Programming SCORE

Eylon Caspi

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2005-25
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-25.html

December 16, 2005

Programming SCORE

Eylon Caspi
eylon@cs.berkeley.edu

BRASS research group
http://brass.cs.berkeley.edu/

Original April 19, 2000
Revised Feb. 4, 2005

1 Introduction

SCORE (*Stream Computation Organized for Reconfigurable Execution*) is a computation and programming model designed to exploit dynamically reconfigurable hardware transparently using compiler and operating system support. SCORE targets a hardware environment featuring a conventional microprocessor and a reconfigurable coprocessor (*e.g.* FPGA). The compute model is a process network variant, inspired by dynamic data-flow [BUCK93] and Petri nets [MURA89]. Programming in SCORE is done using a combination of C++ and a custom language, TDF.

This document describes the computational model, languages, and programming disciplines of SCORE. Section 2 gives an overview of the major components of the computational model. Section 3 describes the TDF language for specifying SCORE operators. Section 4 describes the C/C++ library API for SCORE and how it interacts with TDF.

This document uses typographic conventions of `monospace` for code fragments and *italics* for implementation notes.

2 SCORE Overview

Computation in SCORE is based on three fundamental mechanisms: streams, operators, and segments. Streams represent communication using sequences of tokens. Operators represent computation, triggered by arrival of tokens. Segments represent blocks of memory accessed via streams.

Operators may run on the processor or on the reconfigurable array and may communicate across the processor-array boundary. Behind the scenes, an optimizing compiler partitions array-based operators into hardware pages, and an operating system scheduling service loads and executes them as necessary.

Simple stream processing operators slated for acceleration on reconfigurable resources are written in the custom language TDF (*Task Description Format*). Operators may also be written in C++, in which case they will run only on the processor and have full access to library routines. C++ operators may dynamically create and connect other operators.

2.1 Streams

Streams are the basic communication mechanism in SCORE. They are the edges of data-flow graphs, each transmitting a sequence of data tokens between a pair of operators. Logically, a stream is a typed, first-in-first-out (FIFO) queue, having unbounded capacity, and having a token type extended with an out-of-band end-of-stream ("eos") value. The FIFO discipline allows stream contents to be stored, delayed, or transferred by any software or hardware mechanism. For instance, a stream whose consuming operator is not presently running may have its tokens diverted into a memory buffer. The actual capacity of stream buffers is implementation dependent and unknown to operators, but it should support the abstraction of unbounded capacity.

In TDF, streams are specified as formal arguments of operators. An inlined call to an operator specifies static stream connections by the actual arguments. A state-machine notation also exists for specifying actions to take in response to arrival of input tokens.

In C++, stream connections are specified dynamically during instantiation of operators. Token emission and reception are done using library calls or macros which act on `ScoreStream` objects.

2.2 Operators

SCORE operators encapsulate computation. They are the nodes of data-flow graphs, receiving input tokens and emitting output tokens via streams. Operators are typically event-driven, taking action ("firing") in response to token arrival. Operators with no inputs are also permitted, *e.g.* a random number generator. Under data-flow semantics, an operator may fire when it has sufficient input tokens and space for output tokens.

SCORE operators can be parameterized by special "param" inputs. Such inputs are bound at compile-time or when "instantiating" an operator at run-time. They may be used for type polymorphism (*e.g.* as the bit-width of an input or output) or as late-bound data (*e.g.* an initial value or operating mode). Multiple implementations of each operator may co-exist, each specialized for a different parameterization, and targetting the processor or the array. Choosing which implementation to use at which time is the job of the compiler and operating system.

SCORE operators are bound tightly to their I/O streams and live only as long as those streams are open. Unless coded to do otherwise, an operator will flush its internal pipeline and terminate upon receiving an "eos" token on any input stream.

2.3 Segments

SCORE segments represent memory blocks which may be accessed by operators via streams. A number of streaming interfaces are available, supporting sequential as well as random access, with read-only, write-only, or read-write restrictions. A segment's contents may also be accessed from C++ using traditional array access. Segments are typed as an array of identically-typed elements.

A segment is always bound to one owner, that owner being a SCORE operator or the processor in general. The owner has exclusive access to the segment. Any operator or processor code attempting to access a segment owned by another owner is made to stall until the segment becomes available. When a segment owner is destroyed (*e.g.* an operator terminates), the segment reverts to its previous owner.

A segment bound to a behavioral or compositional operator is given a particular streaming interface by encapsulating it in a segment operator and connecting its streams to the owning operator. A segment bound to an embedded C++ operator may be accessed by the operator's code as a C++ array. When not bound to an operator, a segment is owned by the processor and may be accessed by user code as an array. Such user code, like any segment owner, must be made to stall when accessing a segment presently owned by an operator.

3 The TDF Language

The SCORE Task Description Format (TDF) is a language for specifying compute operators at the register-transfer level (RTL). Operators can be defined either as finite state machines, as static compositions of other operators, or as embedded C++ code. The former two describe static data-flow graphs, whereas the latter may dynamically create data-flow graphs. The language is strongly-typed with support for explicit bit-widths and width-based polymorphism. An exception mechanism is defined to support fatal and resumable exceptions. Multiple implementations of each TDF operator may coexist, being automatically managed by operating system runtime support (*e.g.* a software version, parameterized FPGA versions, *etc.*).

This section describes the TDF language, first by example, and then in detail. The detailed discussion is structured around the components of the language grammar in Backus-Naur form.

3.1 A Simple Example

We present a simple example of a TDF behavioral operator to implement the canonical "select" actor of boolean dataflow [BUCK93]. The "select" actor merges two data streams into one, the order of tokens being specified by a boolean control stream. When the control stream contains `true`, the actor passes a token from the true-side input to the output, and when the control stream contains `false`, the actor passes a token from the false-side input to the output. A TDF implementation of "select" follows.

```
select (input boolean s,      input unsigned[8] t,
        input unsigned[8] f, output unsigned[8] o)
{
  state get_s (s) : if (s==true) goto get_t; else goto get_f;
  state get_t (t) : o=t; goto get_s;
  state get_f (f) : o=f; goto get_s;
}
```

The TDF operator begins with a declaration that specifies its name and I/O streams. Each stream is given a direction, either `input` or `output`, and a token data type. Type `unsigned[8]` means an 8-bit unsigned integer. For "select," stream `s` is the select control input, `t` is the true-side data input, `f` is the false-side data input, and `o` is the data output.

