1 Basic [atomicc.basic]

1.1 Introduction [atomicc.intro]

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 Bluespec-style[1, 2, 3] 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 with no impact on the correct operation of the component modules:

— Modules interactions are performed with Latency Insensitive (LI) **method** calls, automating the transitive effect of stalls.

— Module behaviorial statements are encapsulated into single cycle transactions (**rules**), replacing unreliable manual analysis of parallel operation with compile time hazard detection and runtime stall logic.

— Modules can export multiple, named Abstract Data Type (ADT) **interfaces**, promoting flexibility in module coupling. Program clarity and expressiveness is also improved through the explicit textual addition of interface name to references, rather than relying on multiple inheritance and usage casting as is traditional in C++.

— All state elements in the hardware netlist are explicit in the source code of the design.

These features support the efficient reuse of pre-compiled, incrementally validated libraries, improving productivity on large designs.

AtomicC does not attempt to emulate the behavior of all C++ constructs in hardware, instead uses a subset of the C++ language to specify behavioral assignments to state elements. Standard C++ templates allow parametric polymorphic reuse of **modules**. In AtomicC, all module data is private; an instantiator of a module can only interact with it via interface method invocations.

Like Connectal, AtomicC designs may include both hardware and software components, using interfaces to specify hardware/software communication in a type safe manner. The AtomicC compiler generates the code and transactors to pass arguments between hardware and software.

The AtomicC compiler generates a single Verilog module for each AtomicC module declared. Standard Verilog backend tools are then used to synthesize the resulting ASIC or FPGA. Integration with existing backend Verilog infrastructure including reuse of existing Verilog components.

The basic building block of AtomicC is the module declaration. Modules consist of 3 parts:

— Instantiation of state elements used by the module,

— Interface declarations for interacting from other modules,

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

1.2 Interfaces [atomicc.interface]

An interface definition is a list of method signatures, but no method bodies, defining an abstract data type(ADT) [4] view exported by the module. AtomicC follows in the tradition of the Go language[5], promoting the explicit declaration of **interfaces** for composition and coupling rather than using multiple inheritance to support heterogenous exported interface types. Module interfaces are named, allowing explicit denotation in the program text of which module ADTs are connected to and accessable from other state elements. In addition to defining and exporting interfaces, a module definition can also import interface references, exported from an externally defined module, providing flexibility in algorithmic description.

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.

1.3 Interface Methods [atomicc.method]

There are 2 types of methods:

— **Value method functions** provide read-only access to module state elements.

— **Action method procedures** perform transactions on state elements and do not have a return value. Parameters are used to pass transaction data to the instantiated module. A compiler generated **valid** signal indicates that the caller wishes to perform the method invocation.

Both types of methods use a compiler generated **ready** signal to indicate when the callee is available and to prevent scheduling of the calling transaction until all necessary elements are available.

Automate the handling of the transitive effect of stalls.

Latency-Insensitive Design assumes that modules are stallable. In future designs, a signal will need more than 10 clock cycles to traverse the entire chip area. Partition long wires into subsections, connected with relay stations. Can be implemented using *hand-shaking signalling* techniques. [6]

A patient system is one whose components are *stallable*. Two signals are latency equiv if they present the same sequence of informative events. Patient process: 1) intersection of 2 patient processes is a patient process, 2) give 2 pairs of latency equiv patient processes, their pairwise intersections are also latency equiv.

AtomicC uses a **valid/ready** handshake process[7, 8] to invoke action methods, giving both the invoker(master) and invokee(slave) the ability to control execution timing. The master uses the **valid** signal of the method to show when parameter data is available and the transaction should be performed The method invocation transaction succeeds only when both **valid** and **ready** are HIGH ($\pi(M_i) \equiv ready(M_i)$ && $valid(M_i)$).

1.4 Rules [atomicc.modrule]

Grouping operations on state elements into transactions (**rules**) enables compile time formal verification of the correctness of composite parallel execution of transactions in the synthesized logic. Transactions are units of consistency (preserving program invariants), following ACID

semantics. In AtomicC, all enabled statements in all modules execute on every clock cycle. The compiler synthesizes control signals, allowing rules to fire only when their dependent elements and referenced method invocations are ready.

Scheduling of rules guarantees 'Isolation', (also called 'consistency', 'concurrency control', 'serializability', or 'locking') [9, Sec. 7.1].

The AtomicC compiler validates that all rules executed during a given clock cycle are "equivalent" to some linear, atomic ordering ("sequentially consistent"(SC)) [10]. Even though all concurrent rules are executed during the same clock cycle, SC allows us to compute the outcome of each rule independently of any other rules that could be executing at the same time.

