1 Basic

1.1 Background

Moore's Law improvements in manufacturing costs of hardware have driven dramatic increases in the variety and complexity of hardware systems that can be economically manufactured. Unfortunately, high costs of producing verified hardware designs limit deployment to devices that can amortize development costs over very high manufacturing volumes. We have seen dramatic improvements in general purpose processors and memory devices, but much more limited improvements in application specific devices.

Deploying new versions of hardware designs can be quite expensive, placing extreme emphasis on gaining confidence of correct operation before manufacturing, causing the costs of testing to dominate product development costs. How can we make the development of *verified* designs more efficient and predictable?

The key to efficent production of designs is the ability to guarantee safety (nothing "bad" happens throughout execution) and liveness (something "good" eventually does happen). If we can guarantee each of these aspects at compile-time, we significantly reduce the dependency on runtime verification.

1.2 Approach

To reduce verification costs, we propose:

— Compiler guarantees program runtime safety and (where possible) liveness

  — Through HDL structuring, guarantee the absence of dynamic concurrency interleaving errors in generated Verilog.

  — When rules conflict, the compiler synthesizes fair arbitration, guaranteeing liveness between rules.

  — To guarantee liveness of user code, assert statements must be written by the user. When possible the compiler attempts to verify them with model checking or proof tools. Since designing effective assert statements can be quite difficult, a standardized runtime is provided to allow access to assertion failures at runtime in synthesized logic as well as providing access to program trace information.

— Reuse modules without reverification. Amortize module costs through reuse.

  — A single Verilog module is generated for each defined AtomicC module. The generated Verilog is highly readable, giving designers confidence that intended operations are correctly realized.

  — Module schedules calculated using only local information and are independent of module usage context.

  — Parameterized generated code: propagation of module parameterization from source through to generated Verilog allows reuse without generation for each parameter instance.

  — Source modules are independently compiled to re-usable Verilog modules.

  - Stable generated Verilog: "local AtomicC source code changes (generally) should only cause local changes to generated Verilog". This minimizes the reverification effort for ECOs.

    In hardware design, the design phase is followed by significant validation effort to ensure manufacturability. Since the number of system parameters is large, there is an extreme need to keep all unrelated areas of a design constant when implementing engineering change orders to fix problems descovered in the manufacturing process. By modular generation of Verilog code and by enforced runtime isolation of modules, ECOs are independant.
- Standardized debug runtime
  - Display and trace runtime support is provided, easing unit testing as modules are created.
  - When not possible to statically verify asserts, a standard runtime exists for reporting errors.
  - hardware/software interactions are facilitated by a NOC DMA to processor memory, smoothing interactions between high clock speed/narrow thread software and low clock speed/wide thread hardware.
  - runtime support for scan-chain based testing
- Integration with existing workflow
  - Existing Verilog modules can be called from and can call AtomicC generated modules.
  - Standard Verilog backend tools are used to synthesize the resulting ASIC or FPGA.

1.3 AtomicC Overview

AtomicC is a timed, structural hardware description language for the high level specification of algorithms to be instantiated directly in hardware. AtomicC extends C++ with support for Guarded Atomic Actions [1, 2, 3]: Bluespec-style[4] modules, rules, interfaces, and methods.

The language is designed for the construction of **modules** that are *correct-by-construction composable*: validated smaller modules can be aggregated to form a larger validated module:

- Module interactions are performed with latency insensitive **method** calls, allowing methods to enforce invocation pre-conditions and transitive support for stalling.
- An **interface** is a named collection of method signatures and/or interface instances. Modules can declare multiple **interfaces**, giving each interface an explicit name, giving flexibility in coupling with other modules. Interfaces can be exported (defined in the module) or imported (used in the module, but defined externally), giving flexibility in algorithm representation.
- All state elements in the hardware netlist are explicit in the source code of the design. All module data is private, externally accessable only by interface method invocation.
- In AtomicC, user operations are written as SSA transactions, called **rules**. Since the compiler can statically analyze the read and write sets of the rule as well as the invocation conditions, it can guarantee that the generated code always executes

in a Sequentially Consistent (SC) manner: concurrent execution of transactions is guaranteed to be isolated. For conflicts, a fair schedule is generated.

