the simulation. In the output, messages printed by the sender are interleaved with messages from the GEZEL program.

```
> armzilla system.connect
Initializing StrongARM processor sender
Initializing StrongARM processor receiver
Initializing GEZEL processor network.fdl
armzillasource: set processor sender
armzillasource: set address 2147483656
armzillasink: set processor receiver
armzillasink: set address 2147483656
```

```
Cycle: 1 channel = 0
Cycle: 22951 channel = 0
Cycle: 22952 channel = 80000000
Sender sends 0
Cycle: 32662 channel = 80000000
Cycle: 32663 channel = 1
Sender sends 1
Cycle: 34230 channel = 1
Cycle: 34231 channel = 80000002
Sender sends 2
Cycle: 35759 channel = 80000002
Cycle: 35760 channel = 3
Sender sends 3
Cycle: 37296 channel = 3
Cycle: 37297 channel = 80000004

... Some output skipped ...

Total allocated OSMs : 26789
Total retired OSMs   : 26786
Total cycles         : 59393
Equivalent time on 206.4MHz host: 0.0003 sec.
Total memory pages allocated : 371
```

The output gives the shows that there are about 3000 cycles required for each iteration of the handshake protocol. However, this is due to the print statements in the sender program.

## 6.4 The gezel51 tool

`gezel51` is a cosimulator for GEZEL designs and an 8051 microcontroller. The command line is as follows

```
gezel51 [-d] [-g gezel_file] -x hex_file
```

Parameters in between square brackets are optional.

The `-d` is the debug flag. When it is enabled, the GEZEL code is run in value-change dump mode (See Section 4.4 on page 36).

The `-g gezel_file` provides the GEZEL description. This is an optional parameter. When it is absent, gezel51 will work as an 8051 instruction-set simulator.

The `-x hex_file` provides the program for the 8051 as an Intel hex formatted file.

Let's consider a small example of an 8051 cosimulation, a program that will simply transfer data values from the 8051 microcontroller to the GEZEL simulation

**LISTING 20. A GEZEL description of the 8051 'hello' coprocessor**

```
dp hello_decoder(in   ins : ns(8);
                 in   din : ns(8)) {
```

```
3.   reg insreg : ns(8);
4.   reg dinreg : ns(8);
5.   sfg decode   { insreg = ins;
6.                  dinreg = din; }
7.   sfg hello    { $display($cycle, " Hello! You gave me ", dinreg); }
8. }
9. fsm fhello_decoder(hello_decoder) {
10.   initial s0;
11.   state s1, s2;
12.   @s0 (decode) -> s1;
13.   @s1 if (insreg == 1) then (hello, decode) -> s2;
14.                        else (decode)        -> s1;
15.   @s2 if (insreg == 0) then (decode)        -> s1;
16.                        else (decode)        -> s2;
17. }
18.
19. ipblock b_ins(out data : ns(8)) {
20.   iptype "i8051source";
21.   ipparm "port=P0";
22. }
23. ipblock b_datain(out data : ns(8)) {
24.   iptype "i8051source";
25.   ipparm "port=P1";
26. }
27.
28. system S {
29.   hello_decoder(ins, din);
30.   b_ins(ins);
31.   b_datain(din);
32. }
```

The first part of the program, lines 1—17, is a one-way handshake, that accepts data values and prints them. Of particular interest for this example are the hardware/software interfaces in lines 19—26. The cosimulation interfaces with an 8051 are not memory-mapped but rather port-mapped. The 8051 has four ports, labeled P0 to P3, which are mapped to its' internal memory space but which are available as IO ports on the core. These ports are intended to attach peripherals, and in this case are used to attach a GEZEL processor. To learn more about the 8051, refer to the UCR Dalton project (`http://www.cs.ucr.edu/~dalton/i8051/`) or the numerous other sources of 8051 information on the web.

Here is a driver program in C for this coprocessor.

**LISTING 21. 8051 Driver program for the Hello coprocessor**

```
#include <8051.h>
enum {ins_idle, ins_hello};
void sayhello(char d) {
  P1 = d;
  P0 = ins_hello;
  P0 = ins_idle;
}
```

```
8. void terminate() {
9.   // special command to stop simulator
10.   P3 = 0x55;
11. }
12. void main() {
13.   sayhello(3);
14.   sayhello(2);
15.   sayhello(1);
16.   terminate();
17. }
```

The program transfers a values to the GEZEL coprocessor using `sayhello` in lines 3—8. The include file on line 1 is specific for this 8051 processor. Unlike a standard C program, a C program on the 8051 never terminates, and there is no concept of standard C library. Consequently, there are no `printf` functions and so on; these would be of little use within a micro-controller. The include file `8051.h` contains several defintions, including those of ports `P0` to `P3`. The special function `terminate` in lines 8 to 11 is used to stop the cosimulation.

The simulation proceeds as follows. First compile the 8051 program, using the Small Devices C Compiler (sdcc)

```
> sdcc driver.c
```

The compiler creates several intermediate files, as well as a hex-dump format of the compiled code in Intel Hex format, `driver.ihx`. This file can be used for `gezel51` cosimulation:

```
> gezel51 -g hello.fdl -x driver.ihx
Initializing GEZEL processor hello.fdl
Initializing 8051 processor driver.ihx
P0      P1      P2      P3
0xFF    0xFF    0xFF    0xFF
0xFF    0x03    0xFF    0xFF
9590 Hello! You gave me 3/3
0x01    0x03    0xFF    0xFF
0x00    0x03    0xFF    0xFF
0x00    0x02    0xFF    0xFF
9734 Hello! You gave me 2/2
0x01    0x02    0xFF    0xFF
0x00    0x02    0xFF    0xFF
0x00    0x01    0xFF    0xFF
9878 Hello! You gave me 1/1
0x01    0x01    0xFF    0xFF
0x00    0x01    0xFF    0xFF
0x00    0x01    0xFF    0x55
```

The output of the simulation shows the `$display` output from GEZEL, in addition to a value-change trace of the 8051's ports (`P0` to `P3`). The 8051 uses many clock cycles; there is one 'machine cycle' for each 12 clock cycles. Typically, a single instruction can execute in one machine cycle.

## 6.5 Simulation Effects

In general, the speed of a good instruction-set simulator is far higher than that of the GEZEL kernel. This is because the ISS was developed with the architecture of the processor it must model in mind, which is not possible for the GEZEL kernel. Also, the GEZEL kernel can simulate arbitrarily big hardware designs, while an ISS is focused on one single design.

As an optimization, the GEZEL simulator uses a strategy of sleep/awake modes as was discussed in Section 4.1 on page 32. This mode switching is also important for cosimulation. If this is possible, a user should develop the GEZEL hardware model in such a way that periods of idle or inactive operation also imply no datapath register changes and no state changes in the GEZEL controllers. This will result not only in a more energy efficient implementation (less useless toggling nets), but also in improved simulation speed.

# 7.0 Cosimulating GEZEL with SystemC

GEZEL supports cosimulation with SystemC, a C++ library for hardware modeling as well as system level simulation created by the Open SystemC Initiative. SystemC can be downloaded from `http://www.systemc.org`. A detailed reference manual for SystemC is included in the release available from that website.

This chapter presents the cosimulation interfaces between GEZEL and SystemC. This cosimulation interface allows users with legacy SystemC code to try out GEZEL modeling without leaving their existing system level environment.

## 7.1 Cosimulation Setup

The coupling between GEZEL and SystemC is done by integrating GEZEL as one or more modules (`SC_MODULE`) in SystemC.

- A single module is required to couple the GEZEL kernel to the SystemC kernel. In the resulting simulation, GEZEL is a slave simulator attached to a SystemC clock.
- One additional module is required for each data channel from GEZEL to SystemC or the other direction.

The same remarks as made in Section 6.1 on page 54 hold: the cosimulation interfaces for data exchange do not guarantee synchronization of the GEZEL application with the SystemC application. A synchronization protocol might still be required.

GEZEL uses a scripted approach for designs, while SystemC uses a compiled approach. Therefore, running a cosimulation must be done in two separate steps.

1. A SystemC program must be written that uses the GEZEL cosimulation interface, included in the C++ library `libgzlsysc.a`. This library is part of the standard GEZEL release, provided it has been configured with SystemC support (See Section A.4 on page 93). This SystemC program must be compiled and linked into an executable. The resulting program will have a module, the GEZEL model, which is defined by means of a filename, say `mygezel.fdl`.
2. When running the SystemC executable, the GEZEL description included in `mygezel.fdl` will be parsed in before the SystemC simulation starts, as part of the simulator initialization.

Once the SystemC executable is created, step 2 can be taken many times and each time the GEZEL description can be changed. Typically, the GEZEL parse times are only a fraction of the SystemC compilation times. When a module is interactively being debugged, GEZEL offers a more responsive way of working than the compiled approach.

## 7.2 GEZEL/SystemC Cosimulation Interfaces

As with C-based cosimulation discussed in Section 6.2 on page 55, a cosimulation interface has two sides: one side in GEZEL and one side in the cosimulated environment. The cosimulation interfaces on the GEZEL side will be captured in an `ipblock`. The cosimulation interfaces on the SystemC side will be captured in `SC_MODULE`.

GEZEL interfaces are unidrectional, and must specify the symbolic name of the interface variable they capture. An interface that transports data from GEZEL to SystemC is captured in a `systemcsink`. An interface that transports data from SystemC to GEZEL is captured in a `systemcsource`.

This example presents an interface to transport 32-bit signed numbers from SystemC to GEZEL. The symbolic interface name is `sample`, this is the name that SystemC will refer to.

```
ipblock systemc_sample(out data : tc(32)) {
  iptype "systemcsource";
  ipparm "var=sample";
}
```

This example presents an interface to transport 1-bit numbers from GEZEL to SystemC. The symbolic interface name is `output_data_ready`.

```
ipblock systemc_output_data_ready(in data : ns(1)) {
  iptype "systemcsink";
  ipparm "var=output_data_ready";
}
```

The cosimulation interfaces at the SystemC side are complementary to the above ones. They are represented as `SC_MODULE` of the type `gezel_inport` or `gezel_outport`. These module types are declared in an include file `systemc_itf.h`, which is part of the standard GEZEL installation. The counterparts for the above interfaces in SystemC are defined as follows.

```
#include “systemc_itf.h”

gezel_inport block1("block1","sample");
block1.datain(sample_int);

gezel_outport block2 ("block2","output_data_ready");
block2.dataout(output_data_ready_int);
```

The SystemC modules accept two parameters, the first being the SystemC module name, and the second the name of the interface variable. The SystemC modules each define also an input- resp output-port as `datain` resp `dataout`.

In the current version of the GEZEL/SystemC cosimulation interface, the type of these ports is fixed to `sc_int<32>`.

The SystemC/GEZEL cosimulation gives SystemC control over the GEZEL file that should be used, as well as over the clock that should be used for the simulation. A SystemC module called `gezel_module` is used for this purpose. This module is declared in `systemc_itf.h` as well.

Assuming that `clock` is an `sc_clock`, then the following SystemC definition will read in the GEZEL file `fir.fdl` upon simulation startup, and connect the GEZEL simulator to `clock`. This means that each upgoing positive edge in `clock` will result in a single cycle of GEZEL simulation.

```
gezel_module G("gezel_block", "fir.fdl");
G.clk(clock.signal());
```

In the current version of the GEZEL/SystemC cosimulation interface, only a single GEZEL file can be read in a SystemC simulation.

## 7.3 A FIR filter

To illustrate the use of the interface, design a FIR filter starting from the FIR filter included in the current SystemC 2.0.1 release. This example is a 16-bit FIR filter included under `examples/systemc/fir`. The goal is to substitute the FIR core in SystemC with an identical FIR in GEZEL, while keeping the surrounding testbench identical. First, look at the way the SystemC version of the filter is integrated (file `main_rtl.cpp`) in `sc_main`:

```
sc_clock        clock;
sc_signal<bool> reset;
sc_signal<bool> input_valid;
sc_signal<int>  sample;
sc_signal<bool> output_data_ready;
sc_signal<int>  result;

fir_top   fir_top1    ( "process_body");
fir_top1.RESET(reset);
fir_top1.IN_VALID(input_valid);
fir_top1.SAMPLE(sample);
fir_top1.OUTPUT_DATA_READY(output_data_ready);
fir_top1.RESULT(result);
fir_top1.CLK(clock.signal());
```

The block has three input ports and two output ports. Each of these ports will map to either a `gezel_inport` or a `gezel_outport`. In addition, a `gezel_module` will be required to hook up the GEZEL simulator into SystemC.

