GEZEL Language Reference 2.0

Patrick Schaumont
pschaumont@gmail.com

This is a language reference manual that explains the syntax and semantics of the GEZEL language in a tool-independent way.

1.0 Concepts

The GEZEL language models networks of cycle-true, finite-state-machine and datapath (FSMD) modules. Interconnections between modules or library blocks are wires. GEZEL makes synchronous (cycle-true) descriptions. All modules are attached to a single, implicit clock.

A module is a combination of a controller and a datapath, or a library block. A datapath has a number of input ports or output ports that connect to other modules. A datapath is defined in terms of signal flow graphs, pieces of behavior that describe the operations in that datapath during a single clock cycle. Those operations take values from input ports or datapath registers, and transform those using expressions into results that can be written to output ports or data path registers. A controller is used to schedule datapath instructions per clock cycle. Several controller types are available, going from a simple hardwired controller to a more elaborate finite state machine.

Library blocks are predefined blocks provided by the GEZEL environment. They are not developed in the GEZEL language. They are used for specific tasks, such as hardware/ software codesign interfaces and memory modules.

2.0 Language Building blocks

Formatting. The GEZEL language is context-free and ignores whitespace. A comment starts with a double slash (//) or a sharp-exclamation (#!) and runs to the end of the line.

Identifiers. An identifier is a sequence of characters (a-z, A-Z) or numbers or underscore. The first character must not be a number.

Constants. A numeric constant can be in decimal, hexadecimal (e.g. 0xFF) or binary format (e.g. 0b110101). String constants, used by some directives, contain a character sequence between double quotes. Subsequent strings in double quotes are treated as a single string.

Types. GEZEL variables and ports have a wordlength and a sign. The sign is unsigned (ns) or two's complement signed (tc). A type specification is given by combining a sign

with a wordlength. For example, ns(10) is an unsigned, 10-bit type. tc(256) is a 256-bit signed type.

Keywords. The following is a list of keywords. None of them can be used as identifier.

$bin	always	ns
$cycle	dp	out
$dec	else	reg
$display	fsm	sequencer
$dp	hardwired	sfg
$finish	if	sig
$hex	initial	state
$sfg	in	stimulus
$trace	ipblock	system
$option	ipparm	tc
	iptype	then
	lookup	use

3.0 Datapaths

Example:

```
dp accum( in a : tc(8) ) {
  reg acc : ns (8);
  sfg clear {
    acc = 0;
  }
  sfg add {
    acc = (acc + a);
    $display("acc=", acc);
  }
}
```

This datapath defines a single input port, a, which accepts 8-bit signed signals. The datapath contains an accumulator acc and two instructions, clear and add. Both of them contain operations on the accumulator. The ‘add’ instruction also contains a simulation directive: a display statement. The datapath only specifies the available instructions, but not the schedule of those instructions. It is the task of a controller (to be described further) to define this schedule.

3.1 Port definition

The list of ports of a datapath is a list of input and output port definitions separated by semicolons. The list opens and closes with round brackets. In case a datapath has no ports, the datapath port list including the round brackets is absent.

A port has a direction (**in** or **out**) and a type. If the type and direction of several ports is identical, the identifiers of those ports can share the same type specification as a comma-separated list.

Example:

```
dp thedp(in a, b, c : ns(5); out d : ns(1); in e : tc(20))
```

a, b and c are input ports accepting 5-bit unsigned numbers.
d is an output port generating a 1-bit unsigned number.
e is an input port accepting a 20-bit signed number.

3.2 Registers, Signals and Lookup Tables

Registers and signals are used to create expressions inside of datapath instructions. A register has two values: a current value (the value obtained when reading from the register) and a next-value (the value assigned when writing into a register). At each clock edge, the next-value is copied into the current value. A signal has a single, immediate value and is semantically identical to wiring. A signal must be defined in the same clock cycle as it is consumed. A register will hold its value until it is reassigned.

Example:

```
reg q : tc(32);
sig v, w : ns(1);
```

q is a 32-bit signed register.
v and w are 1-bit unsigned signals.

GEZEL does not support arrays. GEZEL does support lookup tables, which are constant arrays. The contents of a lookup table is defined at the same position as signals and registers. A lookup table has a name and a type specification. The type specification indicates the type of individual elements. A lookup table is accessed with the lookup operation.

Example:

```
lookup T : ns(8) = {5, 4, 3, 2, 1};
```

T is a lookup table with 8-bit, unsigned elements. 5 elements are defined. Location 0 holds the value 5, location 1 holds the value 4, and so on.