- PSA support for assertions: assert, assume, restrict, cover.
  - Safety. A safety property asserts that nothing "bad" happens throughout execution
    - Refinement checking
    - Invariant: Basis. Inductive step.
    - Deadlock freedom
    - Reachability
  - Liveness. A liveness property asserts that something "good" eventually does happen.
    - Termination
    - Response
    - Request-response properties
    - real-time (system acts "in time")
  - Trace equivalence
- As with Connectal[5], AtomicC designs may include both hardware and software components, using interfaces to specify type safe communication. The AtomicC compiler generates the code and transactors to pass arguments between hardware and software.

1.4 Future work

Multiple clock domains

Need to describe multi-cycle rules and pipelining.

Need to describe joining rules

Need to have a way to support sequencing of operations

For NOC: allocate NOC like clock tree, after physical placement known. Can do method calls across NOC using credit.

- state-based: causal dependency between events
- property-based: global

Physical partitioning is used to separate design into separately synthesized pieces, connected using "long distance" signalling. Parallel synthesis; bitstreams combined.

Incremental place and route for changes:

- Automated physical partitioning of the design into locally clocked logic islands connected through delay tolerant network on chip
- Quick edit/compile/test cycles minimize the time between making a change and seeing the effect, allowing the developer to remain "in context"
- By physically partitioning the design into connected, placed regions, only the logic block that has been actually changed needs to be recompiled. Modular reuse through pre-placed modules

Timing closure – simplifying retiming:

- timing closure issues can be separated to 2 groups: interblock and intrablock. By using self-timed interfaces intrablock, long timing closure paths can be eliminated
- simple decoration of source expression trees can be used to generate fifo pipelined implementation of transactions.

Need to have a way to support model checking (say 'module B is a behavioral description of module A') Show example with diff eqn solver from Sharp thesis.

Coding FSMs as 'case' statements in Verilog (to integrate with verification tools).

2 Modules

The basic building block of AtomicC is the module declaration, made of 3 parts:

— Instantiation of state elements used by the module,

— Interface declarations for interacting with other modules,

— Rules, which group assignment statements and method invocations into atomic transactions.

2.1 Module interface definition

An **interface** is a named collection of method signatures and/or interface instances.

There are 2 types of methods:

— **Value method functions** allow *inspection* of module state elements.

— **Action method procedures** allow *modification* (including inspection) of state elements, can take parameters and do not have return values. A compiler generated **valid** signal indicates that the caller wishes to perform the method invocation.

Both value and action methods use a compiler generated **ready** signal to indicate when the callee is available and stall scheduling of the calling transaction until execution pre-conditions are statisfied.

[*Example*:

```
__interface EchoRequest {
    void say(__int(32) v);
    void say2(__int(16) a, __int(16) b);
};
```

— *end example*]

2.2 Module declaration and definition

Modules are independently compiled. Rule and interface method scheduling logic is generated as part of the generated module. Scheduling constraints (read set, write set and relation to other scheduled elements) are generated into a metadata file, allowing schedule consistency checks between modules to be verified by the linker.

The design is separated into modules that can export and import interfaces to other modules. Each source language module compiles into a single verilog module. Modules are independantly compiled, depending only on the interface definitions for referenced modules. Referencing modules do not depend on the internal implementation of referenced modules, even if they textually exist in the same compilation unit. Scheduling of rules in a module is performed "inside out", with the resulting schedule dependancies written to a metadata file during compilation.

Exported interfaces can be used in several ways:

— invoked directly by the instantiator of the module,

— forwarded transparently, becoming another exported interface of the instantiating module,

— ’connected’ to an ’interface reference’ of another module in the instantiating scope.

[*Example*:

```
__module Echo {
    EchoRequest      request;              // exported interface (defined by this module)
    EchoIndication   *indication;          // imported interface (defined by the instantiator of this module)
    bool busy;
    __int(32) itemSay;
    ...
    // implementation of method request.say(). Note the guard "if (!busy)".
    void request.say(__int(32) v) if(!busy) {
        itemSay = v;
        ...
    }
    void request.saw(__int(16) a, __int(16) b) if(!busy) {
        ...
    }
};
```

— *end example*]

To reference a module from a separate compilation unit, use "___emodule". External module definitions need only specify the exported/imported interfaces.