The body of the operator is structured as an extended finite state machine, where each named state specifies desired inputs and an action using C-like statements. TDF execution semantics dictate that on entry to a given state, the operator issues blocking reads to the input streams specified in its state *signature*. Once enough tokens are available on those input streams, and once space is available on the output streams, the operator may *fire*. A firing entails consuming the tokens specified in the state signature, executing the state action, and transitioning to a next state.

In the example, "select" is implemented as an infinite loop of: consume `s`, evaluate `s`, then either consume and re-emit `t` or consume and re-emit `f`. This sequential, stateful implementation is required because TDF does not support value matching in a firing guard (except for "eos" end-of-stream). Instead, the value of `s` must be evaluated in a state action, after consuming.

Input and output streams are used syntactically like variables inside a state action. Use of an input stream variable refers to the most recently consumed data value from that stream. Assignment to an output stream variable emits a token to that stream.

3.2 Notation

We describe the TDF grammar using *Backus-Naur Form* (BNF), a common notation originally proposed by John Backus and Peter Naur for describing Algol 60 unambiguously [NAUR63]. BNF describes a language's syntax using rewrite rules of the form:

$$<\textit{non-terminal}> ::= <\textit{expanded-form}>$$

Such rule means that any instance of the left-side <*non-terminal*> can be expanded into a form from the right-side. Repeated rewrites can transform a single, top-level non-terminal into an entire program. We use the following notation conventions for right-sides of rules:

<*x*>	denotes a non-terminal, defined on the left-side of some rule
"x"	denotes a terminal (literal)
[x]	denotes an optional item
{x}	denotes an item repeated 0 or more times
(...)	groups items
|	separates mutually-exclusive choices

The remainder of the TDF language description is organized around the BNF grammar, providing explanations for each rewrite rule.

3.3 Types

TDF has strong typing with support for explicit bit-widths. Parameterized-width types enable width-based polymorphism. Width parameters may be bound at compile-time or at operator construction time.

$$
\begin{aligned}
\text{<}\textit{type}\text{>} ::= {} & \text{“boolean” [<}\textit{arraySize}\text{>]} && (1)\\
| {} & \text{<}\textit{sign}\text{> [<}\textit{arraySize}\text{>] “[”<}\textit{expr}\text{>“]”} && (2)\\
| {} & \text{<}\textit{sign}\text{> [<}\textit{arraySize}\text{>] “[”<}\textit{expr}\text{>“.”<}\textit{expr}\text{>“]”} && (3)\\
\text{<}\textit{sign}\text{>} ::= {} & \text{“unsigned” | “signed”} && (4)\\
\text{<}\textit{arraySize}\text{>} ::= {} & \text{“[”<}\textit{expr}\text{>“]”} && (5)
\end{aligned}
$$

Boolean types (rule (1)) are single-bit. Integer types (rule (2)) are specified using an integer width expression. Fixed-point types (rule (3)) are specified using a fractional width expression “$i.f$” with widths i for the integer part and f for the fraction part. Zero-width types are allowed, but they can represent only one value, namely unsigned zero.[1]

The “unsigned” qualifier denotes a simple bit-vector representation. The “signed” qualifier denotes a two’s complement representation whose specified bit-width includes a leading sign-bit. For fixed-point types, a sign-bit is needed only in the integer part. *Implementation note: Support for fixed-point types is presently incomplete.*

A type without the optional <*arraySize*> specifier (rule (5)) denotes a single word. A type with <*arraySize*> denotes an array of words. The expression in <*arraySize*> is a positive integer specifying the number words in the array. The storage implementation for arrays (*e.g.* DRAM block, registers, look-up table in logic, *etc.*) is determined by the compiler. *Implementation note: At present, all arrays are implemented as segments and are accessed by streams, except for read-only arrays in the Verilog back-end.*

Examples:

`boolean`	boolean
`signed[8]`	8-bit word in two’s complement
`unsigned[16.16]`	fixed-point with 16 integer bits, 16 fraction bits
`unsigned[1024][8]`	Array of 1024 bytes

3.4 Constants and Identifiers

$$
\begin{aligned}
\text{<}\textit{const}\text{>} ::= {} & \text{“true” | “false”} && (6)\\
| {} & \text{<}\textit{integer}\text{>} && (7)\\
| {} & \text{<}\textit{integer}\text{>“.”<}\textit{integer}\text{>} && (8)
\end{aligned}
$$

[1] Zero-width types are useful for synchronizing operators through streams, *e.g.* for exceptions, where a token carries no useful data, only a “presence” bit.

TDF supports three kinds of constants: boolean, integer, and fixed-point. Boolean constants (rule (6)) may be "true" or "false". Integer constants (rule (7)) represent unsigned words whose bit-width is the minimum required to represent their value. Fixed-point constants (rule (8)), formed as "*i.f*", are of minimum bit-width provided that the fraction's base-conversion to binary is exact. If a fixed-point constant is specified in decimal with no exact conversion possible, the constant is rounded to three bits per fractional digit. Signed constants are interpreted as a unary minus applied to an unsigned constant, obeying the type-upgrade rules of the unary minus operator (namely widening to add a sign bit). Numerical constants may be entered in decimal, hexadecimal (preceded by `0x`), octal (preceded by `0`), or binary (preceded by `0b`).

Examples:

`10`, `0b10`, `0x10`, `010`	integers: decimal, binary, hexadecimal, octal
`0b1.1`	binary fixed-point (exact) equivalent to `1.5`
`0xd.ac`	hexadecimal fixed-point (exact) equivalent to `0b1101.101011` and `13.671875`
`3.14`	decimal fixed-point (approximated) equivalent to `0b11.001001`

$$<id> ::= (<letter> \mid \text{“\_”})\{<letter> \mid \text{“\_”} \mid <digit>\} \tag{9}$$

Identifiers in TDF obey C naming conventions. They are alphanumeric, case-sensitive, may include underscores, and must begin with a letter or underscore.

3.5 Operator Suites

$$<suite> ::= \{<operator>\} \tag{10}$$

A SCORE application is described by a *suite* of TDF operators (and a C++ entry point not discussed here). A suite is processed initially by the `tdfc` compiler and subsequently by a C++ compiler and/or FPGA back-end tools. This is the top-level rule of the BNF grammar.

