GEZEL v2 Simulator User Manual

(Version August 1, 2005)

Copyright (c) 2004-2005
Patrick Schaumont (pschaumont@gmail.com)
Doris Ching (doris.kc@gmail.com)

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

Table of Contents

Listings

Roadmap to the v2 Simulator 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. 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,* summarizes 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, *Cosimulating GEZEL with Instruction Set Simulators,* explains how GEZEL is used in cosimulation.
- Section 6.0, *Cosimulating GEZEL with SystemC,* discusses the integration of GEZEL into a SystemC simulation.
- Section 7.0, *Cosimulating GEZEL with JAVA,* gives an overview of existing GEZEL library blocks (such as RAM cells), and also explains how you can create your own.
- Section 8.0, *GEZEL Library Blocks,* explains how you can add your own Library blocks to GEZEL, and how to convert those into dynamically-linked libraries that can be linked into the GEZEL simulator.
- 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, *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 familiar 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. 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.

Authors

Patrick Schaumont, UCLA

Doris Ching, UCLA

Herwin Chan, UCLA

Jorgen Steensgaard-Madsen, DTU

Andreas Vad-Lorentzen, DTU

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.

Alessandro Traficante, Politecnico di Milano

Sara Bocchio, ST

David Hwang, UCLA

Bocheng Lai, UCLA

Per Larsen, DTU

Jan Madsen, DTU

Bjarne Mathiesen, DTU

Yusuke Matsuoka, Renesas Technology Corp

Wei Qin, Boston University

Kazuo Sakiyama, KUL

Peter Verner Bojsen Sørensen, DTU

Students of the Spring 2003 EE201A class at UCLA

Students of the Spring 2005 02130 class at 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.
- 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 and Java.
- GEZEL can be customized with user-supplied custom library blocks in C++. Those blocks can be provided as dynamic libraries, those providing an excellent basis for exchange of intellectual-property simulation models.
- A design in the GEZEL language can be automatically translated to synthesizable code. In addition, extra support for stimuli capture is available so that GEZEL simulations can be 'replayed' on the hardware 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.

A typical application of GEZEL would be the design of a coprocessor shown in Figure 1.1. In this application, one writes GEZEL code for the hardware coprocessor, and evaluates the performance of the design in the intended target architecture using cosimulation. The simulation is cycle-true but, because of the cosimulation technology, many times faster than a similar system model in (V)HDL.

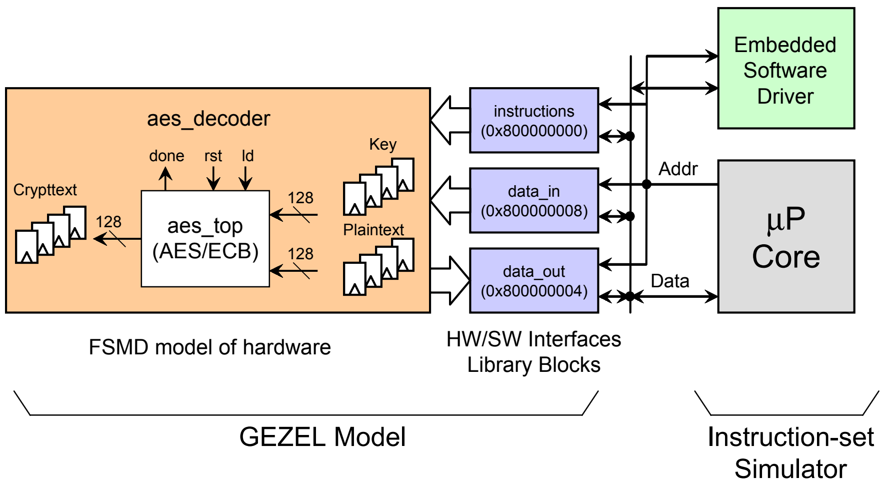


FIGURE 1.1. A GEZEL_based coprocessor simulation.

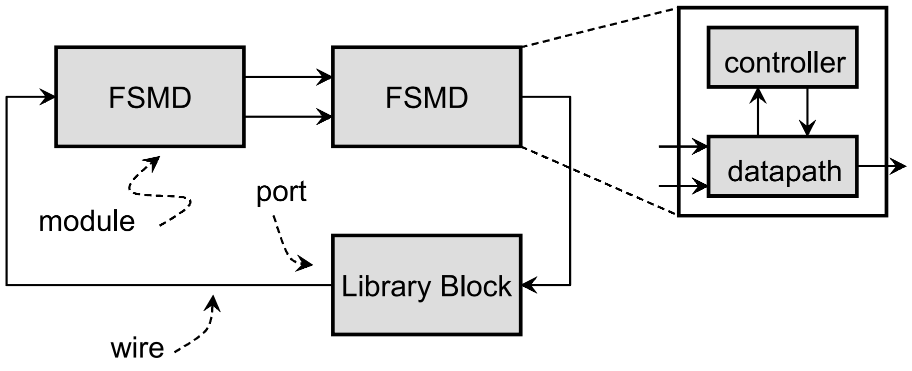


FIGURE 1.2. A sample GEZEL model

GEZEL has also been used in multiprocessor simulations to connect several heterogeneous cores with a network-on-chip. VHDL code generated out of GEZEL has been mapped onto FPGA as well as onto ASIC. The publications list in Section B on page 85 enumerates several papers that describe applications where GEZEL was used.

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

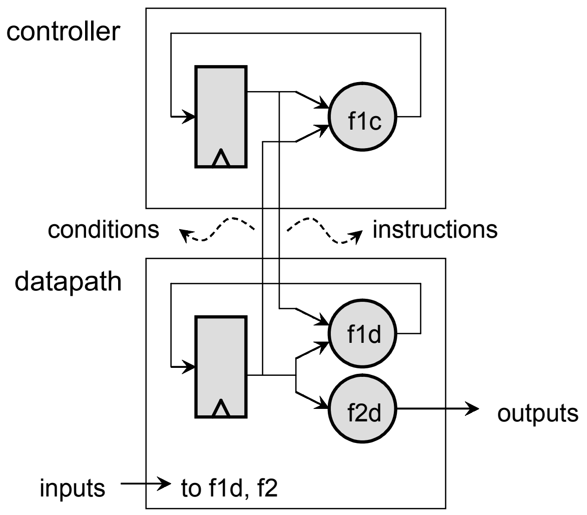


FIGURE 1.3. The GEZEL FSMD Model consists of two cross-coupled finite state machines.

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 can be found in 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 Algorithm 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 20. The next subsection presents a small example on the mapping of an algorithm into the FSMD model. The GEZEL syntax is introduced as well.