However, the signal types of the testbench do no correpond to the types supported by the cosimulation interfaces. A possible solution is to insert type translation modules in the SystemC simulation. For example, the following block converts `bool` (such as needed for `input_valid)` into an `sc_int<32>`.

```
SC_MODULE(bool2int32) {
  sc_in< bool > din;
  sc_out< sc_int<32> > dout;

  SC_CTOR(bool2int32) {
    SC_METHOD(run);
    sensitive << din;
}
  void run() {
    if (din.read()) dout.write(1); else dout.write(0);
  }
};
```

While not particularly compact nor fast, this glue code will do the job until the cosimulation interfaces will support a wider range of types. Now substitute the original `fir_top` in `main_rtl.cpp` with a GEZEL version as follows:

```
#include "systemc_itf.h"

int sc_main (int argc , char *argv[]) {

  ...

  sc_signal<sc_int<32> > reset_int;
  sc_signal<sc_int<32> > input_valid_int;
  sc_signal<sc_int<32> > output_data_ready_int;
  sc_signal<sc_int<32> > sample_int;
  sc_signal<sc_int<32> > result_int;

  bool2int32 b1("b1");
        b1.din(reset); b1.dout(reset_int);
  bool2int32 b2("b2");
        b2.din(input_valid); b2.dout(input_valid_int);
  int322bool b3("b3");
        b3.din(output_data_ready_int); b3.dout(output_data_ready);
  int2int32  b4("b4");
        b4.din(sample); b4.dout(sample_int);
  int322int  b5("b5");
        b5.din(result_int); b5.dout(result);

  gezel_module  G ("gezel_block","fir.fdl");
        G.clk(clock.signal());
  gezel_inport  fir_reset("fir_reset","reset");
        fir_reset.datain(reset_int);
  gezel_inport  fir_in_valid("fir_in_valid","input_data_valid");
        fir_in_valid.datain(input_valid_int);
  gezel_inport  fir_sample("fir_sample","sample");
        fir_sample.datain(sample_int);
  gezel_outport fir_output_data_ready (
           "fir_output_data_ready","output_data_ready");
        fir_output_data_ready.dataout(output_data_ready_int);
  gezel_outport fir_result("fir_result", "result");
        fir_result.dataout(result_int);

  ...
}
```

Five extra signals are created of type `sc_int<32>`. These are connected to the GEZEL interfaces and conversion blocks to resolve the type conversion issues discussed earlier. The `fir_top1` module will be replaced by a GEZEL file defined in `fir.fdl`. This file will replace `fir_data.cpp` and `fir_fsm.cpp`.

**LISTING 22. A FIR algorithm in GEZEL**

```
dp fir(in  reset             : ns(1);
       in  input_valid       : ns(1);
       in  sample            : tc(32);
       out output_data_ready : ns(1);
       out result            : tc(32)) {
  lookup coefs : tc(9) = {  -6,  -4,  13,  16,
                           -18, -41,  23, 154,
                           222, 154,  23, -41,
                           -18,  13,  -4, -6};
   reg rdy     : ns(1);
   reg rreset  : ns(1);
   reg rsample : tc(32);
   reg acc     : tc(19);
   reg shft0,  shft1,  shft2,   shft3 : tc(6);
   reg shft4,  shft5,  shft6,   shft7 : tc(6);
   reg shft8,  shft9, shft10,  shft11 : tc(6);
   reg shft12, shft13, shft14, shft15 : tc(6);
   sfg read {
     rsample = sample; rdy = input_valid; rreset = reset;
   }
   sfg rst  {
     acc=0;   shft0=0; shft1=0;  shft2=0;  shft3=0;
     shft4=0; shft5=0; shft6=0;  shft7=0;
     shft8=0; shft9=0; shft10=0; shft11=0;
     shft12=0;shft13=0;shft14=0; shft15=0;
   }
   sfg shft {
     shft0=rsample; shft1=shft0;   shft2=shft1;   shft3=shft2;
     shft4=shft3;   shft5=shft4;   shft6=shft5;   shft7=shft6;
     shft8=shft7;   shft9=shft8;   shft10=shft9;  shft11=shft10;
     shft12=shft11; shft13=shft12; shft14=shft13; shft15=shft14;
   }
   sfg phi0 {
     acc = shft14 * coefs(15) + shft13 * coefs(14) +
           shft12 * coefs(13) + shft11 * coefs(12) +
           sample * coefs(0);
   }
   sfg phi1 {
     acc = acc + shft10 * coefs(11) + shft9 * coefs(10)
                + shft8 * coefs(9)  + shft7 * coefs(8);
   }
   sfg phi2 {
     acc = acc + shft6 * coefs(7)   + shft5 * coefs(6)
               + shft4 * coefs(5)   + shft3 * coefs(4);
   }
   sfg phi3 {
     result = acc + shft2 * coefs(3) + shft1 * coefs(2)
                  + shft0 * coefs(1);
      output_data_ready = 1;
   }
   sfg noout { result = 0; output_data_ready  = 0;}
 }
```

```
53. fsm cfir(fir) {
54.   initial s0;
55.   state s1, s2, s3, s4;
56.
57.   @s0 (read, noout, phi0) -> s1;
58.   @s1 if (rreset)   then (read, rst, noout)  -> s1;
59.       else if (rdy) then (read, noout, phi1) -> s2;
60.            else (read, noout, phi0)          -> s1;
61.   @s2 (noout, phi2)      -> s3;
62.   @s3 (phi3, shft, read) -> s1;
63. }
64.
65. // interfaces
66. ipblock systemc_reset(out data : ns(1)) {
67.   iptype "systemcsource";  ipparm "var=reset";
68. }
69. ipblock systemc_input_valid(out data : ns(1)) {
70.   iptype "systemcsource";  ipparm "var=input_data_valid";
71. }
72. ipblock systemc_sample(out data : tc(32)) {
73.   iptype "systemcsource";  ipparm "var=sample";
74. }
75. ipblock systemc_output_data_ready(in data : ns(1)) {
76.   iptype "systemcsink";  ipparm "var=output_data_ready";
77. }
78. ipblock systemc_result(in data : tc(32)) {
79.   iptype "systemcsink";  ipparm "var=result";
80. }
81.
82. system S {
83.   fir(reset, input_valid, sample, output_data_ready, result);
84.   systemc_reset(reset);
85.   systemc_input_valid(input_valid);
86.   systemc_sample(sample);
87.   systemc_output_data_ready(output_data_ready);
88.   systemc_result(result);
89. }
```

The filter has the same data I/O as the set of interfaces in the SystemC main function (lines 2—5). The filter coeffcients are included as a lookup array (line 6—9). The design also includes registers for condition flags as well as for an accumulator and filter taps for the FIR (lines 10—17). The datapath is composed of a series of instructions that can evaluate the 16 filter taps in four clock cycles (lines 18—52). Thus, there are four multiply-accumulates per instruction.

The filter controller description starts from line 53. The sequencing is dependent on the `reset` input (which will reset the set of filter taps) and the `input_available` input. As soon as this control signal is asserted, the filter will go into one iteration of the algorithm and evaluated the 16 taps of the filter in four subsequent instructions.

The SystemC interfaces are expressed in lines 65—80. These interfaces are simply the couterparts of the ones we have added in `main_rtl.cpp`. Finally, a `system` block connects the filter core to the interfaces.

Now you can compile the SystemC description, link the cosimulator and start the simulation. The compile command for `main_rtl.cpp`, assuming an installation under `/home/guest` looks as follows:

```
> g++ -O3 -Wall -c -I/home/guest/systemc-2.0.1/include \
                   -I/home/guest/gezel/build/include/gezel \
                  main_rtl.cpp -o main_rtl.o
```

The link command that creates the cosimulator is as follows.

```
> g++ -g stimulus.o display.o fir_fsm.o fir_data.o \
      main_rtl.o -L/home/guest/gezel/build/lib/ \
      -lgzlsysc -lfdl -lgzlsysc \
      -L/home/guest/systemc-2.0.1/lib-linux -lsystemc \
     -lgmp -o systemc_cosim
```

## 7.4 Why GEZEL with SystemC ?

The goals of GEZEL and SystemC are not the same. GEZEL focuses on easy modeling of cycle-true micro-architectures. SystemC focuses on solving system integration problems. Both have their place in the design of a complete system.

GEZEL will be particularly useful when any of the following is an issue or critical requirement for you.

- **Compilation Time**. GEZEL does not need to be compiled; it is parsed and interpreted. Compiling a model over and over again for each modification takes time, even if it's only a few seconds for each iteration. For example, in the above FIR filter example, if we make a small modification inside of the SystemC model (in `fir_data.cpp`), then recompiling that file and relinking the SystemC model takes 2.5 seconds on a 3GHz Pentium PC. Making a modification to `fir.fdl` on the other hand and reloading the file takes less then 0.1 seconds.

  In addition, the authors have shown in their research that GEZEL achieves the same simulation speed at cycle-true level as SystemC. So, *interpreted* does not have to mean *slow*.
- **Code Generation**. GEZEL has a build-in path to VHDL implementation.
- **Error Messages**. GEZEL generates error messages directly upon parsing, and directly in terms of the FSMD model. A SystemC design generates C++ error messages. This difference has important consequences on readability, and is most easy to demonstrate by means of an example. Next are two error messages for a similar type of error. The first one is from SystemC:

```
main_gezel.cpp:95:
error: no match for call to `(sc_out<sc_dt::sc_int<32> >) (
  sc_signal<bool>&)'
```

```
/home/schaum/systemc-2.0.1/include/systemc/communication/
sc_port.h:230: error: candidates
   are: void sc_port_b<IF>::operator()(IF&)
        [with IF = sc_signal_inout_if<sc_dt::sc_int<32> >]
/home/schaum/systemc-2.0.1/include/systemc/communication/
sc_port.h:239: error:
               void sc_port_b<IF>::operator()(sc_port_b<IF>&)
               [with IF = sc_signal_inout_if<sc_dt::sc_int<32> >]

make: *** [main_gezel.o] Error 1
```

And here is the error message for a similar offense from GEZEL:

```
*** Error: Signal has no driver S.reset

Context:
(96)     }
(97)
(98)     system S {
(99) >>>   fir(reset,input_valid,sample,output_data_ready,result);
(100)      systemc_reset(output_data_ready);
(101)      systemc_input_valid(input_valid);
```

Without going into the details of the exact error cause, the issue we are pointing at is easy to see.

- **Simplicity**. GEZEL has simple FSMD semantics, easy to learn and to remember. Simple is not always better, but if the task is well defined as the creation of a micro-architecture using FSMD, then GEZEL might just be the tool you need.

# 8.0 GEZEL Library Blocks

GEZEL supports predesigned library blocks. At the outside, these look like datapath modules, and they can be used in the same way. However, their inside is not written in GEZEL code. Instead, the behavior of library blocks is written in C++ and compiled directly into the GEZEL kernel. This enables blocks that run much faster than cycle-true models in GEZEL, for example by raising the simulation abstraction level. It also allows to introduce features in a GEZEL simulation that are not supported in GEZEL code, such as special types of IO or host system function calls.

This chapter presents the use of GEZEL library blocks. This includes the general modeling and usage properties of library blocks, as well as a catalog of available library blocks. And finally, you can also introduce your own library blocks into the GEZEL kernel.

## 8.1 Library Blocks Definition

GEZEL library blocks are created with the keyword `ipblock`. They define three elements. First, they define their IO interface, just as a datapath module does. Second, they define their *type*. And third, they define an optional number of *parameters*. Consider the RAM library block as an example. This block is part of the standard configuration GEZEL kernel, and is used to simulate a RAM memory. It has an outline that correponds to RAM cell; it define an address bus, a data input and — output bus, and read/write control lines. The RAM library block is parametrizable in size and wordlengh, as well. The following GEZEL definition creates a RAM block of 32 positions, of 8-bit words.

```
ipblock M(in address : ns(5);
          in wr,rd   : ns(1);
          in idata   : ns(8);
          out odata  : ns(8)) {
  iptype "ram";
  ipparm "wl=8";
  ipparm "size=32";
}
```

A library block with name `M` is created. It defines five ports. including the address bus (`address`), write and read control strobes (`wr`, `rd`), an input data bus (`idata`) and an output data bus (`odata`).