3.6 Operators

$$<operator> ::= [<type>]\ <id>\text{“(”}<ioDeclList>\text{“)”}\ <opBody> \tag{11}$$

$$<ioDeclList> ::= \{<ioDecl>\ \{\text{“,”}<ioDecl>\}\} \tag{12}$$

$$<ioDecl> ::= <ioKind>\ <type>\ <id> \tag{13}$$

$$<ioKind> ::= \text{“input”} \mid \text{“output”} \mid \text{“param”} \tag{14}$$

Operators are prototyped to specify their typed I/O streams and parameters (rule (11)). An operator may have arbitrarily many "input" and "output" streams, as well as a single "return" stream which is an output stream bearing the same name as the operator. Parameter inputs ("param") are bound once for the lifetime of the operator. Scalar parameters are constants, and may be used anywhere a constant can, including to specify the bit-width of I/O streams and local variables. Array parameters are segments,

passed in by value-return. The type of a "param" input cannot be parameterized by other "param" inputs.

Examples:

`doBit(boolean b)`	inputs one bit at a time, no output
`unsigned[9] add(unsigned[8] a, unsigned[8] b)`	8-bit adder: inputs two 8-bit words, outputs 9-bit word
`unsigned[n+1] add(unsigned[n] a, unsigned[n] b, param unsigned[8] n)`	same as above, bit-widths parameterized by `n`
`unsigned[16] random()`	random number generator: no input, emits 16-bit words

$$\begin{aligned} \text{<}opBody\text{>} ::= & \text{“\{”\{<}typeDecl\text{>}\} \{<}state\text{>}\} \{<}operator\text{>}\}\text{“\}”} && (15)\\ | & \text{“\{”\{<}typeDecl\text{>}\} \{<}callOrAssign\text{>}\} \{<}operator\text{>}\}\text{“\}”} && (16)\\ | & \text{“\%\{”<}C{+}{+}\ code\text{>“\%\}”} && (17) \end{aligned}$$

TDF supports three forms of user-defined operators: behavioral, compositional, and embedded C++. The three forms are introduced here and discussed in detail in subsequent sections.

behavioral operators (rule (15)) are finite state machines whose state actions fire in response to the arrival of input tokens. Variables declared in the <*typeDecl*> section denote registers whose value persists across states (akin to C static variables).

compositional operators (rule (16)) are used to connect other operators together and have no logic or storage of their own. Variables declared in the <*typeDecl*> section are used to name streams between operators.

embedded C++ operators (rule (17)) encapsulate arbitrary, processor-side code blocks which communicate via TDF streams. *Implementation note: Embedded C++ operators are presently not implemented.*

Behavioral and compositional operators may contain other "local" operators ({<*operator*>} in rules (15) and (16)). Local operators are lexically scoped to their enclosing operator and cannot be referenced outside of it. Local embedded C++ operators are typically used to specify exceptions. *Implementation note: Locally scoped operators are presently not implemented.*

In addition to the three forms of user-defined operators, TDF supports a number of built-in operators.

copy operators duplicate one stream into many, to provide fan-out.

segment operators provide a streaming interface for memory segments.

3.7 Variable Declarations, Behavioral

<*typeDecl*> ::= <*type*> <*id*> [“=” <*expr*>] “;” (18)

Typed variables may be declared inside behavioral operators to name and/or store computed quantities. Each variable is lexically scoped to the smallest statement block or operator in which it is declared. Variables have the conventional read/write semantics of imperative languages, so that a value assigned to a variable is visible immediately to all subsequent uses of the variable in its scope, until the variable is assigned again. It is an error to read a variable before it is assigned a value. A variable declaration may include an initial value.

Examples:

`boolean b;`	uninitialized boolean
`signed[4.4] pi=22/7;`	fixed-point initialized by a constant expression
`unsigned rom[4][8]={1,2,3,4};`	array of 4 8-bit words

There are two classes of declared variables: register and temporary.

register variables are declared at the top of behavioral operators. They denote register storage whose value persists across state transitions.

temporary variables are declared inside states. They are used to name computed expressions and/or constants inside a lexical scope, without denoting storage.

In addition, the formal parameters from an operator's declaration (rule (12)) behave like special variables. The <*ioKind*> qualifiers (rule (14)) for formal parameters may be regarded as additional storage classes for variables.

“input” variables denote input streams, and are read-only. Their value is assigned implicitly during firing. See rule (73) for use.

“output” variables denote output streams, and are write-only. Their assignment denotes emitting a token to the stream. *Implementation note: An output stream variable may be assigned only once per firing, denoting a single token emission per firing.*

“param” variables are passed in from a caller. Scalar “param” variables are read-only constants. Array “param” variables are read-write, denoting segments passed in by value-return.

3.8 Variable Declarations, Compositional

<*typeDecl*> ::= <*type*> <*id*> [“(” <*expr*> “)”] [“=” <*expr*>] “;” (19)

Typed variables may be declared inside compositional operators to name streams that connect operators together. The declaration may include a parenthesized *depth hint* to denote a desired minimum buffer size, but its value is considered an implementation hint

and not part of the program semantics. An initial value assignment denotes initial tokens in the stream buffer. *Implementation note: Initial values for streams are presently not supported. Initial tokens must instead be emitted by the stream producer.*

A stream variable may be assigned once, to denote connection to a single producer. A stream variable may be read one or more times, to denote connection to one or more consumers. If multiple consumers are connected, each one receives a copy of all tokens from the producer. *Implementation note: Stream fan-out to multiple consumers is presently implemented by inferring a built-in* `copy` *operator.*

In addition, the formal parameters from an operator's declaration (rule (12)) behave syntactically like stream variables.