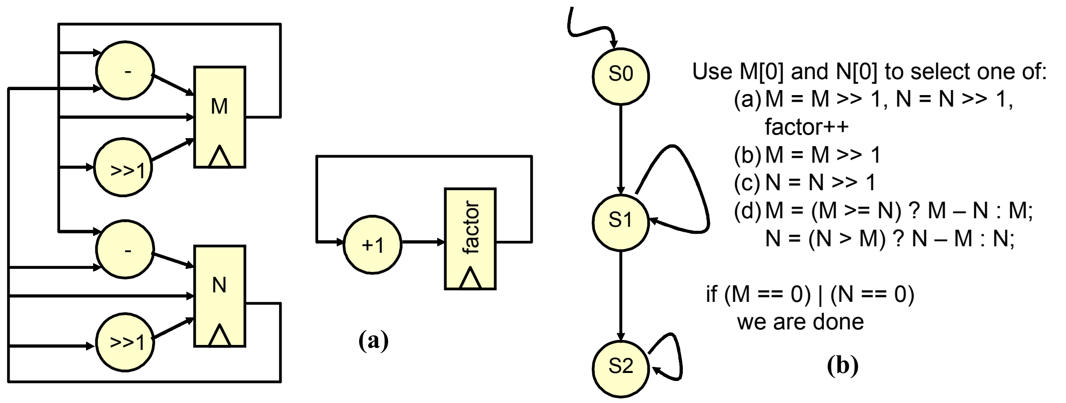

-
M
>>1
-
N
>>1
+1
factor
(a)
S0
S1
S2
Use M[0] and N[0] to select one of:
(a) M = M >> 1, N = N >> 1, factor++
(b) M = M >> 1
(c) N = N >> 1
(d) M = (M >= N) ? M – N : M; N = (N > M) ? N – M : N;
if (M == 0) | (N == 0) we are done
(b)

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(16);
6.
7.    sfg init    { m = m_in; n = n_in; factor = 0; done = 0;
8.                 $display("cycle=", $cycle," m=",m_in," n=", n_in);}
9.    sfg flags   { done = ((m == 0) | (n == 0)); }
10.   sfg shiftm  { m = m >> 1; }
11.   sfg shiftn  { n = n >> 1; }
12.   sfg reduce  { m = (m >= n) ? m - n : m;
13.                 n = (n >  m) ? n - m : n; }
14.   sfg shiftf  { factor = factor + 1; }
15.   sfg outidle { gcd = 0; }
16.   sfg complete{ gcd = ((m > n) ? m : n) << factor;
17.                 $display("cycle=", $cycle, " gcd=", gcd); }
18. }
19.
20. fsm euclid_ctl(euclid) {
21.   initial s0;
22.   state s1, s2;
23.
24.   @s0 (init, outidle) -> s1;
25.   @s1 if (done)              then (complete)                 -> s2;
26.       else if ( m[0] &  n[0]) then (reduce, outidle, flags)   -> s1;
27.       else if ( m[0] & ~n[0]) then (shiftn, outidle, flags)   -> s1;
28.       else if (~m[0] &  n[0]) then (shiftm, outidle, flags)   -> s1;
29.                               else (shiftn, shiftm,
30.                                     shiftf, outidle, flags)   -> s1;
31.   @s2 (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. dp euclid_sys {
43.    sig m, n, gcd : ns(16);
44.    use euclid(m, n, gcd);
45.    use test_euclid(m, n);
46. }
47.
48. system S {
49.   euclid_sys;
50. }
```

The test-bench for the GCD processor is shown in lines 34—50. 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=0 m=912 n=28e
cycle=22 gcd=6
```

The simulator reports that the GCD of the two test values is 6, and that this value is obtained at cycle 22 of the simulation. This line is printed using a simulation directive as shown on line 17 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 12 of Listing 1 contains `'sff reduce'` then the following error message appears:

```
> fdlsim euclid.fdl 25

*** (line 13) Syntax Error

(9)      sfg flags   { done = ((m == 0) | (n == 0));  }
(10)      sfg shiftm  { m = m >> 1; }
(11)      sfg shiftn  { n = n >> 1; }
(12) >>>  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
```

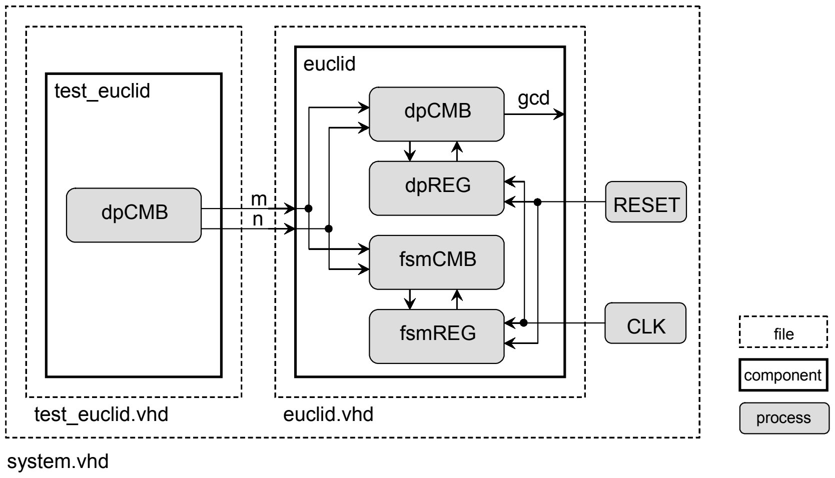


FIGURE 1.5. Component Hierarchy and Process of the generated VHDL code.

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*.

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

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

The keyword `always` indicates that the group of expressions following it will execute each clock cycle. A signal flow graph can hold an arbitrary number of expressions. All expressions within a single signal flow graph are concurrent within one clock cycle. The order in which expressions are evaluated is independent of the order in which they appear in the GEZEL program (i.e., it is independent of their lexical order). 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 within a signal flow graph becomes irrelevant. For example, if you would write:

```
always {
  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, this signal flow graph behaves exactly the same as the one we described before that.

2.4 Named signal flow graphs

Besides the unnamed `always` signal flow graph, you can create signal flow graphs with a name using the `sfg` keyword. For example, the previous signal flow graph could be written as:

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

The difference between a named signal flowgraph (`sfg`) and the unnamed `always` is that the former does *not* automatically execute each clock cycle. GEZEL will allow you to create a controller that schedules the named signal flowgraph. Controller design will be discussed in Section 3.0 on page 20.

2.5 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 named signal flow graphs with a list of input and output signals. A signal flow graph can be thought of as an instruction for that datapath. A datapath can have only a single `always` signal flow graph, but it can have multiple named signal flow graphs.

A datapath is the smallest GEZEL unit that can be simulated. So, subsequent examples will be fully self-contained programs rather than snippets.

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

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

```
1. dp counter(out value : ns(2)) {
2.   reg c : ns(2);
3.   always {
4.     value = c;
5.     c = c + 1;
6.     $display("Cycle ", $cycle, ": counter = ", value);
7.   }
8. }
9.
10. system S {
11.   counter;
12. }
```

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.

Finally, on lines 10—11, the toplevel of the system is expressed. A GEZEL file must always have a `system` statement.

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 named 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.
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

```
// WRONG: output v is not always defined
dp bad1(out v : ns(1)) {
  always {}
}

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

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

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

2.6 Structural Hierarchy

Datapaths can be included inside of other datapaths, thus implementing structural hierarchy. 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

```
dp andgate(in a, b : ns(1); out q : ns(1)) {
  always {
    q = a & b;
  }
}

dp andgate2 : andgate
dp andgate3 : andgate

dp fourinputand(in a, b, c, d : ns(1); out q : ns(1)) {
```

```
11.   sig s1, s2 : ns(1);
12.   use andgate ( a,  b, s1);
13.   use andgate2( c,  d, s2);
14.   use andgate3(s1, s2,  q);
15.   always {
16.      $display(a," ", b, " ", c, " ", d, " -> ", q);
17.   }
18. }
19.
20. dp tst(out a, b, c, d : ns(1)) {
21.   reg n : ns(4);
22.   always {
23.     n = n + 1;
24.     a = n[0]; b = n[1]; c = n[2]; d = n[3];
25.   }
26. }
27.
28. dp sysandgate {
29.   sig a, b, c, d, q : ns(1);
30.   use tst(a, b, c, d);
31.   use fourinputand(a, b, c, d, q);
32. }
33.
34. system S {
35.   sysandgate;
36. }
```

Lines 10—18 define 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 20—26 define a testbench that enumerates all 4-bit input patterns by decomposing the bits of a counter. Finally, lines 28—32 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
```

2.7 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)) {
  always {
    q = a & b;
  }
}

dp andgate2 : andgate
dp andgate3 : andgate
```

Cloning creates an identical but independent copy. If the parent datapath includes a register, then the cloned datapath will contain its' own register.