Next is an indication of the library block type, in the `iptype` statement. Then, a number of library block parameters can be given. These are dependent on the type of the library block. For a RAM, the worldlength of the databus (`wl`) and the number of memory locations (`size`) can be specified. GEZEL will issue a warning when a parameter field is found that is not supported by the library block.

The names as well as the order of the ports of a library block are determined by the type. For a particular type, there is only one ordered set of names, with each port having a predefined direction. For a library block of type `ram` for example, the first port *must* be called `address` and it *must* be an input. GEZEL will issue a warning when there is a mismatch detected. Once a library block is instantiated, it can be used like any other datapath module.

Library blocks can be included within other datapaths using the `use` statement (Section 2.7 on page 18).

The next program fills up the RAM module defined above, and next reads it out again.

**LISTING 23. A RAM library block testbench**

```
1. ipblock M(in address : ns(5);
2.      in wr,rd   : ns(1);
3.      in idata   : ns(8);
4.      out odata  : ns(8)) {
5.   iptype "ram";
6.   ipparm "wl=8";
7.   ipparm "size=32";
8. }
9.
10. dp tmac(out address : ns(5);
11.    out wr, rd   : ns(1);
12.    out idata    : ns(8);
13.    in  odata    : ns(8)) {
14.    reg ar       : ns(5);
15.    reg idr, odr : ns(8);
16.
17.   sfg write  { wr = 1; rd = 0; idata = idr; odr = odata; address = ar;
18.      $display($cycle, ":ar ", ar, " idata ", idata);
19.    }
20.   sfg read   { wr = 0; rd = 1; address = ar; odr = odata; idata = idr;
21.      $display($cycle, ":ar ", ar, " odata ", odata);
22.    }
23.   sfg incadr  { ar = ar + 1; idr = idr + 1;}
24.   sfg clraddr { ar = 0; }
25. }
26.
27. fsm ftmac(tmac) {
28.   state   s1;
29.   initial s0;
30.   @s0 if (ar == 4) then (write, clraddr ) -> s1;
31.   else   (write, incadr)  -> s0;
32.   @s1 if (ar == 4) then (read,  clraddr)  -> s0;
33.   else   (read,  incadr)   -> s1;
34. }
35.
36. system S {
37.   M   (adr, w, r, i, o);
38.   tmac(adr, w, r, i, o);
39. }
```

The testbench that drives the RAM module is included in lines 10—34. The first five locations of the RAM are first written with an increasing number sequence, and next these five locations are read out again.

## 8.2 Catalog of Library Blocks

The following table enumerates the different library blocks. Some library blocks, such as cosimulation interfaces, are part of a particular simulator configuration.

| **Library Blocks in the GEZEL Kernel (available for all programs)** | | | |
|---|---|---|---|
| ram | Function | The RAM block implements a RAM cell with separate read and write control strobes, and a separate data input and data output. One read and one write access is possible per clock cycle. | |
| | IO | `address` | input, ns(log2(size)), holding the address for the RAM. |
| | | `wr` | input, ns(1). write asserted high. |
| | | `rd` | input, ns(1), read asserted high. |
| | | `idata` | input, ns(wl), input data bus. |
| | | `odata` | output, ns(wl), output data bus. |
| | Parameters | `wl` | wordlength of data bus (wl>0) |
| | | `size` | number of RAM locations. |
| tracer | Function | The tracer block implements equivalent functionality as the `$trace` directive, and records the values of a signal into a file at each clock cycle. | |
| | IO | `data` | input, ns(userdefined), data input |
| | Parameters | `file` | quoted string with filename |
| | | `wl` | wordlength of numbers in file |
| rom | Function | Contributed by A.V.Lorentzen and J.Steensgaard-Madsen, DTU (Denmark's Technical University). Originating from an implementation of Andrew Tanenbaum's Mic-1 that conforms to Ray Ontko's Java simulator for it.<br>The ipblock may represent an initialised Mic-1 microcontrol store. The contents may come from a file generated by the microprogram assembler provided by Ray Ontko. Beware of possible endianness problems if you want to generate the contents differently. | |
| | IO | `address` | input, ns(log2(size)) holding an address of the least number of 8-bit bytes capable of holding one word of wl-bits (left justified) |
| | | `rd` | input, ns(1), read asserted high |
| | | `odata` | output, ns(wl), output data |
| | Parameters | `size` | number of locations |
| | | `wl` | wordlength |
| | | `file` | name of file with initial contents |
| | | `startbyte` | (default 4), number of bytes to skip in file |

| | | | |
|---|---|---|---|
| ijvm | Function | Contributed by A.V.Lorentzen and J.Steensgaard-Madsen, DTU (Denmark's Technical University). Originating from an implementation of Andrew Tanenbaum's Mic-1 that conforms to Ray Ontko's Java simulator for it. The ipblock may represent a Mic-1 store with an ijvm-program preloaded. The file from which the preloaded program is read may be generated by the ijvm-assembler provided by Ray Ontko. The current version conforms completely to Ray's programs, including the restriction to just two code-sections. Parameters `sp`, `lv`, and `cpp` must be bound to the initial values of the Mic-1 registers with these names. They do not need to be set to the values chosen in Ray Ontko's Java simulator. | |
| | IO | `address` | input, ns(log2(size)), holding the address of a 32-bit data word |
| | | `wr` | input, ns(1), write asserted high |
| | | `rd` | input, ns(1), read asserted high |
| | | `idata` | input, tc(32), input data bus |
| | | `odata` | output, tc(32), output data bus for values |
| | | `bytes` | input, ns(log2(size)) holding the address of a 4-bytes code sequence |
| | | `fetch` | input, ns(1), read asserted high |
| | | `byteval` | output, ns(32), output data bus for code |
| | Parameters | `size` | number of locations |
| | | `file` | name of file with initial contents |
| | | `sp` | initial value of register stack pointer register |
| | | `lv` | initial value of local variables register |
| | | `cpp` | initial value of constant pool pointer register |
| filesource | Function | The filesource block allows to fetch stimuli from an external file. Wordlength and representation base are parametrizable. The block has a variable number of data ouputs. When the user wires this block up with for example two outputs, then two values will be read from a file for each clock cycle of simulation. | |
| | IO | `d1` | first output |
| | | `d2` | optional second output |
| | | `..` | up to 10 optional outputs are supported |
| | Parameter | `file` | string, name of the file to read. |
| | | `wl` | integer, wordlength |

| | | | |
|---|---|---|---|
| | | `base` | interger, base in which values in the file are expressed. Symbols of the set [0-9a-z] are used to represent values in non-decimal bases. |
| **Library Blocks in `armcosim` (Section 6.2 on page 55)** | | | |
| armsimsource | Function | Memory-mapped Cosimulation interface to transport data from ARM to GEZEL. | |
| | IO | `data` | output, ns(32), data output. |
| | Parameters | `address` | Address of the ARM address space that matches this cosimulation interface port. |
| armsimsink | Function | Memory-mapped Cosimulation interface to transport data from GEZEL to ARM. | |
| | IO | `data` | input, ns(32), data input |
| | Parameters | `address` | Address of the ARM address space that matches this cosimulation interface port. |
| armsimcp | Function | Coprocessorport-based Cosimulation interface to transport data between GEZEL and ARM, bidirectionally. | |
| | IO | `din` | output, ns(32), data channel ARM to GEZEL |
| | | `dout` | input, ns(32), data channel GEZEL to ARM |
| | | `adr` | output, ns(32), additional data channel from ARM to GEZEL, for multiplexing purposes. |
| | | `req` | output, ns(1), request handshake |
| | | `ack` | input, ns(1), ack handshake |
| | | `rd` | output, ns(1), read ARM to GEZEL strobe |
| | Parameters | `device` | holds the device number on which this interface must trigger. |
| **Library Blocks in `armzilla` (Section 6.3 on page 59)** | | | |
| armzillasource | Function | Memory-mapped Cosimulation interface to transport data from ARM to GEZEL. | |
| | IO | `data` | output, ns(32), data output. |
| | Parameters | `process` | quoted string, symbolic name of the ARM in whose address space this interface is located. |
| | | `address` | Address of the ARM address space that matches this cosimulation interface port. |
| armzillasink | Function | Memory-mapped Cosimulation interface to transport data from ARM to GEZEL. | |
| | IO | `data` | input, ns(32), data input. |

| | | | |
|---|---|---|---|
| | Parameteres | `process` | quoted string, symbolic name of the ARM in whose address space this interface is located. |
| | | `address` | Address of the ARM address space that matches this cosimulation interface port. |
| armzillashmem | Function | Shared-memory between multiple ARM and GEZEL. | |
| | Parameteres | `wl` | quoted string, expressing the wordlength of the simulation. |
| | | `size` | Number of memory locations in the shared memory |
| | | `access` | Address range mapping. Multiple mappings can be given, one for each arm. Format: `name@address` |
| | | | where name is the name of a processor, and address is the location where the shared memory starts. |
| armzillashmem_ writeport | Function | GEZEL writeport on a shared memory. | |
| | IO | `address` | input, ns(log2(size)). address value to write. |
| | | `en` | input, ns(1). control pin. When one, write operation proceeds. |
| | | `data` | input, ns(wl). Value to write. |
| | Parameteres | `shmem` | quoted string. Name of the shared memory (the name of the ipblock) to connect this writeport onto. |
| armzillashmem_r eadport | Function | GEZEL readport on a shared memory. | |
| | IO | `address` | input, ns(log2(size)). address value to write. |
| | | `en` | input, ns(1). control pin. When one, read operation proceeds. |
| | | `data` | output, ns(wl). Value to read. |
| | Parameteres | `shmem` | quoted string. Name of the shared memory (the name of the ipblock) to connect this readport onto. |
| armzillacp | Function | Coprocessorport-based Cosimulation interface to transport data between GEZEL and ARM, bidirectionally. | |
| | IO | `din` | output, ns(32), data channel ARM to GEZEL |
| | | `dout` | input, ns(32), data channel GEZEL to ARM |
| | | `adr` | output, ns(32), additional data channel from ARM to GEZEL, for multiplexing purposes. |

| | | | |
|---|---|---|---|
| | | `req` | output, ns(1), request handshake |
| | | `ack` | input, ns(1), ack handshake |
| | | `rd` | output, ns(1), read ARM to GEZEL strobe |
| | Parameters | `process` | quoted string, symbolic name of the ARM in whose address space this interface is located. |
| | | `device` | holds the device number on which this interface must trigger. |

| **Library Blocks in `gezel51` (Section 6.4 on page 63)** | | | |
|---|---|---|---|
| i8051source | Function | Port-mapped cosimulation interface to transport data from 8051 to GEZEL. | |
| | IO | `data` | output, ns(32), data output. |
| | Parameters | `port` | quoted string, one of `P0`, `P1`, `P2`, `P3`. |
| i8051sink | Function | Port-mapped cosimulation interface to transport data from GEZEL to 8051 to GEZEL. | |
| | IO | `data` | input, ns(32), data input. |
| | Parameters | `port` | quoted string, one of `P0`, `P1`, `P2`, `P3`. |
| **Library Block in `libgzlsys.a` (SystemC cosimulator, Section 7.2 on page 67)** | | | |
| systemcsource | Function | Cosimulation interface to transport data from SystemC to GEZEL. | |
| | IO | `data` | output, ns(32), data channel from SystemC to GEZEL |
| | Parameters | `var` | string, indicates the symbolic name of the corresponding SystemC channel. |
| systemcsink | Function | Cosimulation interface to transport data from GEZEL to SystemC. | |
| | IO | `data` | input, ns(32), data channel from GEZEL to SystemC |
| | Parameters | `var` | string, indicates the symbolic name of the corresponding SystemC channel. |

## 8.3 Synthesis View of Library Blocks

Library blocks are converted to black boxes in the resulting VHDL netlist, with the same module name as the instance name in the GEZEL file. In addition, a clock and reset pin are added that are attached to the system clock and reset net. Going back to the RAM module defined in Section 8.1 on page 75, the VHDL outline of this block as obtained with `fdlvhd` looks as follows.

```
component M
   port(
      address:in std_logic_vector(4 downto 0);
```

```
           wr:in std_logic;
           rd:in std_logic;
           idata:in std_logic_vector(7 downto 0);
           odata:out std_logic_vector(7 downto 0);
           RST : in std_logic;
           CLK : in std_logic
        );
```

## 8.4 Custom Library Blocks

Finally, you can add custom library blocks to the GEZEL kernel. Adding custom library blocks allows you to cope with a variety of design problems. Some examples are as follows.

