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 support for Guarded Atomic Actions [1, 2]: Bluespec-style[3] modules, rules, interfaces, and methods. 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.

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 degradation of the component modules:

— Modules interactions are performed with Latency Insensitive (LI) [4] **method** calls, allowing methods to enforce invocation pre-conditions and transitive support for stalling.

— Module behaviorial statements are encapsulated into transactions (**rules**) following ACID semantics [5, 6]. The compiler synthesizes control signals, allowing rules to fire only when their referenced method invocations are ready.

  In AtomicC, all enabled rules in all modules execute on every clock cycle. The compiler validates that rules executed during a given clock cycle are "sequentially consistent"(SC) [7], guaranteeing each rule executes independently of any other rules executing at the same time (*isolation*[6, Sec. 7.1]).

— An **interface** is a named collection of method signatures, defining the behavior of an abstract data type(ADT) [8]. Modules can export and import multiple, named **interfaces**, giving flexibility in coupling with other modules and in algorithm expression.

— All state elements in the hardware netlist are explicit in the source code of the design. All module data is private to the module, accessable externally only by method invocation.

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

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. 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.

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 from other modules,

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

1.2 Interface Methods [atomicc.interface]

There are 2 types of methods:

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

— **Action method procedures** perform write operations on 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 prevent scheduling of the calling transaction until all necessary elements are available.

AtomicC uses a **valid/ready** *hand-shaking signalling* [9, 10] to invoke action methods, giving both the invoker(master) and invokee(slave) the ability to control transaction 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 transaction succeeds only when both **valid** and **ready** are HIGH:

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

1.2.1 Scheduling [atomicc.schedule]

To guarantee transaction isolation in software, *dynamic allocation*[6, p. 377] and locking[11, Sec. 11.2] are used. In contrast in AtomicC, the set of state elements accessed by an AtomicC transaction, the operations on these state elements and the boolean condition when the transaction is performed are all known at compile time. This allows *static allocation*[6, Sec. 7.3.1] of **schedules** (valid sequences of transaction execution) and compile time validation of SC. The scheduling algorithm is:

— Rules and methods that overlap usage of state elements (*read set* and *write set*[11, Sec. 10.1.2] [12]) are greedily gathered into *schedule sets* and will be independently scheduled (since there can be no interactions between sets).

— A *constraint graph* is a partially-ordered digraph modeling the dependencies within a *schedule set*: nodes in a constraint graph represent atomic rule and method instances, edges represent **write-after-read (WAR)** ordering dependency (for a specific storage element) between instances[13, Sec. 3]. Each edge has a symbolic boolean *edge condition* for when the the ordering dependency actually exists: the boolean condition when one rule actually reads a given state element and another rule actually writes it.

— The transitive closure of these orders on the constraint graph nodes dictate the **schedule** in which each rule must *appear* to execute in order to be considered SC [11, Sec. 11.1]. Of course, since all rules execute in a single cycle, "schedule" does not refer to an actual time sequenced evolution of state, but to a *conceptual* "sub-cycle" ordering.

— For each pair of nodes in the constraint digraph, we define the *node condition* between 2 nodes as the conjunction of the *edge conditions* of all the edges between them (i.e., the condition that *any* of the edges causes a dependency). For each cycle in the digraph, we define the *path condition* as the disjunction of the *node conditions* for all sequential pairs of nodes in the cycle (i.e., the condition that *all* the edges, hence the cycle exists).

— When the *path condition* is not identically false, a total ordering of the digraph can not be guaranteed and the *schedule set* is not SC. In this case, the compiler or linker reports an error, requiring resolution by the user.

The compiler can break the cycle under the following conditions:

— if cycle has some method M & some rule R, then rewrite the term $valid(R)$ to add a disjunction with the term $\neg valid(M)$

— if source code has "priority $R1 > R2$" and $R1$, $R2$ are in the cycle, then rewrite the term $valid(R2)$ to add a disjunction with the term $\neg valid(R1)$

Since method invocation conflicts in rules cannot be validated in the absence of concrete information about the constraints within the methods, this processing is delayed until the "module group binding" stage of linking, where separately compiled AtomicC output is combined and verified for SC scheduling.

1.2.2 Previous scheduling work [atomicc.schedprev]

In the Esposito Scheduler[14], 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.

1.3 Compilation [atomicc.modcomp]

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.

1.4 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 atomiclly. 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.

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.