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.

Regarding the `system` statement.

Before GEZEL 1.7, the `system` statement was used to express the toplevel interconnect. Starting with GEZEL 1.7, this practice is however deprecated, and it is suggested to use system blocks with only a single datapath. To express datapath interconnections, make use of structural hierarchy such as for example shown in Listing 4.

The main motivation to do so is to make the modeling style more consistent, and to enable future GEZEL tools to perform type checking on the interconnect.

This modification was done as a result of discussions with Jorgen Steensgaard-Madsen, DTU.

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 2.5, "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 2.5 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.    always {
15.      o = a;
16.      a = a + 2;
17.      $display("C ", $cycle, ": i=", o, " o=", i);
18.    }
19. }
20.
21. dp sysavg {
22.   sig i, o : ns(8);
23.   use avg(i, o);
24.   use tst(o, i);
25. }
26.
27. system S {
28.   sysavg;
29. }
```

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—19 show a simple testbench that will feed a string of even numbers to this four-phase averager. Finally, lines 21—29 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;
}
```

This description style gives you precise control over how the implementation will look like, but requires more modeling as the control operations have to be written down explicitly as expression. We will discuss the tradeoff between single-sfg/ multiple-sfg description styles in Section 3.4, “Choosing a controller style,” on page 27.

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.

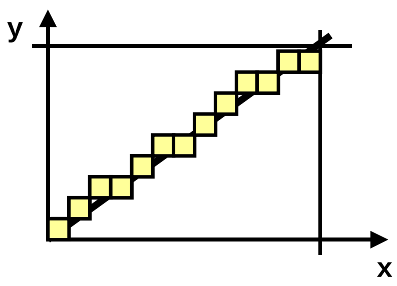
A Finite State Machine will take one state transition per clock cycle. During this state transition, a datapath instruction (one or more `sfg`) can be executed. A state transition can be conditional. In that case, the condition is based on the values of registers in the datapath (or on logical expressions 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. dp sysbresen {
72.   sig x1, y1, x2, y2 : tc(12);
73.   use bresen(x1, y1, x2, y2);
74.   use test_bresen(x1, y1, x2, y2);
75. }
76.
77. system S {
78.   sysbresen;
79. }
```

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—79 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 *behavioral* description. A GEZEL description is called structural if it uses only a single signal flow graph for a datapath that is executed at each clock cycle — cfr. the `alway` signal flow graph 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):

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

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

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

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

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

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

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

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

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

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);

  always {
     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);
  }
}
```

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 64.
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 33.
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.

The target cyclecount is a positive number (long), or the value -1 to indicate infinity (i.e. the target cycle count will never be reached).

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 {
  always {$display("Cycle ", $cycle);}
}

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.   always {$display("Cycle ", $cycle);}
5. }
6.
7. system S {
8.   mydp;
9. }
```

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 a signal flow graph's 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.

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.

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

`$hex, $bin, $dec`	Used as a `$display` argument. Changes the base of the next argument to hexadecimal, binary, or decimal.
`$finish`	Used inside of an `sfg`. Aborts the simulation.
`$trace(expres- sion, “file.txt”);`	Used after register/signal definitions in `dp`. Records value of expression each clock cycle in `file.txt`
`$trace`	Used as an instruction in an FSM state transition. Echoes the state transision to the screen.
`$option “string”`	Includes optional simulator profiling. string is one of `debug, vcd, profile_toggle_upedge_cycles, profile_toggle_alledge_operations, profile_toggle_upedge_operations, profile_toggle_alledge_cycles` (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 illustrated on the Galois Field Multiplier from Section 3.5 on page 28. 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");
```

```

 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);
  always {
    ctl = ctl + 1;
    fp  = 0b0011;
    i1  = 0b1101;
    i2  = 0b1001;
    mul_st = (ctl == 0) ? 1 : 0;
  }
}

dp sysgfmul {
  sig fp, i1, i2, mul : ns(4);
  sig mul_done, mul_st: ns(1);
  use gfmul(fp, i1, i2, mul, mul_st, mul_done);
  use TB   (fp, i1, i2, mul_st);
}
```

```
62.
63. system S {
64.   sysgfmul;
65. }
```

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.

```
> 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 Value-Change Dump (VCD) files

When there are `$trace` statements in your code, and you run the simulator in debug mode (`-d` flag), then a VCD trace file will be generated as well. This file, called `TRACE.vcd`, can be read by digital waveform viewing tools such as GTKWave (http://www.geda.seul.org/tools/gtkwave/). The VCD file will contain only the signals for which you have written `$trace` statements.

The debug mode generates a lot of output on the terminal. For this reason, it is also possible to enable/disable the debug mode and the VCD file generation independently. The statement

```
$option “debug”
```

at the top of your file will enable *only* the debug mode that prints values of registers as they change to the terminal. The statement

```
$option “vcd”
```

at the top of your file will enable *only* the VCD mode that creates the `TRACE.vcd` file.

4.6 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 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. The GEZEL cosimulation engine is called `gplatform`. It supports cosimulations with one or more ARM cores as well with one or more 8051 microcontrollers.

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.

5.1 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.

5.2 The gplatform tool

The gplatform tool is able to do everything what you can do with `armcosim`, `armzilla`, and `gezel51`. It was introduced as of GEZEL 1.7 to start consolidating the amount of cosimulation tools supported in the GEZEL release without trimming down on the flexibility or capabilities.

Eventually, gplatform will support a wide range of system architectures including single-processor systems, loosely coupled as well as tightly coupled multiprocessor architectures, and homogeneous as well as heterogeneous systems. In a *loosely coupled* system, each core has a private memory program space. In a *tightly coupled* system, multiple cores will share a single program space. The current version of gplatform supports loosely-coupled multiprocessor systems including an arbitrary configuration of ARM and i8051 cores.

The command line of gplatform is as follows

```
gplatform [-d] -c max_cycles gezel_file
```

The system configuration is fully contained within `gezel_file`.

The `-c` flag allows to indicate an upperbound for the amount of cycles to simulate. By default, the cosimulation will run until all instruction-set simulators have completed execution of their application program (these stopping conditions may vary from core to core).

Example 1 - Cosimulation with a single ARM

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

LISTING 12. A GEZEL description of hardware-side of hardware/software handshake

```
// ARM core running program ‘listing13’
ipblock myarm {
  iptype "armsystem";
  ipparm "exec=listing13";
}