3.9 Behavioral Operators (Streaming Finite State Machines)

$$\begin{aligned}
\text{<}opBody\text{>} &::= \text{“\{”\{<}typeDecl\text{>}\} \{<}state\text{>}\} \{<}operator\text{>}\}\text{“\}”} && (15)\\
\text{<}state\text{>} &::= \text{“state” <}id\text{>“(”<}inputList\text{>“)” “:”\{<}stmt\text{>}\} && (20)\\
\text{<}inputList\text{>} &::= \{\text{<}input\text{>} \{\text{“,”<}input\text{>}\}\} && (21)\\
\text{<}input\text{>} &::= \text{<}id\text{>[“\#”<}expr\text{>]} && (22)\\
&\quad | \ \text{“eos” “(”<}id\text{>“)”} && (23)
\end{aligned}$$

Behavioral operators are extended finite state machines which steps in response to the arrival of input tokens. The first state listed in a TDF program is taken to be the initial state. The firing guard for a given state is denoted by its *state signature* (rule (20)) which lists how many tokens are required from which input streams (rules (22)-(23)). The optional token count ([“#”<*expr*>]) is taken to be one if omitted. *Implementation note: Input token counts greater than one are presently not supported.* A state signature may specify the receipt of an end-of-stream (“eos”) token rather than a data token, which denotes the closing of an input stream. The arrival of an unexpected “eos” token during a state whose signature demands a data token causes the operator to terminate cleanly, by emitting “eos” to its output streams. Note that back-end tools may generate multiple signatures per state for reasons of efficiency or correctness, but the programmer may not do so.

A *firing*, or stepping of the operator, may occur when the following conditions hold:

1. The state machine is in the appropriate state
2. Sufficient input tokens are available to “match” the state signature
3. Sufficient buffer space is available to emit the state's output tokens

A *firing* is performed as follows:

1. Consume the tokens specified in the state signature
2. Evaluate the state action
3. Transition to a next state

A state action is a sequence of statements. Statements are detailed in Section 3.14.

Examples:

`state s():`	fires spontaneously
`state s(a):`	fires on each `a` token
`state s(a,b#2):`	fires on single `a` token and pair of `b` tokens
`state s(a,eos(b)):`	fires on single `a` token and closing of stream `b`

3.10 Compositional Operators

$$<opBody> ::= \text{“}\{\text{”}\{<typeDecl>\}\ \{<callOrAssign>\}\ \{<operator>\}\text{“}\}\text{”} \quad (16)$$

$$<callOrAssign> ::= <call>\text{“;”} \quad (24)$$

$$| <symbolRef> = (<call> \ | <symbolRef> \ | <inputRef>)\text{“;”} \quad (25)$$

Compositional operators are used to connect other operators together via streams. They contain no logic or storage of their own. Variables declared in the {<*typeDecl*>} section are stream variables used to connect operators (see section 3.8 for semantics of stream variables). Statements and expressions in the {<*callOrAssign*>} section instantiate and connect operators, using operator calls, assignment to streams, and bitwise wiring operations such as bit selection, bit concatenation, and bit shifting. *Implementation note: Bitwise wiring operations on streams are presently not supported.* Stream history using the “@” operator, *i.e.* delay, is not supported in compositional operators.

3.11 Embedded C++ Operators

$$<opBody> ::= \text{“}\%\{\text{”}<C++\ code>\text{“}\%\}\text{”} \quad (17)$$

Embedded C++ operators encapsulate user-defined, processor-side code blocks which communicate via TDF streams. *Implementation note: Embedded C++ operators are presently not implemented.*

3.12 Copy Operators

Copy operators are built-in operators used to fan-out a stream to multiple consumers. A copy operator is equivalent to a behavioral operator with a single input stream and multiple output streams, all having the same token data type. The built-in `copy` operator is polymorphic with respect to the token data type and number of output streams. Its prototype is shown below, with ***`type`*** denoting the token data type and elipsis (...) denoting arbitrarily many additional output streams.

```
copy (input type i, output type o1, output type o2, ...);
```

3.13 Segment Operators

Segment operators do not contain user code, rather they are built-in operators used to provide a streaming interface for SCORE segments. Segment operators may be instantiated in a compositional operator to provide streaming access to arrays (for array parameters as well as locally declared arrays). Segment operators may also be instantiated by C++ code and subsequently connected to operators.

Segment operators exist in several flavors, depending on their access pattern (sequential or random access) and read-write restriction (read-only, write-only, read-write). Sequential segment operators read or write their content sequentially, terminating upon reaching the end of their address range. They need only a data stream. Random access segment operators allow access to any address. They use an address stream as well as an incoming and/or outgoing data stream.

TDF prototypes for all flavors of segment operators are shown below. Parameterization allows a segment to be word-addressable with arbitrarily-wide words. Parameter `dwidth` is the data stream width (in bits). Parameter `awidth` is the address stream width. Parameter `nelems` is the number of words in the segment (`nelems` $\leq 2^{\texttt{awidth}}$). Array parameter `contents` represents the initial and final value of the segment, passed to the operator by "value-return." Note that random access, read-write segments use separate outgoing and incoming data busses (`dataR` and `dataW`, respectively) as well as a read/write mode stream (`write`, high to write, low to read). *Implementation note: Data and address stream widths are presently limited to no more than 64 bits.*

- Sequential, read-only:

```
segment_seq_r (param  unsigned[7]              dwidth,
               param  unsigned[7]              awidth,
               param  unsigned[awidth+1]       nelems,
               param  unsigned[nelems][dwidth] contents,
               output unsigned[dwidth]         data);
```

- Sequential, write-only:

```
segment_seq_w (param  unsigned[7]              dwidth,
               param  unsigned[7]              awidth,
               param  unsigned[awidth+1]       nelems,
               param  unsigned[nelems][dwidth] contents,
               input  unsigned[dwidth]         data);
```

- Random access, read-only:

```
segment_r     (param  unsigned[7]              dwidth,
               param  unsigned[7]              awidth,
               param  unsigned[awidth+1]       nelems,
               param  unsigned[nelems][dwidth] contents,
               input  unsigned[awidth]         addr,
               output unsigned[dwidth]         data);
```

- Random access, write-only:

```
segment_w     (param  unsigned[7]              dwidth,
               param  unsigned[7]              awidth,
               param  unsigned[awidth+1]       nelems,
```

```
                param  unsigned[nelems][dwidth] contents,
                input  unsigned[awidth]        addr,
                input  unsigned[dwidth]        data);
```

- Random access, read-write:

```
segment_rw    (param  unsigned[7]              dwidth,
               param  unsigned[7]              awidth,
               param  unsigned[awidth+1]       nelems,
               param  unsigned[nelems][dwidth] contents,
               input  unsigned[awidth]         addr,
               output unsigned[dwidth]         dataR,
               input  unsigned[dwidth]         dataW,
               input  boolean                  write);
```