In contrast to software, which uses *dynamic allocation* and locking to guarantee isolation [9, p. 377] [11, Sec. 11.2], state elements accessed by an AtomicC transaction are known at compile time as well as the conditions when the transaction is performed. This allows *static allocation* [9, Sec. 7.3.1] of schedules.

Use of transactions supports compositions of modules that retain atomic properties. (usually atomic ops are non-modular, non-compositional [12] [13]) Transactions are compositional: small transactions can be glued together to form larger transactions[12]. Lock-based programs do not compose: correct fragments may fail when composed.

Transactional Datapath Specification. [14] To support overlapped execution of multiple operations, hazard detection and stall logic is introduced to maintain the correctness of operations in the overlapped executions. The single-cycle version serves as a function specification of the pipelined design, used in model checking. T-spec blocks use an established set of handshaking signals: ready, start, done. Each async block can be executed at most once by each transaction.

Time is defined relative to state transition and not the other way around. [15]

There is a graph-based definition of SC: define a dependency graph with node for each load/store. protocol is SC iff all traces have acyclic dependency directed graphs [16]. The transitive closure of these orders on the dependency graph nodes dictate the sequence in which each operation must *appear* to execute in order to be considered correct. Dependency constraints(RAW, WAR, WAW) vs consistency constraints (defined by memory consistency model. for example, if model is SC, then directed cycle cannot be placed in total order). WAR/WAW edges can be removed using register renaming. RAW edges can be removed using value prediction.

$O[i] \cap (I[j] \cup O[j]) = \varnothing, \forall i \neq j$ [9, Sec. 7.3]

1.5 Scheduling [atomicc.schedule]

Each rule has a set of state elements that it reads and another set of element that it writes. Value method invocations are treated as reads; action method invocations are treated as writes. For the execution of a group of rules to be considered to be SC, the following must be true:

— Atomic: All read and write operations for a given rule occur at the same time point in the sequence.

— Read-before-write: A rule that writes a state element must occur later in the sequence than any rules that read the same state element.

— Non-conflicting: A given state element cannot be written by more than one concurrently executable rule.

The compiler statically determines the 'read' and 'write' footprint for every rule and method [11, Sec. 10.1.2] [17].

— read set: $R_i.read$

— write set: $R_i.write$

— base set: $S(R_i) \equiv R_i.read \cup R_i.write$

[11, Sec. 10.1.2] [11, Sec. 11.1] 2 operations on same state element (x) are in conflict if one of them is a write. -> partial order)

The compiler and linker do not resolve SC conflicts automatically. If static compile analysis cannot prove that their execution conditions(guards) or effects(base sets) are always disjoint, then the source text of the program must specify the priority order of scheduling to resolve conflicts.

1.5.1 Algorithm [atomicc.schedalg]

— arc condition: $arcCond(uv) \equiv$
$fold(|, \{\forall E \in u.Write, E \in v.Read\text{: } (u.Write(E).cond \ \& \ v.Read(E).cond)\})$

```
WorkSet = { all rules and methods for module }
for r ∈ WorkSet do
    // Greedily group all rules/methods that have some overlap in read/write sets
    V = {r}
    V = {E|S(E) ∩ S(V) ≠ ∅}   // Create next schedule set
    WorkSet = WorkSet \ V

    // Create 'read-before-write' dependency digraph
    G = (V, E)
    where:
       V is a set of vertices
       E is a set of arcs == { uv | arcCond(uv) is not identically false }

    // Find loops
    L = loops in G

    for L ∈ G do
       if fold (&, { ∀uv ∈ L: arcCond(uv)}) is not identically false
          // 'break' loop L
          if loop has some method M & some rule R
              valid(R) &= ¬valid(M)
          else if source code has "priority R1 > R2" & R1 ∈ L & R2 ∈ L
              valid(R2) &= ¬valid(R1)
          else
              dependency digraph not acyclic, report error
```

1.5.2 Linking [atomicc.schedlink]

Without knowledge of the internals of a method, it must be assumed that all "action method" calls to a state element conflict. In addition, it must be assumed that all "value method" calls must preceed all "action method" calls in any clock cycle.

Since method/method conflicts in a module cannot be validated in the absence of information about their usage, this processing is delayed until the "module group binding" stage of linking. (It is not possible to resolve these conflicts standalone in the instantiated module.)

The linker cannot break any loops, but can only report on errors that are non-DAG.

Memoize checked results.