// Cosimulation interfaces
ipblock b1(in data : ns(8)) {
  iptype "armsystemsink";
  ipparm "core=myarm";
  ipparm "address=0x80000000";
}
ipblock b2(out data : ns(8)) {
  iptype "armsystemsource";
  ipparm "core=myarm";
  ipparm "address=0x80000004";
}
ipblock b3(out data : ns(32)) {
  iptype "armsystemsource";
  ipparm "core=myarm";
  ipparm "address=0x80000008";
}

// hardware receiver
dp D2(in req : ns(8); out ack : ns(8); in data : ns(32)) {
  reg reqreg  : ns(8);
  reg datareg : ns(32);
  sfg sendack {
   ack = 1;
  }
  sfg sendidle {
   ack = 0;
  }
  sfg read {
   reqreg  = req;
   datareg = data;
  }
  sfg rcv {
    $display("data received ", data, " cycle ", $cycle);
  }
}
fsm F2(D2) {
  initial s0;
  state   s1, s2;
  @s0 (read, sendack) -> s1;
  @s1 if (reqreg) then (read, rcv, sendidle) -> s2;
                  else (read, sendack)       -> s1;
  @s2 if (reqreg) then (read, sendidle)      -> s2;
                  else (read, sendack)       -> s1;
}

dp sysD2 {
  sig r, a : ns(8);
  sig d    : ns(32);
  use myarm;
  use D2(r,a,d);
  use b1(a);
  use b2(r);
  use b3(d);
}

// connect hardware to cosimulation interfaces
system S {
  sysD2;
}
```

A GEZEL file for cosimulation will in general include the following elements.

- One or more cores which will be simulated using an instruction-set simulator

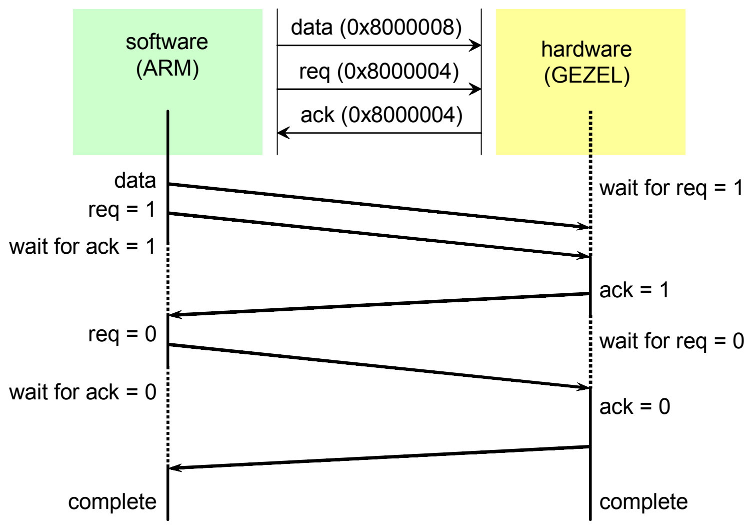


FIGURE 5.6. Two-phase full handshake protcol between software running on an ARM and hardware described in GEZEL.

- One or more cosimulation interfaces that provide communication channels from GEZEL to the application programs on the core
- For the hardware part of the cosimulation, a hardware description using FSMD semantics.

The first two bullets (cores and cosimulation interfaces) are expressed with the GEZEL library block mechanism (`ipblock`). An `ipblock` is a library block with similar semantics as a datapath `dp`. Library blocks are discussed in detail in Section 8.0 on page 64. For the purpose of this discussion, suffices to say that a library block as a type and one or more parameters. A library block's type is expressed using the `ipblock` statement, while a library block's parameters are expressed using the `ipparm` statement.

Lines 1—5 in Listing 12 include an ARM core in the simulation. It has type 'armsystem', which means it is a complete instruction-set simulator including its' program memory. The application program that must be loaded into the program memory is given as a parameter to this library block, on Line 4. In this case, we specify the application program is stored in the executable `listing13`.

Lines 6—22 define three cosimulation interfaces between GEZEL and the ARM. These interfaces are unidirectional, memory-mapped interfaces. There are two types of memory-mapped interfaces:

- `armsystemsink` blocks, such as in lines 8—12. These are channels from GEZEL to the ARM; they are a data sink for GEZEL. These blocks define an *input* port on the library block where data to be send to the ARM is provided.
- `armsystemsource` blocks, such as in lines 18—22. These are channels from the ARM to GEZEL; they are a data source for GEZEL. These blocks define an *output* port on the library block where data that is received from the ARM can be retrieved.

Both armsystemsink and armsystemsource define two parameters using the ipparm field. The first parameter is the name of the ARM core they belong to. The second parameter is the address value of the ARM memory location that is shared between the GEZEL hardware block and the ARM core.

Lines 24—50 define an example hardware module 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 5.6. At the start of the two-phase full-handshake protocol, the hardware module is waiting for the req control signal to become high (lines 46—47). 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 48—49).

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

LISTING 13. A C description of software side of hardware/software handshake

```
1. int main() {
2.   volatile unsigned char *reqp, *ackp;
3.   volatile unsigned int  *datap;
4.   int data = 0;
5.   int i;
6.
7.   reqp  = (volatile unsigned char *) 0x80000004;
8.   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.

Everything 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 `gplatform:`

```
> gplatform listing12.fdl
armsystem: loading executable [listing13]
armsystemsink: set address 2147483648
data received 0 cycle 29365
data received 1 cycle 29527
data received 2 cycle 29563
data received 3 cycle 29599
data received 4 cycle 29635
data received 5 cycle 29671
data received 6 cycle 29707
data received 7 cycle 29743
data received 8 cycle 29779
data received 9 cycle 29815
Total Cycles: 32450
```

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 36 clock cycles, a rather high value because we are working with an unoptimized C program and an unoptimized handshake protocol.

Example 2 - Dual ARM Cosimulation

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 previous example, but uses two ARM processors instead of an ARM processor and a hardware module.

The system is configured according to the following GEZEL description.

LISTING 14. GEZEL interconnect description for a two-ARM system

```
ipblock myarm1 {
  iptype "armsystem";
3.   ipparm "exec=listing15";
4. }
5.
6. ipblock myarm2 {
7.   iptype "armsystem";
8.   ipparm "exec=listing16";
9.   ipparm "period = 2"; // ARM clock = 1/2 system clock
10. }
11.
12. ipblock channelsrc1(out data : ns(32)) {
13.   iptype "armsystemsource";
14.   ipparm "core=myarm1";
15.   ipparm "address=0x80000008";
16. }
17.
18. ipblock channelsnk2(in data : ns(32)) {
19.   iptype "armsystemsink";
20.   ipparm "core=myarm2";
21.   ipparm "address=0x80000008";
22. }
23.
24. dp sys {
25.   sig src1, snk2 : ns(32);
26.
27.   use myarm1;
28.   use myarm2;
29.   use channelsrc1(src1);
30.   use channelsnk2(snk2);
31.
32.   always {
33.     snk2 = src1;
34.   }
35. }
36. system S {
37.   sys;
38. }
```

The two cores are called `myarm1` and `myarm2` respectively. Two memory-mapped interfaces enable a direct connection between these two processors. The cores and interfaces are described using the library block mechanism (Lines 1—22), in a similar fashion as in the previous example. The interconnection network is described using GEZEL semantics, and consists of a very simple point-to-point connection (Lines 24—38).