3.14 Statements

The following statements are supported in behavioral operators. For a discussion of statements supported in compositional operators, see section 3.10.

$$
\begin{aligned}
\textit{<stmt>} ::= & \textit{<stmtIf>} && (26)\\
| & \textit{<stmtGoto>} && (27)\\
| & \textit{<stmtAssign>} && (28)\\
| & \textit{<stmtBuiltin>} && (29)\\
| & \textit{<stmtCall>} && (30)\\
| & \textit{<stmtBlock>} && (31)
\end{aligned}
$$

If-then-else

$$\textit{<stmtIf>} ::= \text{“if”}\ \text{“(”}\textit{<expr>}\text{“)”}\ \textit{<stmt>}\ [\text{“else”}\ \ \textit{<stmt>}] \qquad (32)$$

The classic if-then-else. The predicate expression *<expr>* must be of type “boolean”. Short circuit evaluation is not supported, and all parts of the predicate are computed. If the predicate evaluates to “true”, the first *<stmt>* is performed, otherwise the optional “else” *<stmt>* is performed, if present. “else” statements associate with the nearest “else”-less “if”, unless explicit bracing indicates otherwise.

Examples:

```
if (a<0) a=0; else if (b<0) b=0; else c=0;
```

State transitions (goto, stay)

$$
\begin{aligned}
\textit{<stmtGoto>} ::= & \text{“goto”}\ \textit{<id>}\text{“;”} && (33)\\
| & \text{“stay”}\ \text{“;”} && (34)
\end{aligned}
$$

Implement an immediate transition to a specified next state of the finite state machine. “stay” is equivalent to a “goto” targetting the current state. If neither

form is present in a state action, a "stay" is implied at the end of the action. It is an error to transition to a state that consumes a previously closed input stream. Consequently, it is an error to "stay" in a state with an "eos" in its signature, since the arrival of the "eos" token closes the associated input stream.

Assignment

$$<stmtAssign> ::= <symbolRef> \text{“=”} <expr> \text{“;”} \tag{35}$$

If the left side is a register or temporary variable, assignment binds its value. If the left side is an output stream, assignment emits one output token.

Built-in Statements

$$<stmtBuiltin> ::= \text{“close”} \text{“(”} <id> \text{“)”} \text{“;”} \tag{36}$$

$$\mid \text{“printf”} \text{“(”} <fmt> \{\text{“,”} <expr>\} \text{“)”} \text{“;”} \tag{37}$$

Built-in statements look syntactically like inlined operator calls to operators with no return stream. `close` closes the specified output stream by emitting an "eos" token. `printf` prints a text message, where <*fmt*> is a double-quoted formatting string as per the C `printf` convention. *Implementation note:* ***`printf`*** *is presently implemented only for the C back-end, namely by a call to the standard C* ***`printf`*** *with all arguments cast to type* ***`long long`*** *and referenced in <fmt> by "%ll".*

See also the built-in exception `done` (Section 3.16), which is syntactically identical to a built-in statement.

Inlined Operator Calls

$$<stmtCall> ::= <call> \text{“;”} \tag{38}$$

This form of a call supports operators with no return stream, or ignores the return value of operators with return streams. See rule (71) for more information on calls.

Statement Blocks

$$<stmtBlock> ::= \text{“\{”} \{<typeDecl>\} \{<stmt>\} \text{“\}”} \tag{39}$$

Statement blocks create a local scope inside a state action. All variables declared inside a statement block are temporaries.

3.15 Expressions

$$
\begin{aligned}
\text{<}expr\text{>} ::= & \text{<}exprArray\text{>} && (40)\\
| & \text{<}exprCond\text{>} && (41)\\
| & \text{<}exprBop\text{>} && (42)\\
| & \text{<}exprUop\text{>} && (43)\\
| & \text{<}exprCast\text{>} && (44)\\
| & \text{<}exprBuiltin\text{>} && (45)\\
| & \text{<}call\text{>} && (46)\\
| & \text{<}symbolRef\text{>} && (47)\\
| & \text{<}inputRef\text{>} && (48)\\
| & \text{<}const\text{>} && (49)\\
| & \text{“(”<}expr\text{>“)”} && (50)
\end{aligned}
$$

Array Expressions

$$
\begin{aligned}
\text{<}exprArray\text{>} ::= & \text{“\{”<}exprList\text{>“\}”} && (51)\\
\text{<}exprList\text{>} ::= & \{\text{<}expr\text{>} \{\text{“,”<}expr\text{>}\}\} && (52)
\end{aligned}
$$

Array expressions are used to construct initial values for arrays and streams. Like scalar initial values, array initial values may contain arbitrary expressions, provided each array element evaluates to a compile-time constant or to an expression involving parameter inputs.

Conditionals

$$\text{<}exprCond\text{>} ::= \text{<}expr\text{>}_p \text{ “?” } \text{<}expr\text{>}_t \text{ “:” } \text{<}expr\text{>}_f \quad (53)$$

Conditional expression as in C. The predicate $\text{<}expr\text{>}_p$ must be of type “boolean”. If the predicate evaluates to “true”, the expression takes on the value of $\text{<}expr\text{>}_t$, otherwise it takes that of $\text{<}expr\text{>}_f$. Conditional expressions have the same evaluation semantics as “if” statements (rule (32)), including no short-circuit evaluation.

Examples:

```
max = (a>b)?a:b;
```

```
callOnlyOne = cond ?  F(x) :  G(x);
```

Binary Operators

$$\begin{aligned}
\text{<exprBop>} &::= \text{<expr><bop><expr>} && (54)\\
\text{<bop>} &::= \text{“\&\&” | “||”} && (55)\\
&\quad | \text{ “\&” | “|” | “\^{}”} && (56)\\
&\quad | \text{ “==” | “! =”} && (57)\\
&\quad | \text{ “<=” | “>=” | “<” | “>”} && (58)\\
&\quad | \text{ “<<” | “>>”} && (59)\\
&\quad | \text{ “+” | “−”} && (60)\\
&\quad | \text{ “*” | “/” | “\%”} && (61)\\
&\quad | \text{ “.”} && (62)
\end{aligned}$$

Binary operators perform logic and arithmetic on a pair of operands. Automatic type upgrades are performed to retain natural arithmetic semantics, so that no bits of precision are lost. If only one of the operands is signed, the other is made signed by adding a two's complement sign bit. The narrower operand is typically widened to match the wider operand. The result type is given the smallest width guaranteed to retain all significant bits of result.