1.5.3 Future directions in scheduling [atomicc.schedfuture]

It is possible to create new rules that 'read' the 'commit value' for a state element. This would be done by synthesising a new 'combined rule' and scheduling it [18].

1.5.4 Previous scheduling work [atomicc.schedprev]

In the Esposito Scheduler[19], the Bluespec compiler creates a specific, linear schedule, adding one rule at a time. When a rule violates constraints from the previously scheduled rules, an error is issued and an automatic guard is synthesised to prevent the rule from executing on cycles when conflicting rules are executed.

Since the rules are added to the schedule approximately in the order they are found in the source program text, the resulting schedule (and which rules are inhibited) can be affected by source ordering and edits.

1.6 Compilation [atomicc.modcomp]

The AtomicC compiler generates a separate Verilog module definition for each source AtomicC module definition. This verilog source defines the state elements used in the design as well as their connections (netlist).

Modules independently compiled. Combined with "linking", which validates schedule using header files.

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

AtomicC execution consists of 3 phases:

— static elaboration: netlist generation,

— netlist compilation or implementation

— and runtime.

During netlist generation, modules are instantiated by executing their constructors. During this phase, any C++ constructs may be used, but the resulting netlist must only contain synthesizeable components.

During netlist compilation, the netlist is analyzed and translated to an intermediate representation and then to Verilog for simulation or synthesis. Alternate translations are possible: to native code via LLVM, to System C, to Gallina for formal verification with the Coq Proof Assistant, etc.

1.7 Future work [atomicc.modfuture]

Need to describe multi-cycle rules and pipelining.

Need to have a way to support sequencing of operations

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.

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.

Multiple clock domains

2 Classes [class]

2.1 Module declaration and definition [atomicc.module]

A module, defined using the keyword "___module", results in the generation of a corresponding verilog module in the compilation output file. It includes local state elements, interfaces exported, interfaces imported and rules for clustering operations into atomic transactions.

Modules are independently compiled, even if they exist in the same compilation unit. 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 between modules to be verified by the linker.

[*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*:

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

— *end example*]

2.2 Module interface definition [atomicc.interface]

An AtomicC interface is essentially an abstract class similar to a Java interface. All the methods are virtual and no default implementations are provided. AtomicC style uses composition of interfaces (using ___connect) rather than inheritance.

The ___interface keyword defines a list of methods that are exposed from an object that can be composed as a unit. Instead of using object inheritance to define reusable interfaces, they are defined/exported explicitly by objects, allowing fine-grained specification of interface method visibility.

Methods of a module are translated to value ports for passing the method arguments and a pair of handshaking ports used for scheduling method invocations.

References to an object can only be done through interface methods. State element declarations inside an object (member variables) are private.

[*Example*:

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

— *end example*]

2.3 Guard clauses on module interface methods [atomicc.guard]

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 [atomicc.connect]

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 [atomicc.export]

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.classsyn]

atomicc-class-key:

`__interface`
`__emodule`
`__module`

2.7 Exporting interfaces for use by software [atomicc.softif]

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*]

3 Statements [stmt.stmt]

3.1 ___rule [atomicc.rule]

Rules specify a group of operations that must execute as an atomicly. A rule operates transactionally: when a rule's guard and the guards of all of its method invocations are satisfied, then it is ready to fire. It will fire on a clock cycle when it does not conflict with any higher priority rule.

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 Restrictions on C++ statements [atomicc.nostmt]

Unlike the serialized execution model of C++, AtomicC supports a fully parallel, single cycle execution of rules which satisfy which are able to fire.

Since AtomicC does not generate any extra logic to support 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. Constant bound "for" statements that can be fully unrolled are supported.

— "do", "while" statements

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

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

4 Modularization [atomicc.modularization]

4.1 Independant compilation of modules [atomicc.independant]

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.

4.2 Execution control [atomicc.econtrol]

There are 2 common styles for communication of execution control information for a method:

— Asymmetric (ready/enable signalling) A method/rule is invoked by asserting the "enable" signal. This signal can only be asserted if the "ready" signal was valid, allowing the called module to restrict permissible execution sequences.

— Symmetric (ready/valid signalling) Both caller/callee have "able to be executed" signals. Execution is deemed to take place in each cycle where both "ready" (from the callee) and "valid" (from the caller) are asserted.

Bluespec uses the Asymmetric signalling style, collecting all scheduling control into a central location for analysis/generation. AtomicC uses the Symmetric signalling style, giving modules local control over their allowable execution patterns. Conflicts between local schedules for modules when they are connected together are detected by the linker.