The software running on each of the cores is shown in Listing 15 and Listing 16 The handshaking protocol ilustrated here is an optimized version of the two-phase full-handshake protocol described in Section 5.2 on page 41. 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 15. A Sender C program of the two-ARM multiprocessor

```c
#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 16. A Receiver C program of the two-ARM multiprocessor

```c
#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  \
                                 listin15.c -o listing15

> /usr/local/arm/bin/arm-linux-gcc -static \
                                 listing16.c -o listing16
```

Next, run `gplatform` with the GEZEL file as command line argument. `gplatform` 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.

```
> gplatform listing14.fdl
armsystem: loading executable [listing15]
armsystem: loading executable [listing16]
armsystemsink: set address 2147483656
Sender sends 0
Sender sends 1
Sender sends 2
Sender sends 3
Sender sends 4
Sender sends 5
Sender sends 6
Sender sends 7
Sender sends 8
Sender sends 9
Receiver complete - last data = 9
Total Cycles: 64971
```

Example 3 - 8051 Cosimulation

Now 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 17. A GEZEL description of the 8051 'hello' coprocessor

```
dp hello_decoder(in   ins : ns(8);
                 in   din : ns(8)) {
  reg insreg : ns(8);
  reg dinreg : ns(8);
  sfg decode   { insreg = ins;
                 dinreg = din; }
  sfg hello    { $display($cycle, " Hello! You gave me ", dinreg); }
}

fsm fhello_decoder(hello_decoder) {
  initial s0;
  state s1, s2;
  @s0 (decode) -> s1;
14.   @s1 if (insreg == 1) then (hello, decode) -> s2;
15.                        else (decode)        -> s1;
16.   @s2 if (insreg == 0) then (decode)        -> s1;
17.                        else (decode)        -> s2;
18. }
19.
20. ipblock my8051 {
21.   iptype "i8051system";
22.   ipparm "exec=driver.ihx";
23.   ipparm "verbose=1";
24. }
25.
26. ipblock my8051_ins(out data : ns(8)) {
27.   iptype "i8051systemsource";
28.   ipparm "core=my8051";
29.   ipparm "port=P0";
30. }
31.
32. ipblock my8051_datain(out data : ns(8)) {
33.   iptype "i8051systemsource";
34.   ipparm "core=my8051";
35.   ipparm "port=P1";
36. }
37.
38. dp sys {
39.   sig ins, din : ns(8);
40.
41.   use my8051;
42.   use my8051_ins(ins);
43.   use my8051_datain(din);
44.   use hello_decoder(ins, din);
45. }
46.
47. system S {
48.   sys;
49. }
```

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 26—36. 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 18. 8051 Driver program for the Hello coprocessor

```
#include <8051.h>
```

```
2. enum {ins_idle, ins_hello};
3. void sayhello(char d) {
4.   P1 = d;
5.   P0 = ins_hello;
6.   P0 = ins_idle;
7. }
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. It writes the hex value ‘55’ to port P3; this is a specific convention for this simulator.

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

```
> sdcc listing18.c
```

The compiler creates several intermediate files, as well as a hex-dump format of the compiled code in Intel Hex format, `listing18.ihx`. Next, run the `gplatform` simulator to execute the cosimulation.

```
> gplatform lsiting17.fdl
i8051system: loading executable [listing18.ihx]
0xFF    0x03 0xFF 0xFF
0x01    0x03 0xFF 0xFF
9612 Hello! You gave me 3/3
0x00    0x03 0xFF 0xFF
0x00    0x02 0xFF 0xFF
0x01    0x02 0xFF 0xFF
9753 Hello! You gave me 2/2
0x00    0x02 0xFF 0xFF
0x00    0x01 0xFF 0xFF
0x01    0x01 0xFF 0xFF
9894 Hello! You gave me 1/1
0x00    0x01 0xFF 0xFF
0x00    0x01 0xFF 0x55
Total Cycles: 9987
```

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.

Outlook for gplatform

The goal of gplatform is to support a wide range of system architectures including single-processor systems, loosely coupled as well as tightly coupled multiprocessor architectures, and homogeneous as well as heterogeneous systems. In a *loosely coupled* system, each core has a private memory program space. In a *tightly coupled* system, multiple cores will share a single program space. The current version of gplatform supports loosely-coupled multiprocessor systems including an arbitrary configuration of ARM and i8051 cores.

5.3 Things to keep in mind with cosimulation

In general, the speed of a good instruction-set simulator is far higher than that of the GEZEL kernel. This is because an ISS is developed with the architecture of the processor it must model in mind, which is not possible for the GEZEL kernel. Also, the GEZEL kernel uses scripted simulation, rather than compiled simulation.

As an optimization, the GEZEL simulator uses a strategy of sleep/awake modes as was discussed in Section 4.1 on page 31. 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.

6.0 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.

6.1 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 5.1 on page 40 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.3 on page 79). 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.

6.2 GEZEL/SystemC Cosimulation Interfaces

As with C-based cosimulation discussed in Section 5.2 on page 41, 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.

6.3 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 19. 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. dp sysfir {
83.   sig reset, input_valid, output_data_ready : ns(1);
84.   sig sample, result : tc(32);
85.   use fir(reset, input_valid, sample, output_data_ready, result);
86.   use systemc_reset(reset);
87.   use systemc_input_valid(input_valid);
88.   use systemc_sample(sample);
89.   use systemc_output_data_ready(output_data_ready);
90.   use systemc_result(result);
91. }
92. system S {
93.    sysfir;
94. }
```

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
```

6.4 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 may be just the tool you need.
- **Deterministic Simulation**. GEZEL hardware does not (and cannot) generate race conditions. A semantically correct GEZEL model will never generate an ‘X’ during hardware simulation.

7.0 Cosimulating GEZEL with JAVA

GEZEL supports cosimulation with JAVA, by means of a few native interface classes. This is useful if you need to integrate GEZEL into a JAVA-based simulation environment.

In this chapter, the native interface classes are presented, as well as a few examples. Before you start working with JAVA, make sure you have correctly set up a working JAVA environment, and that you have enabled to JAVA cosimulation capabilities of GEZEL (See Section A.4 on page 80).

7.1 The GEZEL JAVA Native Interface

In JAVA, the interface between GEZEL and JAVA is based in three classes. `GezelModule` enables to load GEZEL code, and `GezelInport` and `GezelOutport` enable communication for writing to GEZEL and reading from GEZEL respectively.

In GEZEL, the interface between GEZEL and JAVA is based on two library blocks, one that will attach to a `GezelInport` class and another one that will attach to a `GezelOutport` class. GEZEL is assumed to be configured as a slave simulator, i.e. the GEZEL clock will be controlled out of JAVA.