3.3 Expressions

Expressions are collections of operations with constants, registers, signals and lookup tables. Expression results can have a precision that is different from their operands. There is a default type combination rule: (a) The wordlength of the result is the maximum wordlength of the operands and (b) if any of the operands is signed, the result will be

signed. The operators, in order of precedence going from low to high, are listed in Table 1. If the type of the result is not indicated, it is created using the default rule.

TABLE 1. Operators in order of precedence (low to high)

a = expr1	Assignment Operation.
	The value of expr1 is casted into the type of a.
a ? b : c	Selection Operator.
	If a is nonzero, result is b else c.
	The type of the result is the default combination of b and c.
a | b	Bitwise Inclusive OR.
a ^ b	Bitwise Exclusive OR.
a & b	Bitwise AND.
a == b	Logical comparison for equality.
a != b	Logical comparison for inequality.
a < b	Locgical comparison for smaller-then.
a > b	Logical comparison for bigger-then.
a <= b	Logical comparison for smaller-or-equal-then.
a >= b	Logical comparison for bigger-or-equal-then.
	The type of the result of all logical comparisons is 1-bit unsigned
a << b	Left Arithmetic Shift.
	The result has the sign of a and the wordlength wl(a) + (1 << wl(b)), with wl(a) = wordlength of a and wl(b) = wordlength of b
a >> b	Right Arithmetic Shift.
a + b	Addition.
a - b	Subtraction.
a # b	Bit concatenation.
	Equivalent to (a << wl(b)) | b, wl(b) is wordlength of b).
	The resulting wordlength is wl(a) + wl(b). The result sign is that of a.
a * b	Multiplication.
	The result wordlength is wl(a) + wl(b). The result sign follows the default rule.
a % b	Modulo-operation
(type_spec) a	Cast a to type_spec. Type spec is a standard type specification, e.g. ns(5) for a 5-bit unsigned number.
	The type of the result is defined by type_spec.

TABLE 1. Operators in order of precedence (low to high)

- a	Unary minus. The result type has the type of a.
~ a	Bitwise NOT. The result type has the type of a.
a[number]	Bit selection (Result is a 1-bit unsigned number).
a[from_num:to_num]	Bit vector selection (Result is (from_num - to_num + 1)-bit unsigned, assuming from_num > to_num).
a(expr)	Lookup operation in lookup table a. Result is defined by the type of the lookup table elements.
(a)	Grouping operator, used to modify precedence rules (e.g. a + (b*c)).

3.4 Signal Flowgraph Definition

Signal flowgraphs define the instructions of a datapath. Each signal flowgraph (**sfg**) collects a number of expressions, separated by a semicolon. Each **sfg** represents 1 cycle of behavior. All expressions in an **sfg** execute concurrently, but will follow the data precedences between expressions and the semantics of registers and signals. Each **sfg** has a symbolic name, by which this instruction can be referred. Any number of **sfg** can be active during a particular clock cycle. This is decided by the controller on top of the datapath. An **sfg** should be thought of as a behavior, not as a structure.

The operands of the expressions in an sfg must be either datapath inputs, datapath outputs, registers defined within the datapath or signals defined within the datapath.

Example:

```
reg a : ns(4);
sig b : ns(4);
sfg myinstruction {
  a = b + 3;
  b = 2;
}
```

In the sfg myinstruction, the bottom expression (b=2) will be evaluated first because b is a signal that is consumed by the first expression. The next-value of a will be 5.

3.5 ‘always’ Signal Flowgraph

A datapath can specificy an instruction which is to be executed *each* clock cycle, regardless of the controller specification. Such an **sfg** can be expressed with the **always** keyword. An **always** instruction has no identifier.

Example:

```
reg a : ns(4);
sig b : ns(4);
always {
  a = b + 3;
  b = 2;
}
```

This instruction will assign, each clock cycle, the value 2 to the signal b, and the value 5 to the register a.

4.0 Controllers

A controller specifies a schedule for the instructions in a dapapath. Each datapath can have only a single controller. The **sfg** instructions of a datapath can only execute when a controller is specified. The **always** instruction of a datapath will execute regardless of the presence of a controller.

GEZEL provides three types of controllers.

4.1 Hardwired Controller

A hardwired controller selects a single instruction for perpetual execution.

Example:

```
hardwired mycontroller(mydatapath)           { doit; }
hardwired myothercontroller(myotherdatapath) { doit; doit2; }
```

In the first example, the controller mycontroller is attached to the datapath mydatapath and selects the **sfg** doit for execution at any clock cycle. In the second example, the controller myothercontroller is attached to datapath myotherdatapath and selects the **sfg** doit and doit for simultaneous execution at any clock cycle.