- Including legacy C code (jpeg code, crypto libraries) in a GEZEL simulation.
- Adding new cosimulation interfaces, for example including socket or IPC communication primitives to enable network-based cosimulation.
- Adding advanced I/O capabilities, for example formatting blocks that create graphical output either directly on the screen or else into a file.
- Adding advanced runtime analysis capabilities, such as a block that records the histogram of values on a bus.

There are three steps to take in order to create a new custom library block. First, you must decide how the outline of the block looks like, as well the parameter set you will support with it. Next, you have to develop the behavior of the block in C++. And finally, you have to integrate the block into GEZEL, recompile the GEZEL kernel and relink it to your application.

**Step 1 - Design the outline and functionality.** The first step is to decide on the outline of the block. Indeed, before starting to write C++ code, it is useful to write out in GEZEL code how the block will look like and think about the desired behavior.

As an example, we will develop a runlength encoding block. A runlength encoder creates a compact, tuple-based representation of a sequence of numbers. For example, if a runlength encoder reads the number string

```
1, 1, 1, 3, 4, 4, 6, 6, 6, 6
```

Then it would produce a tuple sequence with (value, count) tuples as

```
(1, 3), (3, 1), (4, 2), (6, 4)
```

We will use an outline that looks as follows.

```
ipblock my_rle(in data : ns(8);
               out tupdat : ns(8);
               out tupnum : ns(8)) {
  iptype “rle”;
```

```
    ipparm “maxlen=32”;
  }
```

The block reads 8-bit data input values and performs runlength encoding on them. The block has two outputs that will provide runlength-encoded data. The type of the block is *rle* (runlength encoder), and it supports one parameter call `maxlen`. This number holds the maximum runlength that we’ll allow before a codeword is forced. In the example we allow a maximum runlength of 32. This means that, if the input data would consist of 34 consecutive zeroes, then we expect the output to consist of two runlength tuples, one `(0,32)` and the next `(0,2)`.

There is still one issue to address. Library blocks are cycle-true functions. This means we need to develop the function in such a way that it can read input and produce output each clock cycle. For a runlength encoder, an output will not be available each clock cycle however. We will deal with this situation in our runlength encoder by producing dummy output tuples for which `tupnum` equals to zero.

We thus can express the behavior of the runlength encoder in pseudocode as follows.

```
intialize:
  previous_input_data = not_a_number;
  runlength = 0;

execute:
  read input data;
  (tuplenum, tupledata) = (0,0);
  if (input_data == previous_input_data) {
    runlength = runlength + 1;
    if (runlength == maxlen) {
      (tuplenum, tupledata) = (runlength, input_data);
      runlength = 0;
    }
  } else {
    if (runlength != 0) {
     (tuplenum, tupledata) = (runlength, previous_data);
    }
    runlength = 1;
  }
  previous_input_data = input_data;
  write (tuplenum, tupledata);
```

**Step 2 - Design the C++ implementation.** We are now ready to design the block into a GEZEL library block. Library blocks are derived from a baseclass `aipblock`. This block has a number of virtual functions that can be user-defined in derived classes.

```
class aipblock {
 protected:
  enum iodir {input, output};
 public:
  vector<gval *> ioval;
  aipblock(char *_name);
  virtual ~aipblock();
```

```
    virtual void run();
    virtual void setparm(char *_name);
    virtual bool checkterminal(int n, char *tname, iodir dir);
    virtual bool needsWakeupTest();
    virtual bool cannotSleepTest();
    virtual void touch();
  };
```

The vector `ioval` contains the values appearing on the actual ports. The first element of this vector corresponds to the first port, the second element to the second port, and so on.

The function `run` is called each clock cycle to execute the block.

The function `setparm` is called when the GEZEL parser finds a field `ipparm`. The argument of this function contains the quoted string that is found in the GEZEL code. For example, when the GEZEL code contains `ipparm “maxlen=32”` then the argument of `setparm` will be `“maxlen=32”`.

The function `checkterminal` is called by GEZEL for each port. It allows to verify that the user of the GEZEL block has used the correct names and directions of the ports of this block. The function returns a boolean, which must return true of no problem is found. The arguments of the function correspond  to the data found in the GEZEL program. `n` holds the port index, with the first port having index 0. `tname` holds the name the user of the block has used for the port. `dir` indicates if it is an input or an output.

The functions `needsWakeupTest()` and `cannotSleepTest()` are used to support the sleep mode of the GEZEL simulator (See Section 4.1 on page 32). When the simulator is running, each clock cycle the function `cannotSleepTest()` is called. This function needs to return `true` if sleep mode cannot be started. Once the simulator is in sleep mode, the function `needsWakeupTest()` is called every skipped clock cycle. The function returns `true` when the GEZEL simulation needs to wake up again. The function `touch` is used in the context of cosimulation interfaces, to force the next call to `needsWake-upTest` to return `true`. To get insight into these different functions, it is best to study one of the cosimulation interfaces of the GEZEL tools. For example, file `arm_itf.cxx` in `armcosim`.

Listing 24 shows how to program the runlength encoder as a derived class from the base class `aipblock.`

**LISTING 24. A runlength encoder library block for GEZEL**

```
#include "ipblock.h"

class rle : public aipblock {
  int previous_data_value;
  int runlength;
  int maxlen;
public:
  rle(char *name) : aipblock(name) {
    previous_data_value = -1;
10.     runlength = 0;
11.     maxlen = 256;
12.   }
13.   void run() {
14.     ioval[1]->assignulong(0);
15.     ioval[2]->assignulong(0);
16.     if (ioval[0]->toulong() == (unsigned) previous_data_value) {
17.       runlength = runlength + 1;
18.       if (runlength == maxlen) {
19.         ioval[1]->assignulong(runlength);
20.         ioval[2]->assignulong(ioval[0]->toulong());
21.         runlength = 0;
22.       }
23.     } else {
24.       if (runlength != 0) {
25.         ioval[1]->assignulong(runlength);
26.         ioval[2]->assignulong(previous_data_value);
27.       }
28.       runlength = 1;
29.     }
30.     previous_data_value = ioval[0]->toulong();
31.   }
32.   bool checkterminal(int n, char *tname, aipblock::iodir dir) {
33.     switch (n) {
34.     case 0:
35.       return (isinput(dir) && isname(tname, "data"));
36.       break;
37.     case 1:
38.       return (isoutput(dir) && isname(tname, "tuplenum"));
39.       break;
40.     case 2:
41.       return (isoutput(dir) && isname(tname, "tupledata"));
42.       break;
43.     }
44.     return false;
45.   }
46.   void setparm(char *_name) {
47.     gval *v = make_gval(32,0);
48.     if (matchparm(_name, "maxlen", *v))
49.       maxlen = v->toulong();
50.     else
51.       printf("Error: rke does not recognize parameter %s\n",_name);
52.   }
53.   bool cannotSleepTest() {
54.     return false;
55.   }
56. };
```

The constructor (lines 8—12) and the runtime function (lines 13—31) correspond to the initialization part and the execution part of the pseudocode shown earlier. The data type of the `ioval` array is `gval` (defined in `gval.h` of the GEZEL release). The functions `assignulong` and `toulong` provide conversions from and to C data types. Of course, these conversions can loose precision if the GEZEL wordlength exceeds that of the wordlength of a C `long`.

The port verification method `checkterminal` in lines 32—45 checks if the port names chosen by the GEZEL user correspond to the ones we have chosen for the runlength encoder outline, as shown earlier.

The `setparm` method in lines 46—52 accepts parameters, if any. The function call `matchparm` is a member of `aipblock` and helps in parsing the parameters. Finally, the function `cannotSleepTest` illustrates the minimal implementation for a block that does not affect the sleep-mode mechanism of the GEZEL simulator.

**Step 3 - Integrate the block and test it.** The final step is to integrate the C++ code of the library block into GEZEL. We show one possible way to integrate the block as part of the GEZEL code.

Copy the C++ code into the `gezel/` subdirectory of your GEZEL installation. For example, if your installation is under `/home/guest/gezelcode`, then execute

```
> cp iprle.h /home/guest/gezelcode/gezel
```

Make changes to the ipblock configuration file, `ipconfig.cxx`, such that your block will be available for standard GEZEL programs. In particular, find the function `ipblockcreate` inside of `ipconfig.cxx` and include the block `iprle` in the list of `CREATE()` macro's.

Next, modify the compiler makefiles such that the new files will be compiled together with the existing GEZEL files. In particular, edit the file `Makefile.am` in the `gezel/` subdirectory and add files to the `CORELIB` and `pkginclude_HEADERS` as needed.

After these steps, rerun the configuration and recompile GEZEL (See Section A on page 88).

Now you have obtained a GEZEL simulator that can work with an new library block. An example GEZEL program that uses the runlength encoder program is shown

**LISTING 25. A runlength encoder testbench**

```
ipblock my_rle(in data : ns(8);
               out tuplenum : ns(8);
               out tupledata : ns(8)) {
  iptype "rle";
  ipparm "maxlen=32";
}

dp senddata(out data      : ns(8);
            in  tuplenum  : ns(8);
            in  tupledata : ns(8)) {
  lookup T : ns(8) = {1, 1, 1, 3, 4, 4, 6, 6, 6, 6};
  reg c : ns(8);

  sfg run {
   c = (c == 9) ? 0 : c + 1;
   data = T(c);
    $display($cycle, ": ", data, " -> (", tuplenum, ", ", tupledata,
")");
  }
}
hardwired hsenddata(senddata) {run; }

system S {
  my_rle(i, tn, td);
  senddata(i, tn, td);
}
```

The program generates the following output to confirm the correct operation of the run-length encoder.

```
> fdlsim rle.fdl 15
1:  1 -> (0, 0)
2:  1 -> (0, 0)
3:  1 -> (0, 0)
4:  3 -> (3, 1)
5:  4 -> (1, 3)
6:  4 -> (0, 0)
7:  6 -> (2, 4)
8:  6 -> (0, 0)
9:  6 -> (0, 0)
10: 6 -> (0, 0)
11: 1 -> (4, 6)
12: 1 -> (0, 0)
13: 1 -> (0, 0)
14: 3 -> (3, 1)
15: 4 -> (1, 3)
Activity(%) on 1 registers: 100 (15/15)
```

# Appendix A: Installing GEZEL

The homepage URL for GEZEL is at `http://www.ee.ucla.edu/~schaum/gezel`. GEZEL can be downloaded as a source-only package. A precompiled package is available for Linux as well. The source package consists of several components.

- A standalone simulator for GEZEL code.
- A VHDL code generator to convert GEZEL code.
- A cosimulator with an ARM ISS (SimIt-ARM-2.0.x).
- A multiprocessor version of the ARM cosimulator.
- A cosimulator for an 8051 ISS.
- A cosimulator for SystemC 2.0.1

After downloading the package, uncompress it using `tar`:

```
> tar xfvz gezel-1.x.tar.gz (On Linux)

> cd gezel-1.x
```

## A.1 Standalone tools

GEZEL is written in C++ and compiles under a standard GNU build environment using the GNU C compiler, GCC. The package has automatic configuration.

Apart from GCC, you will also need the following:

- GNU Multiprecision Library (`http://www.swox.com/gmp/`)
- The Bison/YACC parser generator (`http://www.gnu.org/software/bison/bison.html`), in case you make modifications to the `fdl.y` parser.
- The Flex/Lex lexical analyzer (`http://www.gnu.org/software/flex/`), in case you make modifications to the `fdl.ll` scanner, or if the installed version of flex is imcompatible with the version installed on the build machine.

To create the makefiles on your system, execute

```
> ./configure
```

If the GNU Multiprecision library is not installed in a standard location, you will need to define CPPFLAGS and LDFLAGS as arguments to configure. For example, assuming the GMP library (`libgmp.a/so`) is installed under `/opt/gmp/lib` and the include files for GMP are under `/opt/gmp/include`, then you would run

```
> ./configure CPPFLAGS=-I/opt/gmp/include LDFLAGS=-L/opt/gmp/lib
```

If you want to select an installation directory, use the `--prefix` option of configure. The default installation directory is the `./build` subdirectory from where you installed the

GEZEL source. Use the `--help` option in configure to see a list of available command line options.