A `GezelModule` is initialized using a GEZEL source file as argument. This file is parsed during instantiation of the JAVA object. Once the object is instantiated, the GEZEL simulation can be advanced one clock cycle by calling `tick()`. In the present implementation of the cosimulation interface, only a single `GezelModule` class is allowed per JAVA program. This class can of course contain multiple FSMD modules that each have their own interface to the JAVA program.

```
class GezelModule {
    public native void loadfile(String filename);
    public native void tick();
    GezelModule(String filename) {
        loadfile(filename);
    }
}
```

A `GezelInport` is a communication channel from JAVA to GEZEL. During construction of such a port, a symbolic name must be given to this part. This symbolic name can be referred to from within GEZEL to link up to corresponding library block. Once the port class is created, data can be send to GEZEL using the `write()` method. Communication is always immediate and will be available to GEZEL during the next clock cycle.

```
class GezelInport {
    int portId;
    static int glbPortId;
    public native void portname(String portname);
    public native void write(int n);
    GezelInport(String _portname) {
        portId = glbPortId++;
        portname(_portname);
    }
}
```

A `GezelOutport` is a communication channel from GEZEL to JAVA. During construction of this object, a symbolic name must be given that can be referred to from within GEZEL. Once the object is created, data can be read from GEZEL using the `read()` method. Communication is immediate, and will return the value evaluated by GEZEL in the present clock cycle.

```
class GezelOutport {
    int portId;
    static int glbPortId;
    public native void portname(String _portname);
    public native int read( );
    GezelOutport(String _portname) {
        portId = glbPortId++;
        portname(_portname);
    }
}
```

7.2 A small example

We present the case of a counter in GEZEL, integrated into a JAVA simulation. The increment value of the counter will be programmed out of JAVA. First, consider the GEZEL description of the counter.

LISTING 20. A GEZEL counter interfacing to JAVA

```
dp mycounter(in v : ns(32)) {
  reg a : ns(5);
  always run {
   a = a + v;
   $display("counter = ", a);
  }
}

ipblock myjavasource(out data : ns(32)) {
  iptype "javasource";
  ipparm "var=myinput";
}

dp syscounter {
  sig a : ns(32);
  use mycounter(a);
  use myjavasource(a);
}

system S {
  syscounter;
}
```

A counter block (Lines 1—8) is attached to a library block that represents the JAVA interface (Lines 10—13). The `javasource` library block transports data from JAVA to GEZEL. This particular block has the symbolic variable name `myinput` (Line 12).

The JAVA code that uses this counter block is shown in Listing 21.

LISTING 21. JAVA driver for GEZEL counter

```
1. class counter3 {
2.     public static void main(String[] args) {
3.         System.loadLibrary("gzljava"); // gezel-java interface
4.         GezelModule  m = new GezelModule("counter3.fdl");
5.         GezelInport  p = new GezelInport("myinput");
6.
7.         p.write(5);
8.         for (int i=0; i< 10; i++) {
9.             m.tick();
10.         }
11.     }
12. }
```

The JAVA-GEZEL interface makes use of native class implementations, which must be read in and linked at runtime. A shared library `gzljava` is loaded for this purpose at line 3. Lines 4 and 5 illustrate how a GEZEL module is instantiated, and a communication channel is established. The GEZEL file is parsed during construction of the `GezelModule` class. Lines 4—10 give a small example how the GEZEL counter is exercised. An increment value of 5 is provided, and the GEZEL simulation is run for 10 clock cycles.

To compile and run this cosimulation, first compile the JAVA code into a class file. The class path is set up to point to the location of the native classes for the GEZEL interface. This path will vary depending on the location where you have installed the GEZEL environment.

```
> javac -classpath build/share counter3.java
```

You are now ready to run. You need to make sure the JAVA virtual machine will be able to find the GEZEL-JAVA shared library that contains the GEZEL simulator. This can be done through the environment variable `LD_LIBRARY_PATH`. You also need to make sure the native classes for the GEZEL interface can be found, by selecting an appropriate `classpath` parameter. With these two paths correctly set, the simulation generates the following output.

```
> export LD_LIBRARY_PATH=build/lib; \
  java -classpath build/share:. counter3
javasource: set variable myinput
counter = 0/5
counter = 5/a
counter = a/f
counter = f/14
counter = 14/19
counter = 19/1e
counter = 1e/3
counter = 3/8
counter = 8/d
counter = d/12
```

7.3 Cosimulation with AVRORA

The JAVA cosimulation interface can be used to cosimulate GEZEL with the AVRORA simulator, an instruction-set simulator for the Atmel AVR developed by B. Titzer at UCLA (http://compilers.cs.ucla.edu/avrora/). Some examples of AVR-GEZEL cosimulation are provided in the examples directory of GEZEL (test/java/arvorax).

In order to use the AVR cosimulator, you will need to download and configure the AVRORA tools, as well as a cross-compiler for the AVR.

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 2.6 on page 17).

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

LISTING 22. A RAM library block testbench

```
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";
}

dp tmac(out address : ns(5);
   out wr, rd   : ns(1);
   out idata    : ns(8);
   in  odata    : ns(8)) {
   reg ar       : ns(5);
   reg idr, odr : ns(8);

  sfg write  { wr = 1; rd = 0; idata = idr; odr = odata; address = ar;
     $display($cycle, ":ar ", ar, " idata ", idata);
   }
  sfg read   { wr = 0; rd = 1; address = ar; odr = odata; idata = idr;
     $display($cycle, ":ar ", ar, " odata ", odata);
   }
  sfg incadr  { ar = ar + 1; idr = idr + 1;}
  sfg clraddr { ar = 0; }
}

fsm ftmac(tmac) {
  state   s1;
  initial s0;
  @s0 if (ar == 4) then (write, clraddr ) -> s1;
  else   (write, incadr)  -> s0;
  @s1 if (ar == 4) then (read,  clraddr)  -> s0;
  else   (read,  incadr)   -> s1;
}

dp sysRAM {
  sig adr  : ns(5);
  sig w, r : ns(1);
  sig i, o : ns(8);
  use M   (adr, w, r, i, o);
  use tmac(adr, w, r, i, o);
}

```

```
44. system S {
45.   sysRAM;
46. }
```

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. 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.
rand16	Function	A 16-bit random number generator.	
	IO	`o`	output data
Library Blocks in `gplatform` (Section 5.2 on page 41)			
armsystem	Function	ARM Core + program memory. The ISS is SimIt-ARM.	
	IO		
	Parameters	`exec`	Name of the statically linked ELF binary to be executed on the ARM
		`verbose`	When set to 1, this ARM will execute in verbose (debug) mode, visualizing all system calls as they proceed.
		`period`	Relative clock period, default 1. When set to e.g. 2, the ARM will run at half speed relative to the system (gezel) clock.
armsystemsource	Function	Memory-mapped cosimulation interface for an ARM core intercepting memory writes on this core.	
	IO	`data`	data output, ns(32)
	Parameters	`core`	Name of the armsystem block this cosimulation interface is connected to.
		`address`	Address decoded by this cosimulation interface.
armsystemsink	Function	Memory-mapped cosimulation interface for an ARM core intercepting memory reads from this core.	
	IO	`data`	data input, ns(32)
	Parameters	`core`	Name of the armsystem block this cosimulation interface is connected to.
		`address`	Address decoded by this cosimulation interface.
armsystemprobe	Function	This block allows the application software running on the ARM ISS to send messages directly to another ipblock (through the probe function as discussed in Section 8.4 on page 75).	
	IO		t