[*Example*:

```
__emodule EchoResponder {
    EchoIndication   indication;          // exported interface
};
```

— *end example*]

2.3 Guard clauses on module interface methods

1 Method definitions in ___module declarations have the form:

atomicc-method-definition:
  decl-specifier-seq$_{opt}$ *interface-qualifier-seq identifier parameters-and-qualifiers function-body*

interface-qualifier:
  identifier .

interface-qualifier-seq:
  interface-qualifier
  interface-qualifier-seq interface-qualifier

atomicc-function-body:
  if-guard$_{opt}$ *compound-statement*

if-guard:
  `if ( `*condition*` )`

Rules are only ready to fire if the rule’s guard is true and all the guards on methods invoked within the rule are also true.

```
    void request.say(__int(32) v) if(!busy) {
        itemSay = v;
        ...
    }
```

2.4 Connecting exported interfaces to imported references

The ___connect statement allows exported interface declarations to be connected with imported interface references between objects within a module declaration.

connect-declaration:

`__connect` *identifier* = *identifier* ;

[*Example*:

AtomicC example

```
__interface ExampleRequest {
    void say(__int(32) v);
};

__module A {
     ExampleRequest callIn;
};

__module B {
    ExampleRequest *callOut;
};

__module C {
    A consumer;
    B producer;
    __connect producer.callOut = consumer.callIn;
};
```

BSV example

```
BSV example
BSV example
BSV example
BSV example
BSV example
```

— *end example*]

Comparision with BSV:

— The declaration for 'A' is just like BSV. In BSV, the declaration for B requires the interface instance for 'callOut' be passed in as an interface parameter (forcing a textual ordering to the source code declaration sequence).

— In AtomicC, the interfaces are stitched together outside in any convenient sequence in a location where both the concrete instances for A and B are visible.

2.5 Exporting interfaces from contained objects

In a design, there are times when the engineer wishes to declare an object locally, but allow external modules to access specific interfaces of the local object. This is done by declaring an interface to the containing object of compatible type and just 'assigning' the local object's interface to it.

[*Example*:

```
__module CWrapper {
    A consumer;
    ExampleRequest request = A.callIn;
 };
```

— *end example*]

CWrapper just forwards the interface 'request' down into the instance 'consumer'.

2.6 Syntax extension to C++

atomicc-class-key:

```
__interface
__emodule
__module
```

3 Statements

AtomicC does not attempt to emulate the serialized execution behavior of all C++ constructs in hardware. Instead it uses a subset of the C++ language to specify code blocks that have at most one enabled assignment to any state element. This form is called *static single assignment* form, or SSA form[6]. In runtime execution, these assignments are all made in a single clock cycle, when the rule or method is enabled.

Constant bound "for" statements that can be fully unrolled are supported.

Since AtomicC does not generate logic to orchestrate sequential execution behavior from language constructs, traditional C++ statements with non-static control flow behavior are not supported.

Examples include:

— Non-constant bound "for" statements.

— "do", "while" statements

— Usages of "goto" that result in a cyclic directed graph of execution blocks

— Local method and function calls that are not inlinable at compilation time (for example, recursion is prohibited)

3.1 __rule

Rules specify a group of operations that must execute transactionally: when a rule's guard is satisfied, then it is ready to fire.

Module behaviorial statements are encapsulated into transactions (**rules**) following ACID semantics [7, 8]; all rules concurrently executed during a given clock cycle are *sequentially consistent* (SC) [9], guaranteeing each rule's execution is isolated [8, Sec. 7.1].

rule-statement:
 `__rule` *identifier* *if-guard*$_{opt}$ *compound-statement*

[*Example*:

```
__rule respond_rule if (responseAvail) {
    fifo->out.deq();
    ind->heard(fifo->out.first());
}
```

— *end example*]

3.2 Model details

C block semantics do not correctly process the 2 statements: a = b; b = a;. (binding of read values should occur at beginning of block, so that it is clear the 2nd assign refers to the 'previous' value). Thinking again: if we retain C semantics, we have: temp = a; a = b; b = temp;, which gives the correct value mapping.