Next type:

```
> make
```

followed by

```
> make install
```

The default configuration will create the library, the stand-alone simulator and the code generation. The standalone simulator is call `fdlsim`. You can test the simulator on one of the examples in the `test/` directory. For example, 8 cycles from the Bresenham vector generator application can be simulated using:

```
> cd test/gezel

> ../../build/bin/fdlsim bresen.fdl 8
Cycle: 2 Plot point (5/5,2/1)
Cycle: 3 Plot point (5/5,1/0)
Cycle: 4 Plot point (5/5,0/-1)
Cycle: 5 Plot point (5/5,-1/-2)
Cycle: 6 Plot point (5/5,-2/-3)
Cycle: 7 Plot point (5/5,-3/-4)
Cycle: 8 Plot point (5/5,-4/-5)
Activity(%) on 10 registers: 27.5 (22/80)
FDL Cycles 8, sleeping cycles 0(0%)
```

The use of `fdlsim` is discussed in Section 4.0 on page 32. You can also convert the Bresenham vector generator from GEZEL to VHDL using `fdlvhd`, the code generator.

```
> cd test/gezel

> ../../build/bin/fdlvhd bresen.fdl
Generate file: bresen.vhd
Generate file: test_bresen.vhd
Generate file: system.vhd
```

The generated files are self-contained apart from a dependency on a VHDL package `std_logic_arithtext`. This package is included in the source code directory (`gezel/`), as file `std_logic_arithext.vhd`.

The use of the code generator and the generated VHDL files is discussed in Section 5.0 on page 41.

## A.2 SimIt-ARM Cosimulators

GEZEL can be cosimulated with instruction set simulators, using the C++ API on the backend of GEZEL. The source distribution of GEZEL contains a cosimulator with SimIt-ARM, a StrongARM instruction set simulator developed by W. Qin at Princeton University, NJ. The homepage for SimIt-ARM is `http://sourceforge.net/projects/simit-arm/`.

The cosimulation is written on top of Version 2.0 (or later) of SimIt-ARM. After you have downloaded SimIT-ARM, unpack it.

```
> tar zxfv SimIt-ARM-2.0.tgz
```

SimIT-ARM-2.0 has a built-in cosimulation interface, that must be enabled with the macro `COSIM_STUB` while the packge is configured and installed.

```
> cd SimIt-ARM-2.0

> ./configure CPPFLAGS=-'DCOSIM_STUB'

> make

> make install
```

This will install the SimIt-ARM ISS (as stand-alone libraries as well as executables) under `SimIt-ARM-2.0/build`.

Next we will build the cosimulator in GEZEL. There are two versions of the cosimulator. The first one is called `armcosim` and supports a single ARM, the second one is `armzilla` and supports a network of ARM. The latter one is slightly more complicated to use than the single-processor version. A typical use of `armcosim` is to validate a coprocessor developed in GEZEL in a cosimulation. A typical use of `armzilla` is to test a network-on-chip developed in GEZEL with several ARM applications as network clients.

To configure GEZEL for the ARM cosimulators, there are a few extra parameters to the `configure` command. The `--enable-armcosim` flag will enable compilation of `armcosim`. The `--enable-armzilla` flag will enable compilation of `armzilla`. Both flags can be used separately or at the same time. You also need to specify the path where the SimIt-ARM library installation can be found with the `--with-simit` configuration option. As an example, we will configure to compile both ARM cosimulators as follows:

```
> ./configure --enable-armcosim --enable-armzilla \
--with-simit=/home/guest/SimIT-ARM-2.0/build
```

If you happen to have the GMP library installed in a non-standard location, do not forget to include CPPFLAGS and LDFLAGS for that one as well. For example,

```
> ./configure --enable-armcosim --enable-armzilla \
--with-simit=/home/guest/SimIT-ARM-2.0/build \
CPPFLAGS='I/opt/gmp/include' LDFLAGS='-L/opt/gmp/lib'
```

Next, make and install the cosimulator in GEZEL. This will create, in the directory `build`/, an executable `armcosim` as well as an executable `armzilla`:

```
> make
> make install
```

To run the `armcosim` cosimulator, you need to provide a GEZEL file and an ARM-ELF executable. The ARM-ELF executable must be statically linked. These executables can be created using an ARM cross-compiler. This compiler can be downloaded for example from the ARM-Linux FTP site (`ftp://ftp.arm.linux.org.uk`).

The example in `test/armcosim/hshake` illustrates a two-way memory-mapped handshake between ARM software and GEZEL hardware. You can compile and run it as follows.

```
> cd test/armcosim/hshake

> /usr/local/arm/bin/arm-linux-gcc -static\
 hshakedriver.c -o hshakedriver

> ../../../build/bin/armcosim -g hshake.fdl -a hshakedriver
Initializing StrongARM processor arm_processor_0
Initializing GEZEL processor hshake.fdl
armsimsink: set address 2147483648
armsimsource: set address 2147483652
armsimsource: set address 2147483656
data received 0 cycle 28723
data received 1 cycle 28878
data received 2 cycle 28921
data received 3 cycle 28964
data received 4 cycle 29007
.. etc
```

To run the `armzilla` cosimulator you need to provide three kind of files:

- A GEZEL file representing the hardware model
- An ARM-ELF executable for each ARM ISS in the system. These ARM-ELF executables need to be statically linked.
- A CONNECT file describing the configuration of the simulator. This file specifies the names of ARM-ELF executables, their symbolic processor name, and the GEZEL file name.

The example in test/armzilla/ssidehsk illustrates a one-way handshake between two ARM processors, with GEZEL implementing the communication channel in between those.

```
> cd test/armzilla/ssidehsk

> make
/usr/local/arm/bin/arm-linux-gcc -static  sender.c -o sender.exe
/usr/local/arm/bin/arm-linux-gcc -static  receiver.c \
   -o receiver.exe

> make sim
../../../build/bin/armzilla system.connect
Initializing StrongARM processor sender
Initializing StrongARM processor receiver
Initializing GEZEL processor network.fdl
armzillasource: set processor sender
armzillasource: set address 2147483656
armzillasink: set processor receiver
armzillasink: set address 2147483656
Cycle: 1 channel = 0
Cycle: 30742 channel = 0
Cycle: 30743 channel = 80000000
Cycle: 40621 channel = 80000000
Cycle: 40622 channel = 1
Cycle: 42146 channel = 1
Cycle: 42147 channel = 80000002
Cycle: 43429 channel = 80000002
... etc
```

The use of the `armcosim` and `armzilla` cosimulators is discussed in Section 6.0 on page 54.

## A.3 8051 Cosimulator

The 8051 cosimulator is based on the instruction-set simulator from the Dalton project at UC Riverside (`http://www.cs.ucr.edu/~dalton/i8051/`). The instruction-set simulator itself is included in the source code, and contains a few small modifications to include the cosimulation interfaces.

The enable the 8051 in the GEZEL build, use the `--enable-gezel51` flag when configuring GEZEL. Compile the cosimulator as follows,

```
> ./configure --enable-gezel51

> make

> make install
```

If you have included any special CPPFLAGS are LDFLAGS in the standalone build (see Section A.1 on page 88), then those must be included here as well. For example:

```
> ./configure --enable-gezel51 \
             CPPFLAGS=-I/opt/gmp/include \
            LDFLAGS=-L/opt/gmp/lib
```

The 8051 programs for `gezel51` are provided in Intel Hex format. They can be created using the Small Devices C Compiler, available from `http://sdcc.sourceforge.net/`. Refer to that page for download and installation instructions of the sdcc compiler.

To test everything, use the example in `test/i8051/hello`:

```
> cd test/i8051/hello

>  make
> sdcc driver.c

> make sim
../../../build/bin/gezel51 -g hello.fdl -x driver.ihx
Initializing GEZEL processor hello.fdl
Initializing 8051 processor driver.ihx
P0      P1      P2      P3
0xFF    0xFF    0xFF    0xFF
0xFF    0x03    0xFF    0xFF
9590 Hello! You gave me 3/3
0x01    0x03    0xFF    0xFF
0x00    0x03    0xFF    0xFF
0x00    0x02    0xFF    0xFF
9734 Hello! You gave me 2/2
0x01    0x02    0xFF    0xFF
0x00    0x02    0xFF    0xFF
0x00    0x01    0xFF    0xFF
9878 Hello! You gave me 1/1
0x01    0x01    0xFF    0xFF
0x00    0x01    0xFF    0xFF
0x00    0x01    0xFF    0x55
```

The use of the 8051 cosimulator is discussed in Section 6.4 on page 63.

## A.4 SystemC Cosimulator

SystemC adds hardware-oriented constructs as a class library implemented in standard C++. Its use spans design and verification from concept to implementation in hardware and software. SystemC provides an interoperable modeling platform which enables the development and exchange of very fast system-level C++ models. It also provides a stable platform for development of system-level tools.

GEZEL blocks can be embedded in a SystemC simulation. SystemC is used as a simulation backbone, but can support modules described in GEZEL FSMD. This is convenient to add hardware 'scripting' to a particular environment. Each time the SystemC simulator starts, it can parse a new GEZEL description.

The cosimulation uses SystemC 2.0.1, available from `http://www.systemc.org` You need to install this package before creating the cosimulator. You can build it as follows

```
> tar zxfv systemc-2.0.1.tgz

> cd systemc-2.0.1

> configure

> make

> make install
```

The installation is done by default under Systemc-2.0.1. We will assume this location in the following.

The cosimulator in GEZEL is a C++ library with cosimulation interfaces. It is created as follows. We configure GEZEL with the `--enable-systemccosim` flag. You also need to indicate the location where the SystemC library can be found with the `--with-systemc-lib` configuration flag. The library path of SystemC is dependent on the host machine type. We assume a Linux machine here.

```
> cd gezel

> ./configure --enable-systemccosim \
--with-systemc-lib=/home/guest/systemc-2.0.1/lib-linux
```

If you happen to have the GMP library installed in a non-standard location, do not forget to include CPPFLAGS and LDFLAGS for that one as well. For example,

```
> ./configure --enable-systemccosim \
  --with-systemc-lib=/home/guest/systemc-2.0.1/lib-linux \
  CPPFLAGS='-I/opt/gmp/include' \
  LDFLAGS='-L/opt/gmp/lib'
```

Next, make and install the cosimulator in GEZEL. This will create a library `libgzl-sysc.a`

```
> make

> make install
```

The systemc cosimulation can be tested on the examples in `test/systemc`. Compile the program as for a normal SystemC program. The include path should contain both the SystemC include path as well as the GEZEL include path.

```
> g++ -g -O3 -Wall -c \
 -I/home/schaum/systemc-2.0.1/include \
 -I../../../devel/build/include/gezel \
 accum_sc.cxx -o accum_sc.o
```

After compilation, link with the SystemC library, the GEZEL library, the library with cosimulation interfaces and finally the gmp library.

```
> g++ -g accum_sc.o -L../../../devel/build/lib/ \
  -lgzlsysc -lfdl -lgzlsysc \
  -L/home/guest/systemc-2.0.1/lib-linux \
  -lsystemc -lgmp -o systemc_cosim
```

Note the link order for gzlsysc and fdl. They are cross-dependent and therefore `-lgzl-sysc` is provided twice in the link command.

The cosimulation can then be run as any other SystemC simulation

```
> ./systemc_cosim
SystemC 2.0.1 --- Sep 15 2003 14:48:27
```

```
Copyright (c) 1996-2002 by all Contributors
           ALL RIGHTS RESERVED
systemcsource: set variable var1
systemcsink: set variable var2
Sim starts
data_2 value is 0
data_2 value is 0
data_2 value is 1
data_2 value is 3
data_2 value is 6
etc ...
```

The creation of SystemC cosimulations is discussed in Section 7.0 on page 67.

## A.5 Reporting Problems

GEZEL is an open source environment, distributed under the GNU Lesser Public License. Basically, this gives you a free license to use GEZEL. LGPL also means that GEZEL comes with *no warranty, nor are we liable* for the consequences of your use of GEZEL. The LGPL license (in the file `COPYING`) gives you all the details.

Still, this does not mean we don't want to hear from you! The preferred way of feedback to GEZEL is through the GEZEL email list:

`http://groups.yahoo.com/group/gezel/`

If you have a very specific problem or question, you can also contact the developers. We appreciate your feedback a lot.