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 CONNECT-NET2 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) Scheduling Example [lpmSchedule]

A.1 Source program [lpmExample.sw]

Example from: http://csg.csail.mit.edu/pubs/memos/Memo-500/memo500.pdf

```
typedef struct {
    int a;
    int b;
} ValuePair;

__interface LpmIndication {
    void heard(int meth, int v);
};

__interface LpmRequest {
    void say(int meth, int v);
};

__interface LpmMem {
    void req(ValuePair v);
    void resAccept(void);
    ValuePair resValue(void);
};

__module Lpm {
    Fifo1<ValuePair> inQ;
    Fifo1<ValuePair> fifo;
    Fifo1<ValuePair> outQ;
    LpmMemory        mem;
    int doneCount;
    void request.say(int meth, int v) {
        ValuePair temp;
        temp.a = meth;
        temp.b = v;
        inQ.in.enq(temp);
    }
    bool done() {
        doneCount++;
        return !(doneCount % 5);
    }
    LpmIndication *ind;
    LpmRequest request;
public:
    Lpm() {
            __rule recirc {
                ValuePair temp = fifo.out.first();
                ValuePair mtemp = mem.ifc.resValue();
                mem.ifc.resAccept();
                fifo.out.deq();
                fifo.in.enq(mtemp);
                mem.ifc.req(temp);
                };
            __rule exit_rule {
                ValuePair temp = fifo.out.first();
                ValuePair mtemp = mem.ifc.resValue();
                mem.ifc.resAccept();
                fifo.out.deq();
                outQ.in.enq(temp);
                };
            __rule enter {
                ValuePair temp = inQ.out.first();
                inQ.out.deq();
                fifo.in.enq(temp);
                mem.ifc.req(temp);
                };
            __rule respond {
                ValuePair temp = outQ.out.first();
                outQ.out.deq();
                ind->heard(temp.a, temp.b);
                };
            atomiccSchedulePriority("recirc", "enter;exit", 0);
    };
};

Lpm lpmbase;
```

A.2 Verilog output [lpmExample.verilog]

```
module Lpm (input wire CLK, input wire nRST,
    output wire ind$heard__ENA,
    output wire [31:0]ind$heard$meth,
    output wire [31:0]ind$heard$v,
    input wire ind$heard__RDY,
    input wire request$say__ENA,
    input wire [31:0]request$say$meth,
    input wire [31:0]request$say$v,
    output wire request$say__RDY);
    reg [31:0]doneCount;
    wire RULE$exit_rule__ENA;
    wire RULE$exit_rule__RDY;
    wire RULE$recirc__ENA;
    wire RULE$recirc__RDY;
    wire [95:0]fifo$in$enq$v;
    wire fifo$in$enq__ENA;
    wire fifo$in$enq__RDY;
    wire fifo$out$deq__ENA;
    wire fifo$out$deq__RDY;
    wire [95:0]fifo$out$first;
    wire fifo$out$first__RDY;
    wire inQ$in$enq__RDY;
    wire inQ$out$deq__RDY;
    wire [95:0]inQ$out$first;
    wire inQ$out$first__RDY;
    wire [95:0]mem$ifc$req$v;
    wire mem$ifc$req__ENA;
    wire mem$ifc$req__RDY;
    wire mem$ifc$resAccept__ENA;
    wire mem$ifc$resAccept__RDY;
    wire [95:0]mem$ifc$resValue;
    wire mem$ifc$resValue__RDY;
    wire outQ$in$enq__RDY;
    wire outQ$out$deq__RDY;
    wire [95:0]outQ$out$first;
    wire outQ$out$first__RDY;
    wire [31:0]request$say__ENA$temp$c;
    Fifo1Base#(96) inQ (.CLK(CLK), .nRST(nRST),
        .in$enq__ENA(request$say__ENA),
        .in$enq$v({ request$say__ENA$temp$c , request$say$v , request$say$meth }),
        .in$enq__RDY(inQ$in$enq__RDY),
        .out$deq__ENA(inQ$out$first__RDY & fifo$in$enq__RDY & mem$ifc$req__RDY),
        .out$deq__RDY(inQ$out$deq__RDY),
        .out$first(inQ$out$first),
        .out$first__RDY(inQ$out$first__RDY));
    Fifo2 fifo (.CLK(CLK), .nRST(nRST),
        .in$enq__ENA(fifo$in$enq__ENA),
        .in$enq$v(fifo$in$enq$v),
        .in$enq__RDY(fifo$in$enq__RDY),
        .out$deq__ENA(fifo$out$deq__ENA),
        .out$deq__RDY(fifo$out$deq__RDY),
        .out$first(fifo$out$first),
        .out$first__RDY(fifo$out$first__RDY));
    Fifo1Base#(96) outQ (.CLK(CLK), .nRST(nRST),
        .in$enq__ENA(fifo$out$first__RDY & mem$ifc$resValue__RDY & mem$ifc$resAccept__RDY & fifo$out$deq__RDY),
        .in$enq$v({ fifo$out$first[ 95 : 64 ] , fifo$out$first[ 63 : 32 ] , fifo$out$first[ 31 : 0 ] }),
        .in$enq__RDY(outQ$in$enq__RDY),
        .out$deq__ENA(outQ$out$first__RDY & ind$heard__RDY),
        .out$deq__RDY(outQ$out$deq__RDY),
        .out$first(outQ$out$first),
        .out$first__RDY(outQ$out$first__RDY));
    LpmMemory mem (.CLK(CLK), .nRST(nRST),
        .ifc$req__ENA(mem$ifc$req__ENA),
        .ifc$req$v(mem$ifc$req$v),
        .ifc$req__RDY(mem$ifc$req__RDY),
        .ifc$resAccept__ENA(mem$ifc$resAccept__ENA),
        .ifc$resAccept__RDY(mem$ifc$resAccept__RDY),
        .ifc$resValue(mem$ifc$resValue),
        .ifc$resValue__RDY(mem$ifc$resValue__RDY));
    assign fifo$in$enq$v = ( ( inQ$out$first__RDY & inQ$out$deq__RDY & fifo$in$enq__RDY & mem$ifc$req__RDY ) ? { inQ$out$fir
    assign fifo$in$enq__ENA = ( ( inQ$out$first__RDY & inQ$out$deq__RDY ) | ( fifo$out$first__RDY & mem$ifc$resValue__RDY &
    assign fifo$out$deq__ENA = ( fifo$out$first__RDY & mem$ifc$resValue__RDY & mem$ifc$resAccept__RDY & outQ$in$enq__RDY ) |
    assign ind$heard$meth = outQ$out$first[ 31 : 0 ];
    assign ind$heard$v = outQ$out$first[ 63 : 32 ];
    assign ind$heard__ENA = outQ$out$first__RDY & outQ$out$deq__RDY;
    assign mem$ifc$req$v = ( ( inQ$out$first__RDY & inQ$out$deq__RDY & fifo$in$enq__RDY & mem$ifc$req__RDY ) ? { inQ$out$fir
    assign mem$ifc$req__ENA = ( ( inQ$out$first__RDY & inQ$out$deq__RDY ) | ( fifo$out$first__RDY & mem$ifc$resValue__RDY &
    assign mem$ifc$resAccept__ENA = ( fifo$out$first__RDY & mem$ifc$resValue__RDY & fifo$out$deq__RDY & outQ$in$enq__RDY ) |
    assign request$say__RDY = inQ$in$enq__RDY;
    // Extra assigments, not to output wires
    assign RULE$exit_rule__ENA = fifo$out$first__RDY & mem$ifc$resValue__RDY & mem$ifc$resAccept__RDY & fifo$out$deq__RDY &
    assign RULE$exit_rule__RDY = fifo$out$first__RDY & mem$ifc$resValue__RDY & mem$ifc$resAccept__RDY & fifo$out$deq__RDY &
    assign RULE$recirc__ENA = fifo$out$first__RDY & mem$ifc$resValue__RDY & mem$ifc$resAccept__RDY & fifo$out$deq__RDY & fif
    assign RULE$recirc__RDY = fifo$out$first__RDY & mem$ifc$resValue__RDY & mem$ifc$resAccept__RDY & fifo$out$deq__RDY & fif

    always ( posedge CLK) beginif (!nRST) begindoneCount <= 0;end // nRSTelse beginendend // always  (posedge CLK)
endmodule
```