Most of these operators work like their C counterparts. Bit-parallel logic operators, however, require unsigned operands (pure bit-vectors). The following table, listed in order of operator precedence (lowest at top), details the semantics of each binary operator. The dummy symbols a and b refer to application of operator $\square$ as: $a \square b$. merge(a, b) refers to the common, upgraded type for both operands.

Operator	Description	Operand Types	Result Type
“&&” “||”	logical and, or (no short-circuit evaluation)	“boolean”	“boolean”
“&” “|” “^”	bit-parallel and, or, xor	“unsigned[n]”	“unsigned[n]”
“==” “! =”	equals, not equals	any (same for a, b)	“boolean”
“<=” “>=” “<” “>”	comparison	numeric	“boolean”
“<<” “>>”	bit-shift	a=integer, b=integer	same as a
“+” “−”	addition, subtraction	numeric	merge(a, b) widened by 1 bit
“*”	multiplication	numeric	merge(a, b) widened to sum of operand widths.
“/”	division	numeric	same as a [2]
“%”	modulo	integer	same as b
“.”	form fixed-point from int and frac parts	integer	fixed-point concat

[2] The result type “same as a” for “/” is valid only with an integer divisor. *Implementation note: Fixed point division is presently not supported.*

Unary Operators

$$<exprUop> ::= <uop><expr> \quad (63)$$

$$<uop> ::= \text{“+”} \mid \text{“}-\text{”} \quad (64)$$

$$\mid \text{“!”} \mid \text{“}\sim\text{”} \quad (65)$$

Unary operators perform logic and arithmetic on a single operand. All unary operators use prefix notation, *i.e.* are placed before their operand. Automatic sign upgrades are performed for unary “+”, “−” to retain natural arithmetic semantics. The result type is the same as the possibly-upgraded operand type. The following table details the semantics of each unary operator. All unary operators presently have the same precedence level.

Operator	Description	Operand Type
“+” “−”	arithmetic copy, negation	signed integer/fixed
“!”	logical negation	“boolean”
“~”	bit-parallel inversion	“unsigned[n]”

Type-Casting

$$<exprCast> ::= \text{“(”} <type> \text{“)”} <expr> \quad (66)$$

$$\mid \text{“(”} <sign> \text{“)”} <expr> \quad (67)$$

Type-casting provides explicit type upgrading. Casting supports only those information-preserving transformations allowed by automatic type upgrades, namely widening and sign upgrades (“unsigned[m]”→“signed[n]”, $n > m$). Casting is not a mechanism for general format conversion. For that end, use “bitsof” (rule (70)) to retrieve a bit-vector representation, then apply other operators (*e.g.* “.” to create a fixed-point number from integers).

Implementation note: Type casting to a narrower type is presently allowed for convenience, denoting a bit truncation, and generating a compiler warning. This is a work-around for the syntactic omission of inline bit subscripting, which makes `bitsof(x)[y:z]` *impossible and requires subscripting an intermediate variable.*

Examples:

```
signedVersion = (signed)a;
wideVersion = (unsigned[32])a;
```

Built-in Expressions

$$<exprBuiltin> ::= \text{“cat”} \text{“(”} <exprList> \text{“)”} \quad (68)$$

$$\mid \text{“widthof”} \text{“(”} <expr> \text{“)”} \quad (69)$$

$$\mid \text{“bitsof”} \text{“(”} <expr> \text{“)”} \quad (70)$$

$$<exprList> ::= \{<expr> \{\text{“,”} <expr>\}\} \quad (52)$$

Built-in expressions look syntactically like inlined operator calls to operators with return streams. `cat` concatenates bit-vector arguments (type “unsigned[n]”) into a longer bit-vector result. `widthof` returns the number of bits in its argument's bit-representation. `bitsof` returns its argument's bit-representation as type “unsigned[n]”.

Inlined Operator Calls

$$<call> ::= <id> \text{“(”} <exprList> \text{“)”} \quad (71)$$

$$<exprList> ::= \{<expr> \{\text{“,”} <expr>\}\} \quad (52)$$

Inlined operator calls *in a behavioral operator* specify stream communication with a private instance of the given operator. Evaluation of a call will emit a token to each of the called operator's input streams and will then wait for a token from each of the called operator's output streams. Syntactically, an inlined operator call looks like a C function call, with pass-by-value semantics for outgoing tokens and assignment semantics for incoming tokens. In the argument list, arguments in positions of input streams and param inputs cause emission of tokens. Any valued expression may be used for such arguments. Arguments in positions of output streams are assigned the value of returned tokens. Any variable or an operator's own output stream may be used for such arguments. A call to an operator with a return stream has an expression value corresponding to the token on the return stream. That value may be ignored by using the call in a statment context. *Implementation note: Use of inlined calls requires the* `tdfc -xc` *flag;* `-xc` *presently generates highly inefficient code.*

Examples:

```
max = getMax(x,y);
```

```
getMax2(x,y,max);
```

Symbol Reference

$$<symbolRef> ::= <id> [<arrayLoc>] [<bitSelect>] \quad (72)$$

$$<inputRef> ::= <id> [<bitSelect>] [\text{“@”} <expr>] \quad (73)$$

$$<arrayLoc> ::= \text{“[”} <expr> \text{“]”} \quad (74)$$

$$<bitSelect> ::= \text{“[”} <expr> \text{“]”} \quad (75)$$

$$\mid \text{“[”} <expr> \text{“:”} <expr> \text{“]”} \quad (76)$$

Symbol references name streams, “param” inputs, and variables. A reference to an input stream (rule (73)) names the value of its most recently consumed data token,[3] or that of a previous token using an optional unsigned index [“@” <*expr*>] (“@0” is the most recent). A reference to an output stream is used to emit a token to that stream. A reference to a “param” input refers to its value. A reference to a variable may refer to its present value for reading or to its storage for assignment, depending on context.