- Patrick Schaumont (`schaum@ee.ucla.edu`)
- Doris Ching (`dorisc@ee.ucla.edu`)

*The only way an EDA tool can improve is by interacting with users.*

# Appendix B: Reuse in the GEZEL Kernel

The GEZEL kernel is designed for flexibility, and enables extensions to the GEZEL language as well as to the back-end. A single data structure is used for simulation as well as for code generation.

## B.1 Design-environment reuse, design reuse, and abstraction

If one would look for the three virtues that enable to master increasingly-complex SoC design, then the shortlist would need to include: design-environment reuse, design reuse and design at higher levels of abstraction. Design-environment reuse means that a design environment can effortlessly adapt to new applications. Design reuse means that design artefacts have to be designed only once. Design at higher levels of abstraction means that such developments also require less painstakingly detail.

In recent years, C++-based design has become highly popular. Gupta has shown that the basic requirements for hardware modeling are all covered by C++ [Gupta 97]. C++ also deals efficiently with design-environment reuse, design reuse, and abstraction. There is very good support for them due to the object-oriented paradigm. For example, handshake protocols can be abstracted elegantly into objects [Schaumont 99]. Memory accesses can be abstracted into uniform indexed variables, even for non-uniform and distributed memories [Pasko 02].

As illustrated in Figure B.1a, a SystemC design environment uses C++ as a meta-language to express a design in terms of a class library. The design and the design environment (the design kernel) thus live in the same language space. The approach followed by GEZEL is to use a specialized design language as shown in Figure B.1b. A language separates the design environment from the design. From the flexibility point-of-view, clearly Figure B.1a is a more generic solution. However it requires a designer to master meta-language design, i.e. to develop a design in terms of the syntax and semantics of another language. In contrast, a dedicated language approach as shown in Figure B.1b shields a designer with a separate layer of syntax and semantics. Such a layer abstracts out the implementation details of the modeling environment. The downside of Figure B.1b is that a dedicated language restricts flexibility. The ideal environment would be one that can deal with flexibility in the design environment as shown in Figure B.1a, but that has the design robustness of Figure B.1b.

GEZEL targets this optimum. It has a modeling language for cycle-true hardware architectures. But it can be easily extended using library blocks that look like modeling primitives in their own right. These library blocks enable <Underline>design-environment reuse such as cosimulation interfaces.

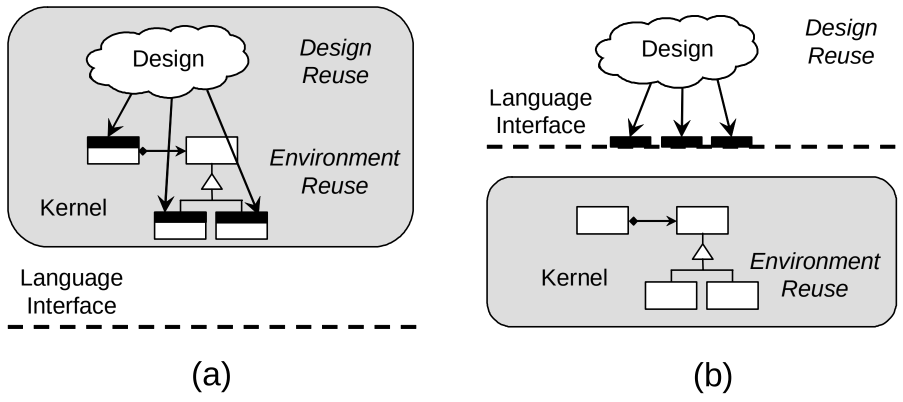


**FIGURE B.1. Reuse with (a) a C++ based approach and (b) GEZEL.**

## B.2 Review of the GEZEL language

Let's make a quick review of the features of the GEZEL language.

**Variables and Data Types.** There are two kinds of variables in GEZEL programs: registers and signals. Each of those variables can represent an unsigned or a two's-complement signed number of selectable precision. A register can transport a value to the next clock cycle, while a signal can only hold a value within a single clock cycle.

**Expressions.** Expressions are formed using operators on registers and signals. Almost all operators from the C programming language are supported, and a few new ones such as for bit-selection and bit-concatenation are added. The precision of expression results are determined by the operands as well as the operators, and GEZEL operators do not loose precision on intermediate results. Precision is controlled through the use of the cast operator, the assignment operator and the wordlengths of registers and signals.

**Signal Flowgraphs.** Expressions are grouped together into signal flowgraphs to represent a single clock cycle of register-transfer behavior. All expressions in a signal flowgraph execute concurrently. The order of evaluation is defined by data precedences and the semantics of registers and signals. Thus, the lexical order of expressions within a signal flowgraph is irrelevant. A signal flowgraph may or may not execute at a particular clock cycle depending on the control schedule. When it does, it is said to be active in that clock cycle.

**Data Paths.** Several signal flowgraphs can be grouped together to form a datapath. A datapath also defines an interface with inputs and outputs. These have the same semantics as signals, and can be used as expression operands and targets in signal flowgraphs. Registers and signals must always be created within the boundaries of a datapath. A datapath can include as many signal flowgraphs as needed. At any particular clock cycle an arbitrary combination of signal flowgraphs can execute as long as the semantic requirements for GEZEL programs defined below are not violated.

**Controllers.** A controller defines a schedule for signal flowgraphs in a datapath. The most general controller is the finite state machine. A finite state machine defines an initial state, other states, and state transitions. State transitions can be conditionally dependent on the value of registers in a datapath. Instructions selected by the controller each correspond to the execution of one or more signal flowgraphs in the datapath. Besides finite state machines, two other controllers exist: hard-wired controllers which have only a single instruction, and sequencers which have a cyclic sequence of instructions.

**Library Blocks.** Library blocks are prebuilt datapaths, with a behavior that is defined within the GEZEL kernel. Library blocks are used to model hardware-software interfaces, ram blocks, functional user models, and so on. Library blocks can be added by the user but require re-compilation of the GEZEL kernel. At the interface, they behave the same as a datapath and they have to obey the same semantic rules.

**System Integration.** Several datapaths, controllers and library blocks can be interconnected in a system, which is the top-level module for GEZEL.

**Structural Hierarchy.** Datapaths and library blocks can be enclosed within other datapaths. This nesting can be done over multiple levels.

**Module Instantiation.** An existing GEZEL datapath or library block can copied using the cloning operator. Datapaths and library blocks enclosed at lower hierarchical levels will be copied as well.

**Directives.** Directives enhance the behavior of a GEZEL simulation without changing the behavior of the design itself. Examples are printing messages to the screen and tracing the value of a variable in a file.

**Semantic Requirements.** A GEZEL description that is syntactically correct has to obey four requirements to be also semantically correct. They guarantee that the GEZEL specification leaves no ambiguity or unknown values. The four properties are the following ones.

- During any clock cycle, all datapath outputs must be defined as the left-hand side of an expression in an active signal flow graph. A signal flowgraph is said to be active in a particular clock cycle when a controller selects it for execution in that clock cycle.
- During any clock cycle, no combinatorial loop between signals can exist. This happens when there is a circular dependence on signal values, i.e. signal a is used to define signal b, and signal b is used to define signal a. All data dependencies eventually must end in a datapath register or a constant value. This condition also holds for loops across multiple datapaths.
- If an expression consumes the value of a signal in an active signal flow graph, then that signal must also appear at the left-hand side of an assignment expression in an active signal flow graph, or it must be a connected datapath input. There can be no ‘unknowns’ in the simulation.
- Neither registers, nor signals or datapath outputs can be assigned more than once in the same clock cycle.

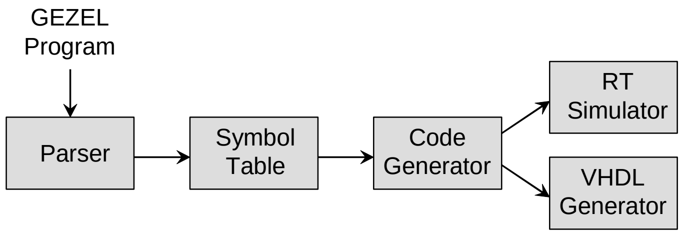


**FIGURE B.2. Architecture of the GEZEL kernel.**

## B.3 Architecture of the GEZEL kernel

Figure B.2 illustrates the architecture of the GEZEL kernel. GEZEL programs are parsed and stored in a generic intermediate format using a symbol table. The symbol table is accessed by a code generator, and converted into either an RT-level simulator or else a VHDL syntax translator. The two architectures — RT-simulator and VHDL syntax translator — are essentially variations on the same data structure.

## B.4 The symbol table

The symbol table sits at the heart of the GEZEL kernel. The symbol table is created by the parser. It is a structured representation of the GEZEL design. The symbols of a GEZEL program are tokens created out of the program text. Consider for example the first line of a datapath definition in GEZEL.

```
dp mydatapath(in myinput : ns(1))
```

The GEZEL parser will create three symbols for this piece of program text:

- A ‘datapath’ symbol, holding the string ‘mydatapath’.
- A ‘datapath-input’ symbol, holding the string ‘myinput’.
- A ‘type’ symbol, holding type information for a 1-bit unsigned number.

In addition the GEZEL parser will store context information for these symbols. The context of the ‘type’ symbol relates to the ‘datapath-input’ symbol, and the context of the ‘datapath-input’ symbol relates to the ‘datapath’ symbol. Once the GEZEL symbol table is created, it serves as a database for the code generators.

The GEZEL symbol table is a class hierarchy, as illustrated in the class diagram of Figure B.3. The notation follows that of the Unified Modeling Language standard for class diagrams [OMG 04]. The rectangles indicate classes and the edges indicate relations. A `symboltable` holds a number of generic `symbols`. The ‘N’ on the edge indicates there are many (N) `symbols` per `symboltable`. The black diamond indicates `symbols` are strongly aggregated by the `symboltable`: if the `symboltable` is destroyed, the `symbols` are thrown away as well. Each `symbol` has a unique identifier (symid) as well as a context. A context holds the unique identifier of a parent symbol. In the example above, the context of the ‘type’ symbol is the identifier of the ‘datapath-input’ symbol. The actual content of a sym-

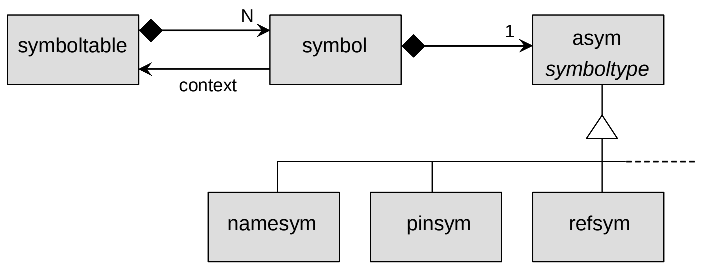


**FIGURE B.3. Class diagram of the symbol table hierarchy.**

bol varies according to the kind of symbol. For example, the symbol information of a constant integer is different than that of an addition operator symbol.