4.2 Sequencer Controller

A sequencer controller selects a sequence of single **sfg** to be executed in a sequences, cyclic fashion.

Example:

```
sequencer mycontroller(mydatapath) { doit1; doit2; doit2; }
```

The controller mycontroller is attached to the datapath mydatapath. At the first clock cycle of the simulation, **sfg** doit1 is executed. At the second and third clock cycle of the simulation, **sfg** doit2 is executed. At the fourth clock cycle of the simulation, doit1 is executed again, and so on.

Example:

```
sequencer mycontroller(mydatapath) { (doit1, doit2); doit2; }
```

The controller mycontroller is attached to the datapath mydatapath. At the first clock cycle of the simulation, **sfg** doit1 and **sfg** doit2 are simultaneously executed. At the second cycle, **sfg** doit2 is executed. The third clock cycle looks again like the first one.

4.3 Finite State Machine Controller

A finite state machine controller (FSM) combines instruction sequencing and decision making in a finite-state model. A FSM defines a finite number of states. At any moment, the FSM is in one of the defined states. One of these states is the initial state, and represents the state in which the FSM starts execution. In between states, state transitions are defined. State transitions take a single clock cycle to complete. During state transition, one or more sfg may be selected for execution. The first example shows an FSM with no decision making:

Example:

```
fsm mycontroller(mydatapath) {
  initial s0;
  state s1, s2;
  state s3;
  @s0 (doit1)        -> s2;  // state transition 1
  @s1 (doit1, doit2) -> s2;  // state transition 2
  @s2 (doit3)        -> s1;  // state transition 3
}
```

The controller mycontroller is attached ot mydatapath. This **fsm** defines 4 states: s0, s1, s2, s3. The initial state is s0, while s1, s2, s3 are normal states. The **fsm** defines three state transitions, all of them are unconditional. The first state transition starts in state s0, the initial state, and moves to state s2. During this state transition, datapath instruction doit1 will be executed. The second state transition starts in state s1 and moves to state s2. During this state transition, instructions doit1 and doit2 will be executed concurrently. The third state transition is similar to the first. In the first clock cycle of simulation, state transition one will execute. In the second clock cycle, state transition three will execute, while in the third clock cycle, state transition two will execute.

Using registers from the datapath, conditional expressions can be formed. These conditional expressions can be used to make conditional state transitions. The operands of these conditional expressions must be datapath registers, and cannot be datapath signals or datapath inputs or outputs.

Example:

```
fsm mycontroller(mydatapath) {
   initial s0;
   state   s1;
   @s0 if (a & b) then (doit1) -> s1;
       else            (doit2) -> s1;
   @s1 if (a) then
```

```
          if (b) then  (doit1) -> s0;
          else         (doit2) -> s0;
       else            (doit2) -> s0;
}
```

The controller mycontroller is attached to mydatapath. The controller defines two states and 5 state transitions, all of them conditional. The first state transition makes a bitwise and of registers a and b of datapath mydatapath, and if the result is nonzero, sfg doit1 is executed. Otherwise, the second state transition is executed together with sfg doit2. The next three state transition, starting out of state s1, tests the same condition but uses a condition hierarchy. A conditional state transition must always be specified in pairs. It is an error to write the if-part without the else part.

5.0 Hierarchy and Module Instantiation

There are no module types in GEZEL. There are no module declarations, only definitions. GEZEL provides support for structural hierarchy (enclosing modules inside of other modules) and module instantiation (module copying from existing modules).

5.1 Hierarchy

Once a datapath is defined, it can be enclosed inside of another one with the ‘use’ statement. Such insertion must be done at the same location where registers and signals are defined. When a datapath is included, it must be connected using an actual port list.

Example:

```
dp innerdp(in a : ns(5); out b : ns(5)) {
  // ...
}

dp outerdp(in c : ns(5); out d : ns(5)) {
   reg v : ns(5);
   use innerdp(c, v);
   sfg doit {
     d = v + 1;
   }
}
```

The datapath outerdp encapsulates a datapath innerdp. The datapath innerdp is connected to an input of the outerdp and a register v. The connections to the innerdp ports are made with positional matching: port a of innerdp is connected to port c of outerdp, and port b of innerdp is connected to register v of outerdp.

5.2 Module Instantiation (cloning)

A module is fully defined when both a datapath and a controller for that datapath are defined. A copy of such a module can be made with the cloning operation. The resulting

clone is functionally identical to the original, but has in an independent set of state variables.