To preserve the standard interpretation of C++ source code in methods and rules, a **modification in-private** [10, Sec. 3.2] execution model is used:

1. Wrap each rule/method with prelude code and postprocessing code
2. Prelude code: For each state element **A** in module, add the declaration of a shadow item:

   ```
   decltype(this->A) A = this->A;      // create shadow of state element
   ```

3. Postprocessing code: For each state element actually written during execution of the code block:

   ```
   this->A = A;                         // update state elements only at end of code block
   ```

   All assignments in the postprocessing section occur on a single clock cycle.

4 External interfacing

4.1 Exporting interfaces for use by software

In systems that have both hardware and software components, there is a need to marshall/demarshall parameterized method invocations across a hardware bus or network-on-chip (NOC). AtomicC provides this with my decorating the interface declarations with the keyword "___software".

The use of the ___software keyword causes the following to be performed:

— The generation of serialization/deserialization code for both software and hardware side modules to allow the method invocations to be performed in each direction

— The generation of header files allowing compilation of software modules that interface with the hardware

— Integration into a modified Connectal execution framework for the orchestration of requests.

[*Example*:

```
__module Echo {
    __software EchoRequest      request;             // exported interface
    __software EchoIndication   *indication;         // imported interface
    bool busy;
    __int(32) itemSay;
    ...
    // implementation of method request.say(). Note the guard "if (!busy)".
    void request.say(__int(32) v) if(!busy) {
        itemSay = v;
        ...
    }
    void request.saw(__int(16) a, __int(16) b) if(!busy) {
        ...
    }
};
```

— *end example*]

[*Example*:

```
#include "EchoIndication.h"   // Header file generated by AtomicC
#include "EchoRequest.h"      // Header file generated by AtomicC

class EchoIndication : public EchoIndicationWrapper
{
public:
    virtual void heard(uint32_t v) {
        // user code for handling indication
    }
    EchoIndication(unsigned int id, PortalTransportFunctions *item, void *param) :
        EchoIndicationWrapper(id, item, param) {}
};

int main(int argc, const char **argv)
{
    EchoIndication echoIndication(IfcNames_EchoIndicationH2S, &transportMux, &param);
    EchoRequestProxy echoRequestProxy(IfcNames_EchoRequestS2H, &transportMux, &param);

    // user code for sending requests
    echoRequestProxy->say(42);
}
```

— *end example*]

4.2 Interfacing with legacy Verilog modules

To reference a module in verilog, fields can be declared in ___interface items.

[*Example*:

```
__interface CNCONNECTNET2 {
    __input  __int(1)        IN1;
    __input  __int(1)        IN2;
    __output __int(1)        OUT1;
    __output __int(1)        OUT2;
};
__emodule CONNECTNET2 {
    CNCONNECTNET2 _;
};
```

— *end example*]

This will allow references/instantiation of an externally defined verilog module CONNECTNET2 that has 2 'input' ports, IN1 and IN2, as well as 2 'output' ports, OUT1 and OUT2.

4.2.1 Parameterized modules

Verilog modules that have module instantiation parameters can also be declared/referenced.

[*Example*:

```
__interface Mmcme2MMCME2_ADV {
    __parameter const char *  BANDWIDTH;
    __parameter float         CLKFBOUT_MULT_F;
    __input  __uint(1)        CLKFBIN;
    __output __uint(1)        CLKFBOUT;
    __output __uint(1)        CLKFBOUTB;
};
__emodule MMCME2_ADV {
    Mmcme2MMCME2_ADV _;
};
```

— *end example*]

This example can be instantiated as:

[*Example*:

```
__module Test {
    ...
    MMCME2_ADV#(BANDWIDTH="WIDE",CLKFBOUT_MULT_F=1.0) mmcm;
    ...
    Test() {
       __rule initRule {
           mmcm._.CLKFBIN = mmcm._.CLKFBOUT;
       }
    }
}
```

— *end example*]

4.2.2 Reference syntax

atomicc-method-declaration:
  attribute-specifier-seq$_{opt}$ *pin-type*$_{opt}$ *decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$;

pin-type:
  `__input`
  `__output`
  `__inout`
  `__parameter`

[*Example*:

```
__interface <interfaceName> {
     __input __uint(1) executeMethod;
     __input __uint(16) methodArgument;
     __output __uint(1) methodReady;
}
```

— *end example*]

For '___parameter' items, supported datatypes include: "const char *", "float", "int".

Factoring of interfaces into sub interfaces is also supported.