Consider the following GEZEL program snippet, a two-instruction datapath.

```
dp updown(out a : ns(3)) {
  reg c : ns(7);
  sfg up { c = c + 1; a = c; }
  sfg dn { c = c - 1; a = c; }
}
```

The symbol table that is created for this datapath is shown in the next table.

| Id | Symbol Type | Symbol kind | Contents | context |
|---|---|---|---|---|
| 0 | namesym | datapath | updown | -1 |
| 1 | namesym | dpoutput | a | 0 |
| 2 | sigtypesym | sigtype | 3-bit unsigned | 1 |
| 3 | namesym | reg | c | 0 |
| 4 | sigtypesym | sigtype | 7-bit unsigned | 3 |
| 5 | namesym | sfg | up | 0 |
| 6 | namesym | const_op | "1" | 5 |
| 7 | binaryopsym | add_op | (L=3, R=6) | 5 |
| 8 | binaryopsym | assign_op | (L=3, R=7) | 5 |
| 9 | binaryopsyn | assign_op | (L=1, R=3) | 5 |
| 10 | namesym | sfg | dn | 0 |
| 11 | namesym | const_op | "1" | 10 |
| 12 | binaryopsym | sub_op | (L=3, R=11) | 10 |
| 13 | binaryopsym | assign_op | (L=3, R=12) | 10 |
| 14 | binaryopsym | assign_op | (L=1, R=3) | 10 |

The table shows three different symbol types (`namesym`, `sigtypesym`, `binaryopsym`). Each symbol type can be one of multiple kinds. For example, a datapath name and a datapath input are both characterized by a string. They have the same symbol type `namesym`, but they are symbols of different kinds (`datapath` and `dpoutput`). GEZEL has 16 symbol types and 66 different symbol kinds. Each symbol has a unique identifier and a context. A

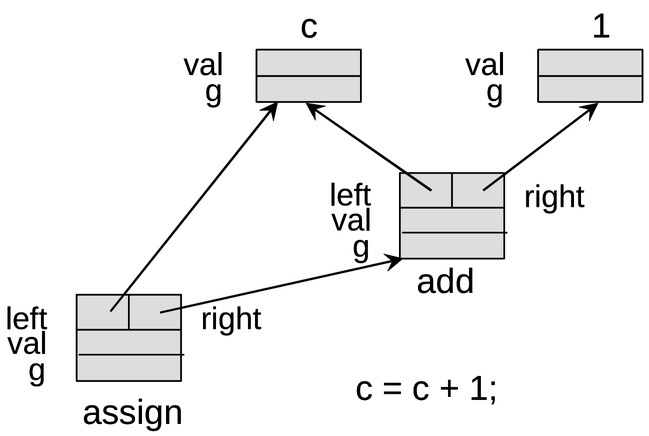


**FIGURE B.4. Data structure to simulate an expression.**

context holds the identifier of a parent symbol. For example, symbol 7 represents an addition contained within symbol 5 (sfg `up`). The symbol identifier is also used to record data precedences. The arguments of symbol 7 are symbol 3 (`c`) and symbol 6 (constant `1`).

The symbol table's primary function is to serve as a database for the code generator. A second function of the symbol table is to support cloning operations in GEZEL. A cloning operation creates an exact copy of an existing datapath or library block. For example, consider an AND gate as follows.

```
dp andgate(in a, b : ns(1); out q : ns(1)) {
  sfg run {
    q = a & b;
  }
}
hardwired h_andgate(andgate) {run; }
```

Then GEZEL allows to create a copy of this block by writing

```
dp andgate2 : andgate
```

Cloning is a quick-hand notation to create an additional instance. Cloning is implemented by copying symbol table entries. The context and other references to symbol identifiers must be adjusted as well. Because symbol identifiers are generated out of a unique, monotone sequence, these references can be cloned by adding a constant offset, equal to the distance between the old top-level symbol identifier (`negate` in the example) and the new top-level symbol identifier (`andgate2` in the example).

## B.5 The code generation interface

The code generation process in GEZEL uses a single symbol table to create the RT simulator as well as the VHDL code. A vital aspect in the interface to the code generators is to support flexibility for the GEZEL language as well as for the code generators. While the current code-generation targets are RT simulation and VHDL, other future targets cannot be excluded (Verilog code, C code, dotty graph format, and so on). At the same time, the

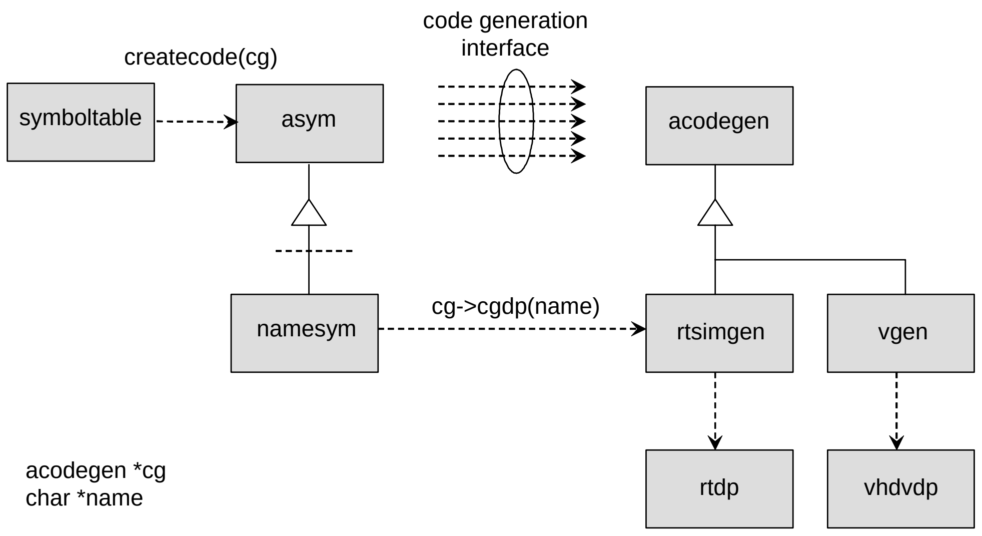


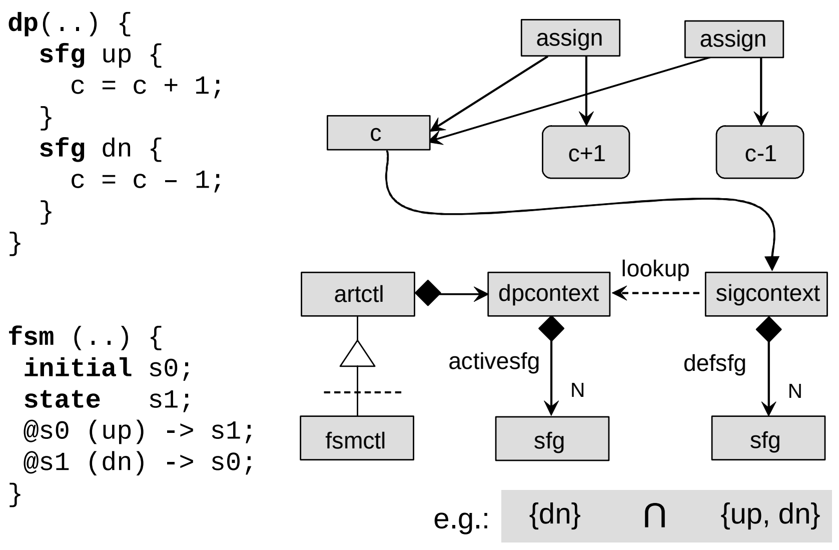
**FIGURE B.5. The code generation interface of GEZEL.**

GEZEL language itself cannot be fixed as improvements are still being suggested by users on a regular basis.

Thus, it’s not a good idea to add an RT-code generation method and a VHDL-code generation method to each symbol class in GEZEL. Including a new code generator in such a software architecture would be a major headache because the code generation is distributed over the ensemble of classes (and source files) that represent symbols. On the other hand, attempting to centralize code generation methods with a separate RT-code generation class and a VHDL-code generation class is not a good solution either, because this distributes the code generation interface for each symbol over multiple classes. Adding new symbol types in such a software architecture is complicated and error-prone.

My solution is shown in Figure B.5. The key is to make use of two class hierarchies, one for symbols and one for code generators. The left of the figure shows the symbol class hierarchy discussed above. The right of the figure shows the code generation class hierarchy. The top-level class, `acodegen`, is an abstract code generator. This class understands all symbol types and kinds of GEZEL, but is not fixed to any particular target. Two derived classes, `rtsimgen` and `vgen`, are code generators for RT simulation and VHDL syntax translation respectively. They understand the internals of their own target, and can create the objects required for RT simulation or VHDL syntax generation.

Figure B.5 also summarizes the activities for code generation of a datapath symbol. The code generation starts at the symbol table class, and requests code generation for the datapath symbol using a particular code generation target `cg`. The `asym` symbols in a symbol table have an abstract code generation interface, implemented in derived symbols. A datapath symbol is a string (name). The code generation method for a datapath is implemented in `namesym`, and this method provides the name of the datapath to the code generator target `cg`. The code generator can be either an RT-simulation target (`rtsimgen`) or else a VHDL target (`vgen`). The exact behavior of `cg->cgdp(name)` will depend on the code generation



```
dp(..) {
  sfg up {
    c = c + 1;
  }
  sfg dn {
    c = c – 1;
  }
}
```

```
fsm (..) {
 initial s0;
 state   s1;
 @s0 (up) -> s1;
 @s1 (dn) -> s0;
}
```

**FIGURE B.6. Runtime resolution of expression trees.**

target. Both targets will create objects that are required for RT simulation (`rtdp`) or VHDL syntax translation (`vhdvdp`) of a datapath respectively.

A key advantage of this method lies in the abstract code generation interface. This interface consists of a set of virtual (non-implemented) methods, one for each symbol kind. The interface is defined at the level of an abstract symbol (asym) in a symbol-independent manner, and at the level of an abstract code generation (acodegen) in a code-generator independent manner. A new type or kind of symbol will result only in an incremental change to the code generation interface. At the same time, adding a new code generator can be done without disturbing any existing functionality in either the symbol table or else the existing code generators.

## 8.5 The RT simulator

The RT simulator is created using code generation out of the symbol table. The basic strategy is to convert expression trees in GEZEL directly into simulation objects. An example of the object structure for a simple expression is shown in Figure B.4. The addition and assignment operations are created as binary-operator objects that hold a pointer to their operands. Simulation of this expression follows the data precedences of the tree. As discussed before in Section 7.5. on page 127, this evaluation is done in a demand-driven fashion, starting at the bottom of the three (the ‘or’ operation) and working towards the leaves.

**Expression Selection.** How to select the proper expressions for assignments? The issue to address is illustrated in Figure B.6: the same variable can be assigned differently in different clock cycles, because different signal flow graphs can be active in each clock cycle. Signal flowgraphs are represented with `sfg` objects. I solved this by cross-matching two objects against each other at runtime: a `dpcontext` object, which maintains the list of all `sfg` active in the current clock cycle (`activesfg`), and a `sigcontext` object, which maintains the list of all `sfg` that hold a definition for a particular variable (`defsfg`). The `dpcon-`

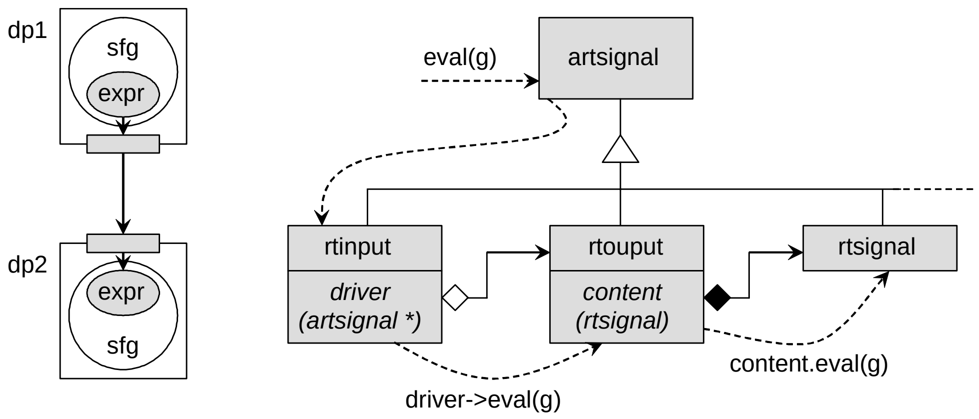


**FIGURE B.7. Resolution of structural hierarchy.**

`text` is maintained by a controller that knows the list of `sfg` to execute each clock cycle. The `sigcontext` on the other hand is a property of a variable. When a definition for a particular variable is required, the cross-section of the lists of `dpcontext` and `sigcontext` must return exactly one `sfg`. If it is zero, then a 'signal undefined' error is thrown. If i+t is more than one, then a 'multiple assignment' error is thrown.

**Detecting Loops.** Another semantic error that is detected at runtime is a combinatorial loop. Each time the demand-driven evaluation of a signal is postponed, a boolean field in that signal object called `loop` is set to true. Whenever the demand-driven evaluation of a signal is attempted which has this loop flag true, a combinatorial loop is detected and a run-time error message is issued. By simple backtracking of the outstanding demands in the evaluation, all signals involved in the combinatorial loop can be detected.