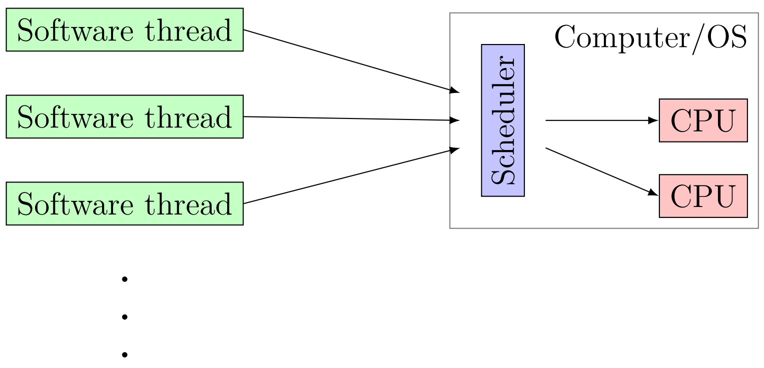
Annex B (informative) Introduction for Programmers [introProg]

1

B.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.



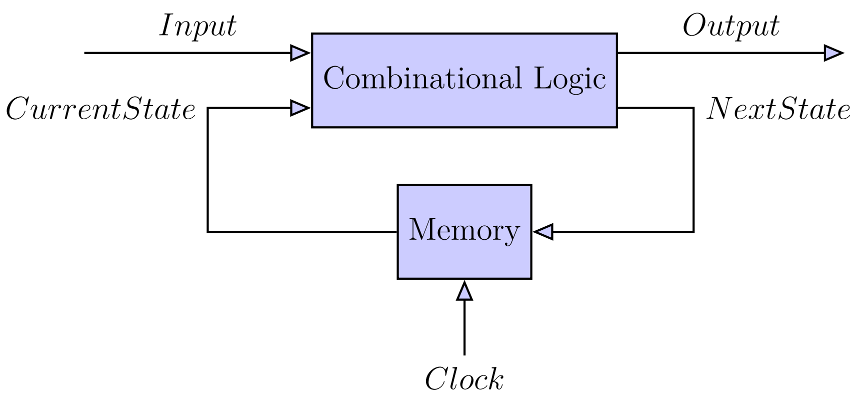
B.2 Hardware [introProg.hw]

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

[4] 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.

[5] R. S. Nikhil, “Formal specification of bsv’s elaboration and dynamic semantics,” https://github.com/rsnikhil/Bluespec_BSV_Formal_Semantics, 2015.

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

[7] 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.

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

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

[10] 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.

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

[12] 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.

[13] 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–.

[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.