Array-type symbols (variables or <*param*> inputs) may be referenced in two contexts. First, an array may be used as a <*param*> argument in an operator call. If the array represents a memory segment, then such use indicates passing ownership of the segment to the called operator. Second, an array may be indexed using the <*arrayLoc*> specifier (rule (74)) to select a particular word of the array. An indexed array reference takes its type from the array-type's element type. *Implementation note: Use of indexed array references requires the* `tdfc -xc` *flag,*

[3] Consuming an “eos” token does not advance an input stream's history, so a subsequent reference `stream@0` will refer to the last data token consumed from that stream.

except for read-only arrays in the Verilog back-end; `-xc` *presently generates highly inefficient code.*

A reference to an "unsigned[n]" bit-vector may optionally select a particular bit as "x[pos]" (rule (75)) or a bit sub-range using "x[$high$:low]" (rule (76)). Bit positions are counted from 0 for the least significant position.

Examples:

```
location5       = array[5];
lowBit          = word[0];
location5LowBit = array[4][0];
lowNibble       = word[3:0];
```

3.16 Exceptions

Exceptions in TDF are themselves operators. Specifically, they are embedded C++ operators, typically scoped inside the faulting operator. A faulting operator may pass data out to the exception as well as receive data back.[4] Suspending and resuming a faulting operator is supported by natural data-flow semantics, by having the operator wait for the exception's return token(s). Zero-width tokens of type "unsigned[0]" may be passed when only synchronization is required. Non-resuming operators need not return anything. *Implementation note: Exceptions as embedded C++ operators are presently not implemented.*

The built-in exception `done` is available to each operator for self-termination. It is available to behavioral operators with the prorotype "unsigned[0] done(unsigned[0])" and to embedded C++ operators through a library call.

Examples:

```
exampleOp (input unsigned in[8], output boolean out)
{
  boolean x,y;

  state normal(i):
    if      (in==255) { close(out); done(); }
    else if (in==254) { exception1();      }
    else if (in==253) { x=exception2(y);  }
    else ...

  ...
```

[4] Implementation note: communication with exception operators typically crosses the processor-array boundary. One possibility for implementation is with conventional streams. This requires allocating FPGA interconnect for exception/operator streams, which may be unjustifiably expensive for rarely-occuring exceptions. An alternative way is to probe the faulting page on the FPGA, extracting the exception's arguments from registers. This method requires hardware support but avoids the need to allocate stream interconnect for each potentially-faulting operator. Hardware support on the FPGA might involve an "attention" line from each page, an exception controller to probe pages, and a common tri-state bus to receive probe results. Instead of probing pages through a tri-state bus, we might also use a hardware read-back mechanism, which would be slower but require less interconnect resources.

```
  exception1 () { ... }

  boolean exception2 (input boolean flag) { ... }
}
```

4 C/C++ Interface

SCORE primitives may be accessed from C or C++ code running on the processor through the SCORE library API. The API represents typed versions of the SCORE primitives, including streams, operators, and segments. SCORE primitives are typed both statically and dynamically to match their TDF data types. The data format of a primitive (*e.g.* signed or unsigned) is encoded directly in its C/C++ type. The data width of a primitive is specified dynamically when creating the primitive.

The following TDF data formats are supported. These formats are denoted by the label *<format>* in the remainder of this section.

`UNSIGNED`	– corresponds to TDF `unsigned[`*w*`]`
`SIGNED`	– corresponds to TDF `signed[`*w*`]`
`FIXED`	– corresponds to TDF `unsigned[`*i*`.`*f*`]`
`SIGNED_FIXED`	– corresponds to TDF `signed[`*i*`.`*f*`]`
`BOOLEAN`	– corresponds to TDF `boolean`

These data formats appear as part of the C/C++ type of a SCORE primitive (*e.g.* `UNSIGNED_SCORE_STREAM`). The width associated with a data type, denoted as *w* or *i.f* above, is not actually part of the C/C++ type, but must be specified dynamically when creating the primitive (*e.g.* `NEW_UNSIGNED_SCORE_STREAM(`*w*`);`). *Implementation note: Token data widths are presently limited to 64 bits.*

4.1 General Setup

Any program using the SCORE API must perform certain actions for setup.

Include headers

- A program must include the general SCORE header: `#include "Score.h"`.
- A program must include a header for each TDF operator used: `#include "`*operatorname*`.h"` (such headers are produced by the `tdfc` compiler).

Program entry and exit

- On startup, a program must call `score_init()`.
- On exit, a program must call `score_exit()`.

4.2 Streams

The SCORE library API represents streams using the *<format>*`_SCORE_STREAM` collection of types. Upon creation, a stream is designated as input (readable by user), output

(writeable by user), or generic (used to connect two operators). Streams are connected to operators by passing them as arguments to operator constructors (see section 4.3). API routines exist for reading, writing, and terminating streams, as well as querying stream state. *Implementation note: It is imperative to terminate streams after their contents are depleted, or else execution of the operator graph containing those streams will never terminate.*

Creating a stream

- `<format>_SCORE_STREAM NEW_<format>_SCORE_STREAM (int width);`
- `<format>_SCORE_STREAM NEW_READ_<format>_SCORE_STREAM (int width);`
- `<format>_SCORE_STREAM NEW_WRITE_<format>_SCORE_STREAM (int width);`
  These functions create and return a stream with data format `<format>` and bit width `width`. For fixed-point data formats, the width argument is replaced by a pair of width arguments for the integer and fractional parts. The first form creates a stream for connecting two operators; its tokens cannot be accessed by the user. The second form creates an input stream which may be read by the user. The third form creates an output stream which may be written-to by the user. In addition, each form has a variant that accepts a *depth hint* as a second argument to specify a desired minimum buffer size. That variant is named by extending the function name with `_DEPTH_HINT`.

Querying stream status

- `int STREAM_EOS(<format>_SCORE_STREAM);`
  This function returns non-zero if and only if the given stream (which must be an input stream) has received an end-of-stream token. If no token is available on the stream, the caller is suspended until one becomes available. Receipt of an end-of-stream token indicates the closing of a stream, after which any attempt to read a data token via `STREAM_READ()` is a run-time error.