4.2.3 Clock/reset ports

Note that if interface port pins are declared in a module interface declaration, then CLK and nRST are _not_ automatically declared/instantiated. (Since the user needs the flexibility to not require them when interfacing with legacy code).

Note that this also allows arbitrary signals (like the output of clock generators) to be passed to modules as CLK/nRST signals. (For Atomicc generated modules, please note that the default clock/reset signals for a module will always have these names)

4.2.4 Import tooling

There is a tool to automate the creation of AtomicC header files from verilog source files. [*Example*:

```
atomiccImport -o MMCME2_ADV.h -C MMCME2_ADV -P Mmcme2 zynq.lib
atomiccImport -o VMMCME2_ADV.h -C MMCME2_ADV -P Mmcme2 MMCME2_ADV.v
```

— *end example*]

5 Compilation

AtomicC execution consists of the following phases:

- *compilation*
  - Parsing,semantic checks,
  - Static elaboration,

    During *static elaboration*, constructors are executed for statically declared data elements, allowing allocation of new instances of modules and parameterized rule creation. Any C++ constructs may be used, but the resulting netlist must only contain synthesizeable components.

    from Newton: Static elaboration does not change the semantics (types, evaluation rules) of the Regiment language, it merely opportunistically pushes evaluation forward into compile time.

    We need the ability to create state elements programmatically, improving efficiency of the design creation process.
  - Translation to an intermediate representation (IR),
  - Verilog netlist generation from the IR,
- *Linking*: Modules across the project are incrementally compiled and unit tested. As modules are reused in varying contexts, it is necessary to validate if restrictions need to be added for concurrent calls to conflicting methods in the module interface. This checking is done by a *linker*, ensuring that rule/method access across a set of Verilog output is free of inter-module schedule conflicts,
- *Logic synthesis, physical backend processing.* This is performed with existing backend tool flows
- *Formal verification* using the Coq Proof Assistant, etc.
- *Verification*: performed using existing backend tooling,
- *Hardware execution.*
- generate Verilog
- synthesize logic, get size
- pack blocks with sizes
- P & R

5.1 Linking of groups of modules

To verify that an instantiated group of modules has SC compliant execution characteristics, a linker is used to cross check information from the metadata files for each module.

Annex A Scheduling Algorithm

A.1 Goals

All changes to state elements are grouped into atomic transactions; any execution trace is expressable as a linear sequence of isolated transaction invocations (sequentially consistent (SC)). To execute transactions concurrently within a single cycle, we need to prove that concurrent execution behaves as though the transactions were scheduled in some (imputed) linear sequence.

Write operation effects become visible only after the end of the clock cycle, so there must be no read operations of the state element in successive transactions within the same cycle (since they will observe "old" data). We define a **write-after-read (WAR)** strict partial order [11, Sec. 3] over the set of rule/method nodes in a module. [12, Sec. 10.1.2] [13]

There is a WAR constraint between rules if they share an state element between a readSet and a writeSet. The constraint only actually exists when guards for the read and the write of the state element are true.

$$\begin{aligned}
&V \equiv \{allrules, allmethods\}\\
&\forall R1, R2 \ \in V, R1 \neq R2 :\\
&\quad arcCond(R1,R2) \equiv \bigvee_{\substack{\forall S \in readSet(R1),\\ S \in writeSet(R2)}} \begin{pmatrix} guard(R1) \wedge readGuard(R1,S)) \\ \wedge guard(R2) \wedge writeGuard(R2,S)) \end{pmatrix}
\end{aligned}$$

If the arcCond is not identically false, then the ordering constraint is added, defining a directed graph.

$$\forall R1, R2 \in V, \neg(arcCond(R1,R2) \rightarrow \perp) : R1 < R2$$

The set of concurrent transactions is SC iff this graph has no cycles for any guard conditions.

When detecting cycles in the graph, we calculate the boolean condition that the entire path of constraints actually occurs and that it is not identically false:

$$pathCond = \bigwedge_{\forall arc(R1,R2) \in path} arcCond(R1,R2)$$

If the pathCond is not identically false, the compiler breaks cycles by one of the following:

- Force rules to defer to methods. if cycle has some method M & some rule R, then the compiler can automatically rewrite the term $valid(R)$ to add a disjunction with the term $\neg valid(M)$, breaking the cycle.
- Fair scheduling between any 2 rules/methods in the cycle.