Example:

```
dp andgate(in a22,b : ns(1); out q : ns(1)) {
  always {
    q = a22 & b;
  }
}

dp andgate2 : andgate
dp andgate3 : andgate
```

The datapaths andgate2 and andgate3 are copies from datapath andgate. Both will have also a controller, that will execute the datapath sfg active in each of datapath2 and datapath3.

6.0 Semantic Requirements for Proper FSMD Description

A datapath with a port definition, register/signal definition and instruction definition and schedule (controller) is considered to be a proper FSMD if it has deterministic behavior. Such behavior implies that the simulation will be unambiguous and race-free. The description of a proper FSMD has to obey the following four requirements.

- During any clock cycle, all datapath outputs must be defined. This means that datapath outputs must always appear at the lefthand-side of an assignment expression.
- 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.
- If an expression consumes 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.
- Neither registers, nor signals or datapath outputs can be assigned more then once during a clock cycle.

The GEZEL simulator will verify these properties during parsing and at run-time.

7.0 Directives

Datapath instructions can contain a number of directives. The effect of directives is tool-dependent, and not part of the GEZEL semantics. In fact, one can always strip out all directives of a GEZEL program without changing the behavior of that program. For a complete description of the directives, refer to the GEZEL User Manual. Directives start

with the '$' sign. Depending on their kind, they can appear at several positions on a GEZEL program.

- At the start of a GEZEL program: **$option**
- Inside of a datapath: **$trace**,
- Inside of an sfg: **$display**, **$finish**
- Inside of the $display directive: **$cycle**, **$dp**, **$sfg**, **$hex**, **$bin**, **$dec**
- Inside of a control step: **$trace**

Example:

```
$option "profile_toggle_alledge_cycles"
```

Example:

```
sig k : ns(20);
$trace(k, "kvalues.txt"); // record k in kvalues.txt
```

Example:

```
$display("The value of a is ", a);
$display($hex, a, " ", b + 1, " ", $bin, c);
$display($cycle, ": executing sfg ", $sfg);
$finish;
```

Example:

```
@s0 (sfg1, sfg2, $trace) -> s1; // use in state transition
```

8.0 Toplevel

The topcell a GEZEL description is expressed in a system block. The topcell is normally a datapath without any inputs or outputs, within which all other datapaths are contained by means of structural hierarchy.

```
system S {
  topcell;
}
```

9.0 Library Blocks

A library block in GEZEL is a custom block, available in the GEZEL environment but not written in the GEZEL language. Library blocks are defined at the moment the GEZEL simulation is configured. The set of library blocks available therefore depends on the GEZEL simulator instance. Library blocks are typically used for memory modules (RAM) and hardware/software codesign interfaces. A library block is treated in the same way as a module. It has a set of input and output ports and can be connected in a system netlist.

9.1 Syntax

The purpose of the examples below is to illustrate only the syntax of the library blocks. The semantics of library blocks is defined by the library block developer, and not a part of the GEZEL language reference.

Example:

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

ipblock nodeB(in data : ns(32)) {
  iptype "armzillasink";
  ipparm "processor=processorB";
  ipparm "address=0x80000010";
}
```

The first library block is a RAM module with 32 locations of 8 bits wide. The port list of a library block depends on the type of that library block. The names and order of the ports are fixed by the type of that library block. The RAM block is part of the GEZEL core system and available in all instances of GEZEL simulations. The **iptype** specification selects the type of the library block. The **ipparm** specification allows to select parameters on the library block. The number and nature of parameters depends on the type of the library block. In this case, the size and wordlength of the RAM module are defined.

The second library block a memory-mapped hardware/software interface. The library block is of type armzillasink, and is a data channel leading from software to hardware (GEZEL). Two parameters are defined, the first one is a processor specification, the second one the address in memory space where this hardware/software interface maps to. This library block is specific to a particular instance of the GEZEL simulator in a cosimulation environment, and is not generally available.

9.2 Hierarchy and cloning of library blocks

Library blocks can be cloned and used hierarchically in the same way as other datapath modules. Each clone of a library block will operate as an independent copy of that block.

Example:

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

```
  ipparm "wl=8";
}
ipblock M2 : M

dp mydp {
  sig a    : ns(5);
  sig w, r : ns(1);
  sig i, o : ns(8);
  use M(a, w, r, i, o);
  // ...
}
```

10.0 Multi-file descriptions and use of the C preprocessor

GEZEL currently does not support macro's or designs descriptions split over multiple files. However, the C preprocessor can be used to preprocess GEZEL files that contain #include directives or #define macro's. Such files can be preprocessed as follows:

```
cpp -P myfile.fdl >mypreprocessedfile.fdl
```

Afterwards, mypreprocessedfile.fdl can be provided to the simulator.