	Parameters	`probe`	Address of the ARM address space that points to the command string for the probe.
		`block`	Target library block that this probe must send messages to.
i8051system	Function	i8051 core + program memory	
	IO		
	Parameters	`exec`	Name of the intel-hex formatted i8051 binary to execute
		`verbose`	When set to 1, run the ISS in verbose (debug) mode.
		`period`	Relative clock period, default 1. When set to e.g. 2, the 8051 will run at half speed relative to the system (gezel) clock.
i8051systemsource	Function	Port-mapped cosimulation interface to transport data from 8051 to GEZEL.	
	IO	`data`	output, ns(32), data output.
	Parameters	`core`	name of the i8051system core this port-mapped interface belongs to.
		`port`	quoted string, one of `P0`, `P1`, `P2`, `P3`.
i8051systemsink	Function	Port-mapped cosimulation interface to transport data from GEZEL to 8051 to GEZEL.	
	IO	`data`	input, ns(32), data input.
	Parameters	`core`	name of the i8051system core this port-mapped interface belongs to.
		`port`	quoted string, one of `P0`, `P1`, `P2`, `P3`.
Library Block in `libgzlsys.a` (SystemC cosimulator, Section 6.2 on page 52)			
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 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 31). 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 23 shows how to program the runlength encoder as a derived class from the base class `aipblock.`

LISTING 23. 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;
     runlength = 0;
     maxlen = 256;
   }
   void run() {
     ioval[1]->assignulong(0);
     ioval[2]->assignulong(0);
     if (ioval[0]->toulong() == (unsigned) previous_data_value) {
       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.

GEZEL supports dynamically linked library blocks. You can compile the C++ code into a shared library. At runtime, the GEZEL kernel will link in these library blocks at the moment you need them. Compilation into a shared library block proceeds as follows. In these commands, BUILD must be substituted with the path the the GEZEL installation.

```
> g++ -fPIC -O3 -Wall -c -IBUILD/include/gezel iprle.cc

> g++ -shared -O3 -Wl,--rpath -Wl,BUILD/lib iprle.o BUILD/lib/libi-
pconfig.so BUILD/lib/libfdl.so -lgmp -ldl -o librle.so
```

This will create a shared library librle.so. When you run the GEZEL simulation (fdlsim or gplatform), this library must be available in the directory where the simulation is run. It will be automatically read in at the moment the rle ipblock is used by your simulation.

An example GEZEL program that uses the runlength encoder program is shown

LISTING 24. 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);

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

dp sysrle {
  sig i, tn, td : ns(8);
  use my_rle(i, tn, td);
  use senddata(i, tn, td);
}

```

```
system S {
  sysrle;
}
```

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)
```

8.4 Other member functions for `aipblock`

```
virtual void aipblock::stop();
```

This function is called in `gplatform` (see Section 5.2 on page 41) when the simulation terminates.

```
virtual void aipblock::probe(char *);
```

This is a *probing* function, to be used in combination with a cosimulation interface. It allows an external simulator (such as an ISS) to query specific GEZEL ipblocks. An example of a library block that calls this function is `armsimprobe`.

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 cosimulator for SystemC 2.0.1
- A platform simulator for ARM and i8051
- A cosimulator for JAVA code
- A test subdirectory with several examples for each cosimulator

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

```
> tar xfvz gezel-2.x.tar.gz (On Linux)

> cd gezel-2.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.

In most cases (including common Linux distributions), these tools will already be available on your system and you do not need to download any extra packages.

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)
```

A.2 gplatform Cosimulator

GEZEL can be cosimulated with instruction set simulators, using the C++ API on the backend of GEZEL.

Installing the ARM instruction-set simulator

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`. If you plan to install the cosimulators in a different location than the standard build subdirectory, use the `--prefix` command line option with configure:

```
> ./configure CPPFLAGS=-'DCOSIM_STUB' --prefix=my_target_dir
```

In particular, it is not a good idea to copy executables from the build directory to a target directory by hand. This is because SimIt-ARM hard-codes the default path to the floating point emulator that it relies on (`nwpfe.bin`).

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`).

Installing the i8051 instruction-set simulator

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 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.

Installing the gplatform cosimulator

Once you have prepared the ARM ISS, you can configure GEZEL to enable compilation of gplatform. Use the `--enable-gplatform` flag for this. If required, also use the --with-simit flag to indicate the path the the ARM ISS.

```
> ./configure --enable-gplatform --with-simit=/opt/Simit/build

> make

> make install
```

To test the installation, run one of the examples of `gplatform` under the `test/` directory, for example:

```
> make
/usr/local/arm-3.3.2/bin/arm-linux-gcc -static  hello.c -o hello

> make sim
../../../build/bin/gplatform  hellomodel.fdl
armsystem: loading executable [hello]
```

```
armsystem: loading executable [hello]
Hello: [Jefke] [Piet] [and] [Pol]
Hello: [Jefke] [Piet] [and] [Pol]
Total Cycles: 43653
```

A.3 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 6.0 on page 52.

A.4 JAVA Cosimulator

The GEZELkernel is also accessible to JAVA applications through a set of three JAVA classes. You can download a JAVA developer kit from SUN (http://java.sun.com) or IBM

(//http://www-106.ibm.com/developerworks/java/jdk/linux140/) if it is not available in the machine you are working on.

The location of the javac and javah tools (which are used to compile the JAVA-gezel link) must be set in the Makefile.am under the java subdirectory of the GEZEL distribution. You have to make this modification *before* running `configure`. The default values on this line are:

```
JAVAC=/opt/IBMJava2-142/bin/javac

JAVAH=/opt/IBMJava2-142/bin/javah
```

After confirming the paths are properly configured, run `automake` to regenerate the Makefiles.

Next, compile the java-GEZEL classes as well as the shared library that links them to the GEZEL kernel, by running configure using the `--enable-java` flag:

```
> ./configure --enable-java

> make

> make install
```

To confirm that the JAVA class library works, try one of the examples under `test/java`. Also here, the path the `javac` and `javah` must be provided. This is done in `test/java/Makefile.rules`.

```
> cd test/java/counter1

> make
/opt/IBMJava2-142/bin/javac -classpath ../../../java counter1.java

> make sim
export  LD_LIBRARY_PATH=../../../build/lib;  /opt/IBMJava2-142/bin/
java -classpath ../../../build/share:. counter1
counter = 0/1
counter = 1/2
counter = 2/3
counter = 3/4
counter = 4/5
counter = 5/6
counter = 6/7
counter = 7/8
counter = 8/9
counter = 9/a
```

A.5 AVRORA Cosimulator

One application of the JAVA cosimulation interface is a cosimulation with the AVRORA instruction-set simulator. AVRORA simulates the Atmel AVR and is developed by B. Titzer at the compilers group at UCLA (http://compilers.cs.ucla.edu/avrora/).