As an optimization, rules and methods that have some overlap of state element usage (*read set* and *write set*) are greedily partitioned into *schedule sets* and independently scheduled.

A simple example of a constraint graph is given in A.3, at the end of this document.

Since AtomicC performs scheduling analysis independantly for each declared module, external method invocation conflicts in rules cannot be validated. Schedule processing for external method calls is delayed until the "module group binding" stage of linking, where separately compiled AtomicC output is combined and verified for SC scheduling. Errors and conflicts detected at this stage must be repaired in the module source text and recompiled before proceeding.

A.2 Previous scheduling work

In Rule Composition[3], scheduling is formulated in terms of rule composition, leading to a succinct discussion of issues involved, including a concise description of the Esposito and Performance Guarantees schedulers. The resulting schedules are quite close to the user-specified scheduling in AtomicC. In contrast to AtomicC, the Bluespec kernel language they use for analysis also has a sequential composition operator, creating rules that execute for multiple clock cycles.

The Esposito Scheduler[14, 3], is the standard scheduler generation algorithm in the Bluespec Compiler. It uses a heuristic designed to produce a concrete total ordering of rules.

The Performance Guarantees scheduler[15] was proposed to address issues with intra-cycle data passing.

A.3 Scheduling example

A.3.1 Source program

```
__interface UserRequest {
    void say(__uint(32) va);
};

__module Order {
    UserRequest                      request;
    __uint(1) running;
    __uint(32) a, outA, outB, offset;
    void request.say(__uint(32) va) if (!running) {
        a = va;
        offset = 1;
        running = 1;
    }
    __rule A if (!__valid(request.say)) {
        outA = a + offset;
        if (running)
            a = a + 1;
    };
    __rule B if (!__valid(request.say)) {
        outB = a + offset;
        if (!running)
            a = 1;
    };
    __rule C if (!__valid(request.say)) {
        offset = offset + 1;
    };
};
```

A.3.2 Constraint graph

Sequentially consistent schedules are:

— when ’running == 1’: A -> B -> C
— when ’running == 0’: B -> A -> C

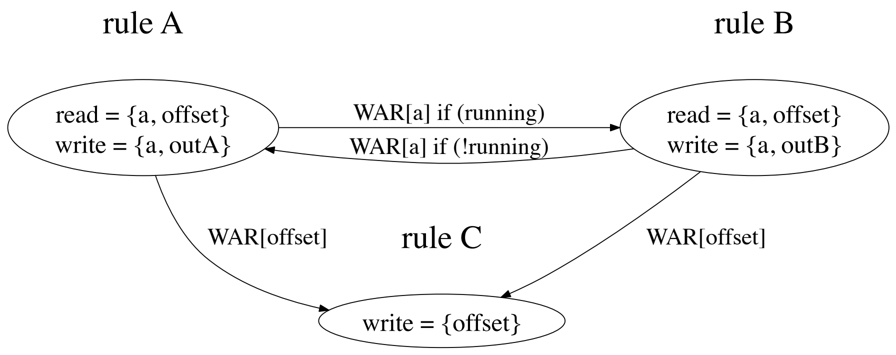

Figure 1 — Simple ordering example

A.4 Signalling

AtomicC uses **valid/ready** *hand-shaking signalling* [16, 17] to invoke action methods, giving both the invoker(master) and invokee(slave) the ability to control invocation execution timing. The master uses the **valid** signal of an action method to show when parameter data is available and the operation should be performed. The method invocation succeeds only when both **valid** and **ready** are HIGH in the same clock cycle.
In TRS notation[1, p. 22]:

$$\pi(M_i) \equiv ready(M_i) \ \wedge valid(M_i).$$

Annex B
Intermediate Representation

B.1 Module Definitions

atomicc-module-definition:
    module-type module-name `{` *module-body-list*$_{opt}$ `}`

module-type:
    `MODULE`
    `EMODULE`
    `INTERFACE`
    `STRUCT`
    `SERIALIZE`

module-body:
    module-body-definition
    module-body-definition module-body

module-body-definition:
    field-definition
    param-definition
    method-definition
    interface-definition
    interface-connect-definition

field-definition:
    `FIELD` *field-option-list*$_{opt}$ *field-type field-name*