Reading and Writing tokens

- `STREAM_READ(<format>_SCORE_STREAM, unsigned long long token);`
  This function attempts to read a token from the given stream (which must be an input stream). If a token is available, its value is returned in `token`. If the stream is empty, the caller is suspended until a token becomes available. It is a run-time error to call `STREAM_READ()` on a stream which has received end-of-stream, *i.e.* a stream for which `STREAM_EOS()` returns non-zero. It is a run-time error to call `STREAM_READ()` on a freed stream, *i.e.* a stream for which `STREAM_FREE()` has been called.
- `STREAM_WRITE(<format>_SCORE_STREAM, unsigned long long token);`
  This function attempts to write a token to the given stream (which must be an output stream). If space is available in the stream buffer, a token with value `token` is written. If the stream buffer is full, the caller is suspended until space becomes available. It is a run-time error to call `STREAM_WRITE()` on a closed stream, *i.e.* a stream for which `STREAM_FREE()` has been called.

Terminating a stream

- `STREAM_CLOSE(<format>_SCORE_STREAM);`
  This function closes an output stream, writing an end-of-stream token to it. It is a run-time error to call `STREAM_WRITE()` on a stream which has been closed with `STREAM_CLOSE()`.
- `STREAM_FREE(<format>_SCORE_STREAM);`
  This function detaches an input stream. It is a run-time error to call `STREAM_READ()` or `STREAM_EOS()` on a stream which has been freed with `STREAM_FREE()`.

4.3 Operators

The SCORE library API allows instantiating and connecting TDF operators using a syntax similar composition in TDF. Each operator compiled by `tdfc` is given a C/C++ instantiation function `NEW_<operatorName>()` whose prototype matches that of the TDF operator: input and output streams become `<format>_SCORE_STREAM` objects; scalar `param` arguments become `unsigned long long` 64-bit integers; and array `param` arguments become `<format>_SCORE_SEGMENT` objects. An operator's instantiation function is prototyped in its *master class* header "`<operatorName>.h`", which must be `#include`-ed by the user.

An operator may be instantiated and connected by calling the C/C++ instantiation function with proper `param` and stream arguments. An actual argument for an input stream formal parameter must be an output or generic C/C++ stream. An actual argument for an output stream formal parameter must be an input or generic C/C++ stream. See section 4.2 for more about stream directions.

Note that the user need not retain any pointer or reference to an operator instance. The operator will be accessed only through its streams and will be automatically destroyed by the operating system when those streams close.

4.4 Segments

The SCORE library API represents segments using the `<format>_SCORE_SEGMENT` family of types. Segments may be passed to operator instantiation functions wherever a `param` argument is expected.

Creating a segment

- `<format>_SCORE_SEGMENT NEW_<format>_SCORE_SEGMENT(int numelems, int width);`
  This function creates and returns a segment containing an array of `numelems` elements, each of data format `<format>` and bit width `width`. For fixed-point data formats, the width argument is replaced by a pair of width arguments for the integer and fractional parts. A segment operator encapsulating this segment must use streams which match the segment's data type. *Implementation note: Each array element is presently padded and limited to a width of 64-bits.*

- `SCORE_SEGMENT NEW_SCORE_SEGMENT(int numelems, int width);`
  This function creates and returns an "untyped" segment sized as an array of `numelems` elements, each `width` bits wide. This untyped form is useful for creating a segment representing a raw, randomly-accessed memory block. *Implementation note: Each array element is presently padded and limited to a width of 64-bits.*

Accessing segment contents

- `void* GET_SEGMENT_DATA(<format>_SCORE_SEGMENT);`
  This function retrieves a pointer to the given segment's data contents. These contents may be accessed using traditional C/C++ pointer/array access. If the segment is owned by another operator when its contents are dereferenced, then the dereferencing code is suspended until the segment becomes available. *Implementation note: Each array element is presently padded and limited to a width of 64-bits. A pointer to a typed segment's contents must be cast to `unsigned long long*` for proper pointer arithmetic.*

4.5 Segment Operators

The SCORE library API represents segment operators using the `NEW_ScoreSegmentOperator<mode>()` family of operator instantiation functions. Segment operators may be treated the same as any user-defined TDF operator (see section 4.3). Their instantiation prototypes are analogous to those of the TDF segment operators described in section 3.13

- Sequential, read-only:
```
NEW_ScoreSegmentOperatorSeqReadOnly(unsigned dwidth,
                                    unsigned awidth,
                                    size_t   nelems,
                                    <format>_SCORE_SEGMENT contents,
                                    <format>_SCORE_STREAM  data);
```
- Sequential, write-only:
```
NEW_ScoreSegmentOperatorSeqWriteOnly(unsigned dwidth,
                                     unsigned awidth,
                                     size_t   nelems,
                                     <format>_SCORE_SEGMENT contents,
                                     <format>_SCORE_STREAM  data);
```
- Random access, read-only:
```
NEW_ScoreSegmentOperatorReadOnly(unsigned dwidth,
                                 unsigned awidth,
                                 size_t   nelems,
                                 <format>_SCORE_SEGMENT contents,
                                 UNSIGNED_SCORE_STREAM  addr,
                                 <format>_SCORE_STREAM  data);
```

- Random access, write-only:
```
NEW_ScoreSegmentOperatorWriteOnly(unsigned dwidth,
                                  unsigned awidth,
                                  size_t   nelems,
                                  <format>_SCORE_SEGMENT contents,
                                  UNSIGNED_SCORE_STREAM  addr,
                                  <format>_SCORE_STREAM  data);
```
- Random access, read-write:
```
NEW_ScoreSegmentOperatorReadWrite(unsigned dwidth,
                                  unsigned awidth,
                                  size_t   nelems,
                                  <format>_SCORE_SEGMENT contents,
                                  UNSIGNED_SCORE_STREAM  addr,
                                  <format>_SCORE_STREAM  data_r,
                                  <format>_SCORE_STREAM  data_w,
                                  BOOLEAN_SCORE_STREAM   write);
```

References

[BUCK93] J. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model,* PhD thesis, University of California, Berkeley, 1993. ERL Technical Report 93/69.

[MURA89] T. Murata, "Petri nets: properties, analysis and applications," *Proc. IEEE* vol. 77 no. 4, April 1989, pp. 541-580.

[NAUR63] P. Naur ed., "Revised report on the algorithmic language Algol 60," *Comm. ACM* vol. 6 no. 1, 1963, pp. 1-17.