4.3 Linking of groups of modules [atomicc.linker]

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.

4.4 Interfacing with verilog modules [atomicc.verilog]

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.

4.4.1 Parameterized modules [atomicc.param]

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*]

4.4.2 Reference syntax [atomicc.refsyntax]

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.

4.4.3 Clock/reset ports [atomicc.clockReset]

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)

4.4.4 Import tooling [atomicc.itool]

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*]

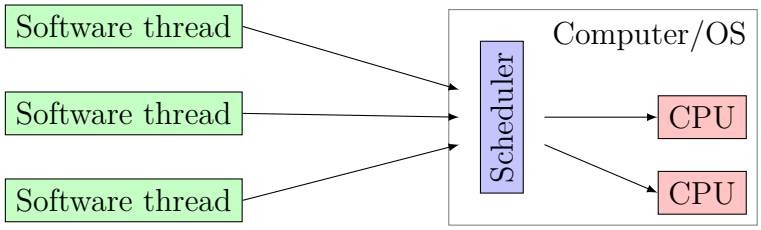
Annex A (informative) Introduction for Programmers [introProg]

1

A.1 Software [introProg.sw]

In software, the core model is the time-multiplexed execution of software threads by one or more central processing units (CPUs). Address arithmetic (pointers and indexing) prevents the compiler from statically determining read/write storage elements sets for a transaction. The programmer is responsible preventing the interleaved execution of multiple threads accessing a single storage element by decoration of the code with library calls to dynamically enforce mutual exclusion (mutex) regions.

In languages like Java, the programmer is able to decorate the storage element declarations to automate calling of these mutex operations.



.
.
.

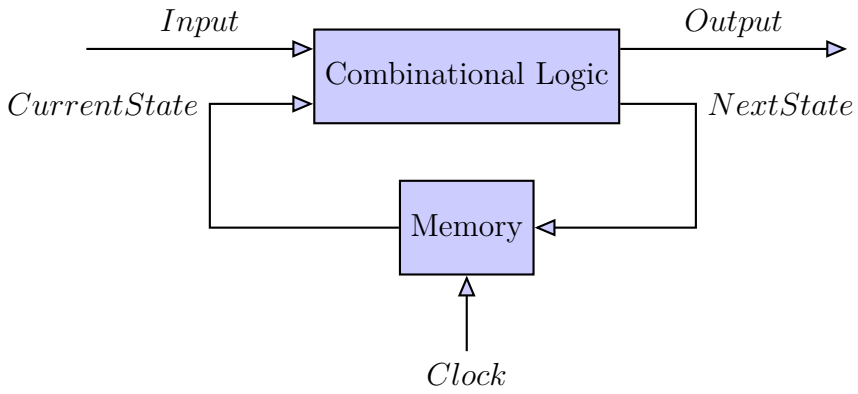
A.2 Hardware [introProg.hw]

In hardware, the core model is clock-based updates to state elements from a combinational logic net.

Combinational logic = current output is a boolean combination of current inputs

Sequential logic = combinational logic + memory elements (also called finite-state machine)

Synchronous logic = sequential logic + clock



From Hoe[2], the Term Rewriting System representation of this is:

s' = if $\pi(\mathbf{s})$ then $\delta(\mathbf{s})$ else **s**

Since all hardware elements are independent, all valid source lines in the program text are executed on every cycle. Access to state elements supports neither pointers nor indexing, allowing the compiler to statically determine parallel access transaction conflict sets, allowing the flagging of all combinations where correct operation cannot be guaranteed.

Bibliography

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

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

[3] 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.

[4] B. Liskov and S. Zilles, "Programming with abstract data types," in *SIGPLAN Notices*, 1974, pp. 50–59.

[5] Pike, Rob, "Less is exponentially more," https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html, 2012.

[6] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

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

[8] 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.

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

[10] 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.

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

[12] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2005, pp. 48–60.

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

[14] E. Nurvitadhi, J. C. Hoe, T. Kam, and S.-L. L. Lu, "Automatic pipelining from transactional datapath specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 3, pp. 441–454, 2011.

[15] A. Fox and N. A. Harman, "Algebraic models of correctness for abstract pipelines," *The Journal of Logic and Algebraic Programming*, vol. 57, no. 1-2, pp. 71–107, 2003.

[16] 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–.

[17] 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.

[18] D. L. Rosenband, "A Performance Driven Approach for Hardware Synthesis of Guarded Atomic Actions," Ph.D. dissertation, MIT, Cambridge, MA, 2005.

[19] 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.