The examples under test/java/avrora* illustrate an integration between GEZEL and AVRORA based on the `Platform` class from AVRORA. To run this example, first download and install the AVRORA class library. The path the the AVRORA ISS needs to be indicated in `test/java/Makefile.rules`

```
JAVAC=/opt/IBMJava2-142/bin/javac

JAVA=/opt/IBMJava2-142/bin/java

AVRORACLS=/home/schaum/avrora/bin

GEZELCLS=/home/schaum/gezel/devel/build/share

GEZELLIB=/home/schaum/gezel/devel/build/lib
```

In order to run the examples (under `test/java/avrorax`) you also need to install a cross compiler for the AVR (such as `avr-gcc`). Then, compile and run the examples in the usual way:

```
> make
/opt/IBMJava2-142/bin/javac   -classpath   /home/schaum/avrora/bin:/
home/schaum/gezel/devel/build/share hwsw.java

> make sim
avr-gcc -mmcu=atmega128 -ggdb simple.c -o app.out
avr-objdump -zhD app.out >app.od
export   LD_LIBRARY_PATH=/home/schaum/gezel/devel/build/lib;   /opt/
IBMJava2-142/bin/java  -classpath  /home/schaum/avrora/bin:.:/home/
schaum/gezel/devel/build/share  avrora.Main  -colors=false  -plat-
form=hwsw app.od

Avrora [Beta 1.4.0] - (c) 2003-2005 UCLA Compilers Group
This simulator and analysis tool is provided with absolutely no war-
ranty, either expressed or implied. It is provided to you with the
hope that it be useful for evaluation of and experimentation with
microcontroller and sensor network programs. For more information
about the license that this software is provided to you under, spec-
ify the "license" option.

hwsw platform
javasource: set variable PA0
javasource: set variable PA1
javasink: set variable PA2
javasink: set variable PA3
=={ Simulation events }==============================
Node       Time   Event
--------------------------------------------------
bits 8/7 buf 0/1
bits 7/6 buf 1/2
bits 6/5 buf 2/5
bits 5/4 buf 5/a
bits 4/3 buf a/15
bits 3/2 buf 15/2a
. . .
```

```
Received data aa/aa
Received data aa/aa
Received data aa/aa
Received data aa/aa
===============================================
Simulated time: 1471 cycles
Time for simulation: 0.024 seconds
Simulator throughput: 0.061291665 mhz
```

A.6 Catalog of examples

Tool	Example	Function
gezel	aes	Advanced Encryption Standard block.
	bresen	Example of a Bresemham line drawing algorithm.
	euclid	Euclid’s GCD algorithm
	gf8inv	Galois Field inversion block with subfields
	ipchecksum	Checksum evaluation block for IP packets
	lfsr	Linear Feedback Shift Register
	ramblock	Illustration of writing and reading into a RAM ipblock.
	scripted	Example of alternative invocation of fdlsim
gplatform	arm2arm	Two-way handshake in a dual-ARM system
	gfmul	8051-based GF multiplier connected to an ARM driver
	aes8051	AES coprocessor attached to an 8051 core
	aesarm	AES coprocessor attached to an ARM core
	euclid	GCD coprocessor attached to an ARM core
	ramprobe	An example of the probing function to query the contents of a RAM ipblock out of software running on the ARM.
	cyclecount	An example of making GEZEL print out current cycle count in a cosimulation.
	hello51	Simple handshake on a 8051 core
	helloarm	Simple handshake on an ARM core
	ssidehsk	One-way handshake in a dual-ARM system
java	avrora1	Serial-to-parallel conversion in GEZEL as AVR coprocessor
	avrora2	Example of port input-output on an AVR platform.
	counter1	A GEZEL counter
	counter2	A JAVA-readable GEZEL counter
	counter3	A JAVA-readable and controllable GEZEL counter
	euclid	Euclid’s algorithm as a GEZEL coprocessor
systemc	accum	Multiply-accumulate in GEZEL called out of SystemC

A.7 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`)

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

Appendix B: References

B.1 Publications on the GEZEL language and tools

P. Schaumont, D. Ching, I. Verbauwhede, "An interactive codesign environment for domain-specific coprocessors," submitted to ACM Transactions on Design Automation for Embedded Systems.

P. Schaumont, I. Verbauwhede, "The descriptive power of GEZEL," Technical Report, Jan 30, 2005. [Available online on the GEZEL homepage].

P. Schaumont, I. Verbauwhede, “Interactive Cosimulation with Partial Evaluation, 2004 Design Automation and Test in Europe (DATE 2004),” Februari 2004.

P. Schaumont, I. Verbauwhede, “Domain-specific Co-design for Embedded Security,” IEEE Computer, April 2003.

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.

B.2 Publications on applications that have used GEZEL

S. Yang, P. Schaumont, I. Verbauwhede, “Microcoded Coprocessor for Embedded Secure Biometric Authentication Systems,” International Conference on Hardware/Software Codesign and System Synthesis, September 2005.

B.C. Lai, P. Schaumont, I. Verbauwhede, "Energy and Performance Analysis of Mapping Parallel Multi-threaded Tasks for An On-Chip Multi-Processor System,” IEEE International Conference on Computer Design (ICCD 2005), October 2005.

I. Verbauwhede, P. Schaumont, "Skiing the embedded systems mountain," accepted for publication in ACM Transactions on Embedded Computing Systems (Special issue on embedded systems education).

P. Schaumont, D. Hwang, I. Verbauwhede, "Platform-based design for an embedded fingerprint authentication device," accepted for publication in IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems.

P. Schaumont, B.C. Lai, W. Qin, I. Verbauwhede, "Cooperative multithreading on embedded multiprocessor architectures enables energy-scalable design," Proc. 2005 Design Automation Conference (DAC 2005), June 2005.

K. Tiri, D. Hwang, A. Hodjat, B.C. Lai, S. Yang, P. Schaumont, I. Verbauwhede, "A Side-Channel Leakage Free Coprocessor IC in .18um CMOS for Embedded AES-Based Cryptographic and Biometric Processing," Proc. 2005 Design Automation Conference (DAC 2005), June 2005.

H. Chan, P. Schaumont, I. Verbauwhede, "A secure multithreaded coprocessor interface," 3th Workshop on Optimizations for DSP and Embedded Systems, March 2005. Available from workshop website.

O. Villa, M. Monchiero, G. Palermo, P. Schaumont, I. Verbauwhede, "Fast Dynamic Memory Integration in Co-Simulation Frameworks for Multiprocessor System on-Chip," Proceeding In DATE 2005- Design, Automation and Test in Europe, Designers' Forum, 2005.

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

Y. Matsuoka, P. Schaumont, K. Tiri, and I. Verbauwhede, "Java cryptography on KVM and its performance and security optimization using HW/SW co-design techniques," Proc. Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2004), pp. 303-311, September 2004.

P. Schaumont, K. Sakiyama, A. Hodjat, I. Verbauwhede, "Embedded software integration for coarse-grain reconfigurable architectures," 2004 Reconfigurable Architectures Workshop (RAW 2004), April 2004.

P. Schaumont, I. Verbauwhede, "ThumbPod puts security under your Thumb", Xilinx Xcell Online, Winter 2003.

P. Schaumont, K. Sakiyama, Y. Fan, D. Hwang, B. Lai, A. Hodjat, S. Yang, I. Verbauwhede, "Testing ThumbPod: softcore bugs are hard to find," IEEE International High Level Design Validation and Test Workshop (HLDVT), November 2003.

D. Hwang, P. Schaumont, Y. Fan, A. Hodjat, B.C. Lai, K. Sakiyama, S. Yang, I. Verbauwhede, "Design flow for HW / SW acceleration transparency in the ThumbPod secure embedded system," 2003 Design Automation Conference, pp. 60-65, Los Angeles, June 2003.

B.3 Publications on SimIT-ARM (ARM Cosimulators)

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.

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.