param-definition:
    `PARAMS` *field-name* `<` *param-value* `>`

field-option-list:
    field-option
    field-option field-option-list

field-option:
    `/Ptr`
    `/shared`
    `/Count` *numeric-constant*
    verilog-field-options

verilog-field-options:
    `/parameter`
    `/input`
    `/output`
    `/inout`

interface-definition:
    `INTERFACE` *interface-option-list*$_{opt}$ *interface-type interface-name*

interface-connect-definition:
  `INTERFACECONNECT` *connect-option-list*$_{opt}$ *interface-name-l interface-name-r interface-type*

connect-option-list:
  connect-option
  connect-option connect-option-list

connect-option-list:
  `/Forward`

B.2 Method Definitions

method-definition:
  `METHOD` *method-name method-options* `{` *method-contents-list* `}`

method-options:
  method-flag-list$_{opt}$ *method-param-list*$_{opt}$ *method-return*$_{opt}$ *method-guard*$_{opt}$

method-flag-list:
  method-flag
  method-flag method-flag-list

method-flag:
  `(`*/Rule)*
  `(`*/Action)*

param-list:
  param-definition
  param-definition `,` *param-list*

param-definition:
  param-type param-name

method-return:
  return-type `=` *expression*

method-guard:
  `if` *boolean-expression*

method-contents-list:
  method-contents
  method-contents method-contents-list

method-contents:
  alloca-definition
  let-definition
  store-definition
  call-definition

alloca-definition:
  `ALLOCA` *alloca-type alloca-name*

let-definition:
  `LET` *let-type action-guard*$_{opt}$ `:` *let-target-name* `=` *source-expression*

store-definition:
  `STORE` *action-guard*$_{opt}$ `:` *store-target-name* `=` *source-expression*

call-definition:
  `CALL` *call-option*$_{opt}$ *action-guard*$_{opt}$ `:` *call-target-name call-param-list*$_{opt}$

action-guard:

    (*boolean-expression*)

Bibliography

[1] J. C. Hoe, "Operation-Centric Hardware Description and Synthesis," Ph.D. dissertation, MIT, Cambridge, MA, 2000.

[2] J. C. Hoe and Arvind, "Operation-Centric Hardware Description and Synthesis," *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 9, September 2004.

[3] N. Dave, Arvind, and M. Pellauer, "Scheduling as rule composition," in *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, ser. MEMOCODE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 51–60.

[4] Bluespec Inc., `http://www.bluespec.com`.

[5] M. King, J. Hicks, and J. Ankcorn, "Software-driven hardware development," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 13–22.

[6] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: ACM, 1988, pp. 1–11.

[7] R. S. Nikhil, "Formal specification of bsv's elaboration and dynamic semantics," https://github.com/rsnikhil/Bluespec_BSV_Formal_Semantics, 2015.

[8] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[9] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[10] A. Prinz and B. Thalheim, "Operational semantics of transactions," in *IN CRPITS'17: PROCEEDINGS OF THE FOURTEENTH AUSTRALASIAN DATABASE CONFERENCE ON DATABASE TECHNOLOGIES 2003*, 2003, pp. 169–179.

[11] H. W. Cain, M. H. Lipasti, and R. Nair, "Constraint graph analysis of multithreaded programs," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 4–.

[12] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[13] D. Rosenkrantz, R. Stearns, and P. Lewis II, "Consistency and serializability in concurrent database systems," *SIAM Journal on Computing*, vol. 13, no. 3, pp. 508–530, 1984.

[14] T. Esposito, M. Lis, R. Nanavati, J. Stoy, and J. Schwartz, "System and method for scheduling TRS rules," United States Patent US 133051-0001, February 2005.

[15] D. L. Rosenband and Arvind, "Hardware Synthesis from Guarded Atomic Actions with Performance Specifications," in *Proceedings of ICCAD'05*, San Jose, CA, 2005.

[16] C. Fletcher, "Eecs150: Interfaces: "fifo" (a.k.a. ready/valid)," https://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf, 2009.

[17] L. ARM, "Amba axi and ace protocol specification," https://developer.arm.com/docs/ihi0022/d/amba-axi-and-ace-protocol-specification-axi3-axi4-and-axi4-lite-ace-and-ace-lite, 2011.