**Dealing with hierarchy.** The strategy for expression selection has to deal with datapath hierarchy as well: signal flowgraphs are contained within datapaths, that may be included within other datapaths. Expression trees in signal flowgraphs thus end at the inputs of a datapath. Datapaths are interconnected in the GEZEL system description, so a datapath input will be connected to the output of another datapath. Hardware simulators usually resolve the hierarchy at the start of a simulation, meaning that they strip out these boundaries. In the case of GEZEL, I resolve the hierarchy at runtime because this results in a faster parsing and simulator construction process. Figure B.7 illustrates that this requires only minimal overhead. All signal objects in GEZEL derive from a single base class `artsignal`. This class has a virtual method `eval()` to evaluate the current value of the signal. A datapath input (`rtinput`) does not really contain a value by itself, but rather a pointer to a corresponding output (`rtouput`) that connects to this input. This signal is called a driver. A datapath output itself is a simple shell around a standard GEZEL signal (`rtsignal`). When the simulator evaluates the input of a datapath, it will be forwarded to the expression tree of the output connected to this input. This kind of hierarchy resolution works also for encapsulated datapaths. The runtime overhead of maintaining hierarchy is a pointer dereferencing per level of hierarchy.

**Design-environment reuse.** The RT simulator supports design-environment reuse in three areas, each time through the concept of C++ inheritance. Within a datapath, the base class `artsignal` sits at the apex of a hierarchy of 34 classes that implement evaluation of expressions, operators and signals. Second, datapath controller descriptions are constructed on top of the base class `artctl`. Finite state machines and sequencers are build using 9 classes. Thirdly, at the GEZEL system level the base class `aipblock` allows users to add new library blocks to the kernel — these are described in slightly more detail below.

## B.6 Library blocks

Practical experience of external users with GEZEL has shown that library blocks are a key concept for the rapid-prototyping of complex systems. GEZEL library blocks provide a standard interface, both syntactically as well as semantically. They allow the addition of reusable functionality. In addition they are developed in C++ and allow a good amount of abstraction. This makes them also faster than equivalent functionality described at the GEZEL level.

Originally the library block modeling facility was added as a catch-all for functionality that could not be modeled in the evolving GEZEL language. Over time however, the library block mechanism has become more useful than I had intended at first. Here are a few examples of library blocks that were created in the past three years.

- Background memories (RAM and ROM), specialized storage architectures, shared memories, and multi-port memories.
- Cosimulation interfaces to many different instruction-set simulators: ARM, SH3, LEON2-SPARC, 8051. A cosimulation interface to SystemC.
- High-level coprocessor functionality for what-if-simulations in hardware-software codesign. Coprocessor functions were build for crypto, image-processing, coding, and network processing.
- Network-on-chip models in the form of 1D-router and 2D-router blocks.
- Simulation support functions to create debug traces and graphical user interfaces in a GEZEL system simulation.

GEZEL library blocks are high-level models written in C++ that are systematically integrated into the GEZEL kernel. Their execution-semantics are compatible with the cycle simulation in GEZEL.

From a high level perspective, a library block is a class with a method `run()` that is called every clock cycle, and that works with a set of values that are present at the input and output ports of the block. The GEZEL simulator will make sure that the `run()` is only called when all input port values are current and stable. The simulator assumes that an output port can immediately change value because of a change at an input port; in other words, that there can be (but not must be) a combinatorial path from input to output.

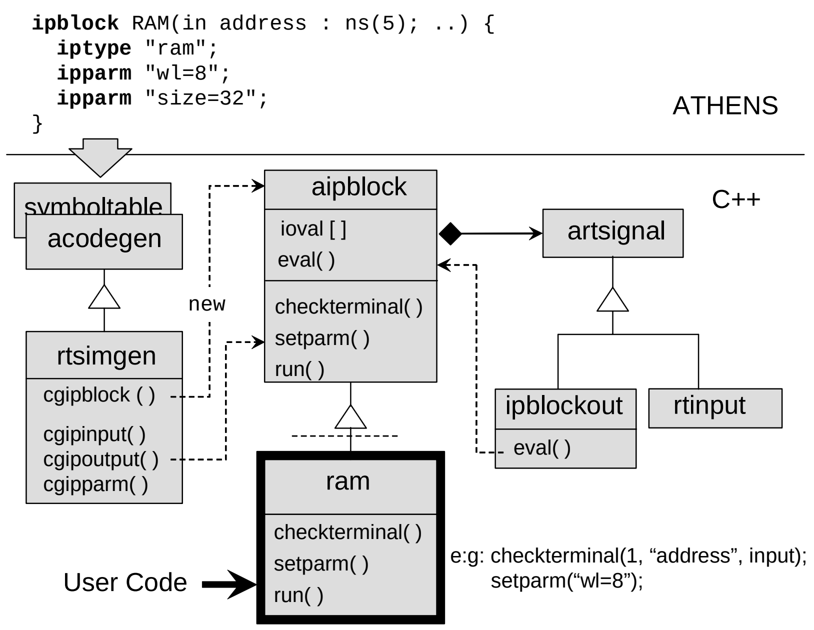


**FIGURE B.8. User-defined functions in GEZEL.**

Consider the interaction of a user-defined library block with the simulation kernel. Here is a library block for a RAM, described in GEZEL as:

```
ipblock RAM(in address : ns(5);
            in wr,rd   : ns(1);
            in idata   : ns(8);
            out odata  : ns(8)) {
  iptype “ram”;
  ipparm “wl=8”;
  ipparm “size=32”;
}
```

This defines a library block of type ‘ram’. The block has 5 ports, defined in a similar way as the ports of a datapath. For a library block, the order and name of ports are considered important as they establish a link between GEZEL code and C++ implementation. Two additional parameters are given as well, ‘`wl=8`’ and ‘`size=32`’. These are user-defined parameters that enable parameterized library block development. In this case, they result in an 8-bit ram with 32 positions.

The activities of library block instantiation can be followed using the class diagram in Figure B.8. The base class for user-defined library blocks is called `aipblock`. The code generator will create an instance of a user-defined library block through the `cgipblock`( ) method, the code generation method for an `ipblock` symbol. If a library block of the desired type is not known by the GEZEL kernel, an error message is generated.

The code generator also creates a series of input and output ports, and pass the parameter strings. The ports are verified for name and direction using a user-defined `checktermi-nal()` method. For the example above, `checkterminal()` and `setparm()` are called for example as

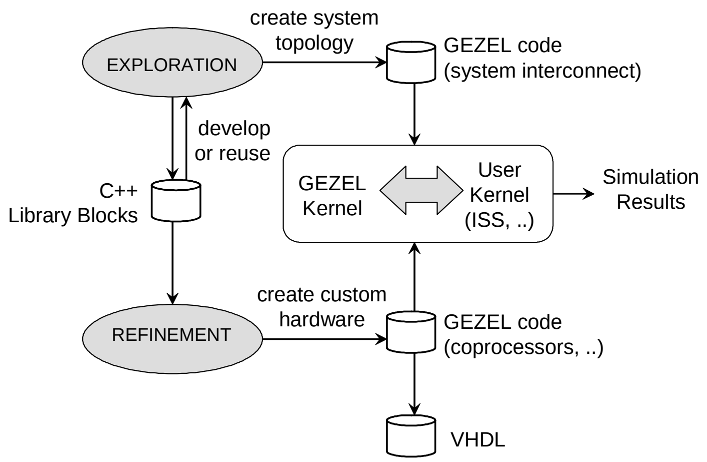


**FIGURE B.9. Design flow with GEZEL library blocks.**

```
checkterminal(1, “address”, input);
checkterminal(2, “wr”, input);
setparm(“wl=8”);
```

Input and output ports are stored in an array `ioval[]`. The user-defined `run()` method uses this array to implement the library block’s function. Inputs of a library block are of type `rtinput`, while outputs are of type `ipblockout`. This special output class is used to enforce the demand-driven evaluation scheme on user-defined blocks; when the value of a library block output is needed and the library block itself has not yet executed in the current clock cycle, it will execute first.

In my experience GEZEL library blocks are a very important element in the design-environment reuse of the GEZEL kernel. In fact, it enables an explore-and-refine design flow as illustrated in Figure B.9. In the explore phase, a user configures a system topology using GEZEL library blocks. The library blocks are reused from other projects or else custom developed (usually it’s a combination of these). Once the desired system architecture is identified, a refinement phase starts in which hardware views are developed in GEZEL for those library blocks that need it. Eventually these hardware views can be translated into synthesizable VHDL. The strong point of this approach is that the exploration-simulator is already a cycle-true model and thus yields accurate simulation results. The scripting approach of GEZEL also makes the simulation interactive, and encourages a designer to explore alternatives.

## B.7 Plan to throw one away

A final word on design-environment reuse in GEZEL. The idea at the core of this chapter is that the GEZEL kernel needs to be extensible. As Brooks observed already long ago, most of the code will eventually be thrown away anyhow [Brooks 95]. Therefore it is a safe strategy to concentrate on extension mechanisms rather than on actual implementations.

The perfect GEZEL kernel would be, I believe, a collection of abstract base classes that will accommodate to a broad range of system-on-chip design problems: simulation and validation, performance evaluation, refinement, implementation, virtual prototyping, and so on. The user (designer) then participates in the GEZEL development process, co-developing a design with extensions to the GEZEL kernel. Indeed, the complexity of system-on-chip nowadays is such that design problems no longer can be captured in a closed and fixed design environment. The expertise of a designer is needed to help define a supporting CAD environment, and to personalize it to specific design requirements. However, the basic infrastructure of this CAD environment must be available; one cannot expect designers to turn into CAD people or C++ developers right away.

The current structure of the GEZEL kernel merely reflects a search to such an extensible yet supportive CAD environment. So far this has resulted in at least one good thing. It has become much easier to involve users and engage new developers in the design of GEZEL. The VHDL code generator was developed by Ching; she also developed a complete network-on-chip design kit using library blocks; Steensgaard-Madsen and Lorentzen at Danmark University have contributed library blocks to support the Mic-1 micro-architecture; Villa and Monchiero at Politecnico Milano have developed a multi-processor architecture based on the library block concept.

## B.8 References

[Gupta 97] R. Gupta, and S. Liao. “Using a Programming Language for Digital System Design.” IEEE Design and Test of Computers, 14(2):72—80, April-June 1997.

[OMG 04] Object Management Group. “Unified Modelling Language;” <http://www.uml.org>.

[Brooks 95] F.P. Brooks, Jr. “The Mythical Man-Month: Essays on Software Engineering.” Twentieth Anniversary Edition, Reading, MA: Addison-Wesley, 1995.

[Schaumont 99] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, and I. Bolsens. “Hardware reuse at the behavioral level.” Design Automation Conference (DAC 1999), 784-789, New Orleans, CA, USA, June 1999.

[Pasko 02] R. Pasko, S. Vernalde, and P. Schaumont. “Techniques to evolve a C++ based system design language.” Design, Automation and Test in Europe Conference and Exhibition (DATE 2002), 302—309, March 2002.

# Appendix C: References

## C.1 Publications on GEZEL

[Verbauwhede 04] I. Verbauwhede, P. Schaumont, *The Happy Marriage of Architecture And Application in next-generation Reconfigurable Systems*, ACM Computing Frontiers 2004, April 2004.

[Ching 04] D. Ching, P. Schaumont, I. Verbauwhede, *Integrated Modeling and Generation of A Reconfigurable Network-On-Chip*, 2004 Reconfigurable Architectures Workshop (RAW 2004), April 2004.

[Schaumont 04a] P. Schaumont, I. Verbauwhede, *Interactive Cosimulation with Partial Evaluation*, 2004 Design Automation and Test in Europe (DATE 2004), Februari 2004.

[Schaumont 04b] P. Schaumont, I. Verbauwhede, *ThumbPod puts security under your Thumb*, Xilinx Xcell Online, Winter 2003.[PDF]

[Schaumont 03] P. Schaumont, I. Verbauwhede, *Domain-specific Co-design for Embedded Security*, IEEE Computer, April 2003.

[Schaumont 02] P. Schaumont, I. Verbauwhede, *Domain-specific tools and methods for application in security processor design*, Kluwer Journal for Design Automation of Embedded Systems, pp. 365-383, November 2002.

## C.2 Publications on SimIT-ARM (ARM Cosimulators)

[Qin 03a] W. Qin, S. Malik. *Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation*, Proceedings of 2003 Design Automation and Test in Europe Conference (DATE 03), Mar, 2003, pp.556-561.

[Qin 03b] W. Qin, S. Malik. *Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits*, Proceedings of the 40th Design Automation Conference (DAC 03), June 2003, pp. 764-769.