CAL Coding Practices Guide

Hardware programming in the CAL actor language

David B. Parlour

language version 1.0 — document edition 1

ASTG
Xilinx Inc
June 6, 2003

Contents

Chapter 1

Introduction

This guide describes a suggested set of coding practices for the CAL language when used in conjection with the SYSTEMBUILDER FPGA programming environment. In addition to documenting the type system and language coverage for synthesis, there are suggested practices for:

- creating actor models
- debugging actor models
- tuning actor models for area and performance

This guide assumes general familiarity with

- conventional programming languages such as C and Java
- the syntax of the CAL actor language (see the CAL Language Reference Manual).
- FPGA hardware design using HDLs

The intended audience is programmers using CAL with the SYSTEMBUILDER simulation and synthesis environment to develop highly concurrent implementations on platform FPGAs.

1.1 Getting started

A new user of CAL and SYSTEMBUILDER should start by inspecting and simulating a sample actor network with the CAL Language Reference Manual at hand. Sample applications include an MPEG-4 decoder in synthesizable form, and an OFDM modulator/demodulator functional model. Learning a new programming model is as much about understanding a new mindset as mastering a syntax, and the sample models are useful for gaining insight into the art of concurrent programming.

The steps for creating a new actor model are usually:

- Creating a testbench with stimuli and golden (expected) output data
- Describing the behavior of the application. In this stage, it is convenient to use the full power of the CAL language without regard to the constraints imposed by the back-end synthesis technology in SYSTEMBUILDER. One of the strengths of the actor programming model is that it naturally encourages the programmer to reason about the "three Cs"
  - concurrency
  - communication
  - control

  that are the biggest determinants of quality-of-result in a platform FPGA implementation.
- Transitioning the application portion of the model to synthesizable types and language constructs.
- Refactoring and refinement of the model to achieve desired performance and size goals.

1.2 Working with this guide

The guide is intended to illustrate good CAL coding practice with heavy reliance on the use of examples, rather than formal definitions. In the end, there is no substitute for just trying things out to discover what works and what doesn't.

The following chapters discuss the CAL type system (or lack thereof), introduce some ideas on debugging and creating test benches, and finally detail language coverage for synthesis as well as performance tuning ideas.

Chapter 2

Types

The CAL language specification includes syntax for declaring types, but does not define a type system. Specific type definitions may be imposed by particular back-end tools for processing CAL actors.

This chapter discusses the types required by the hardware generation back-end, as well as the use of Java native types for creating non-synthesizable actor models and testbenches.

2.1 Types for hardware code generation

2.1.1 bool

The **bool** type is a boolean quantity that can have the value 'true' or 'false'. When CAL is being compiled to hardware, the type **bool** is required or assumed in the following contexts:

- Comparison operators (=, !=, <, <=, >, >=). Result type is **bool**.
- Intrinsic boolean operators (not, and, or). Operands must be of type **bool**, result type is **bool**.
- Guard expressions. Must be of type **bool**.
- Conditional expression in a flow control statement (if, while) must be of type **bool**.

2.1.2 int

The **int** type is the only numerical type currently supported in hardware code generation. This is a two's complement signed binary value whose size may be specified as a type parameter named 'size'. **int**s with an unspecified size default to 32 bits. If the hardware compiler can determine by static analysis that a smaller number of bits will suffice without the loss of any information,

then the generated hardware representation will only have that smaller number of bits. Here is an example which declares several sized **int**s. Note that the type parameter value need not be a literal value, although the expression must evaluate to a known value at compile time. In this example, the I/O port types have parameters that depend on an actor parameter, which is a constant value known at compile time.

```
// Actor with an unsized int as parameter
actor intTest( int IOSZ )
  // I/O ports are typed with sized ints
  int(size = IOSZ) IN ==> int(size = IOSZ+4) OUT

  // actor state variable will default to 32 bits
  int sum := 0;

  // Actor state variable with specified width
  int(size=8) theta := 0;

  ...

end
```

Assignment

When an expression of type **int** is assigned to a variable of type **int**, the expression is modified by either sign extension or truncation to have the same size as the destination type.

The following sections give the rules for typing the result of various operations. In the hardware, the expressions are first evaluated using the rules for the component operations and then converted to the type of the destination. When the conversion implies a truncation of the expression, the final hardware may not generate the redundant bits, for the sake of efficiency.

Arithmetic operations

The following arithmetic operations on **int**s are supported for hardware code generation.

- unary **-**. The resulting size is the same as the operand size.
- **+**, binary **-**. The result of adding or subtracting two **int**s is an **int** whose size is one greater than the larger of the two operand sizes.
- *****. The result of multiplying two **int**s is an **int** whose size is the sum of the sizes of the two operands.
- **/**. Hardware division for divisors and dividends of type **int** is supported, and the result type is the same as the dividend. Note that the hardware that implements division takes multiple system clock cycles to execute.

Bitwise operations

The CAL language does not specify any shift or bitwise boolean operations corresponding to the familiar C operators (>>, <<, &, |, !, ^), so a library of functions is provided. The bit operator library for **int**s is supported both in simulation and code generation. This library is made visible in a CAL source file by adding the following statement before the actor definition:

```
import all caltrop.lib.BitOps;
```

The available bit operations are listed below. Note that all arguments must be of type **int**.

- **bitand**(*arg1*, *arg2*). Bitwise AND. The result size is the larger of the two operand sizes, and the smaller operand is sign extended before applying the operator.
- **bitor**(*arg1*, *arg2*). Bitwise OR. The result size is the larger of the two operand sizes, and the smaller operand is sign extended before applying the operator.
- **bitxor**(*arg1*, *arg2*). Bitwise exclusive OR. The result size is the larger of the two operand sizes, and the smaller operand is sign extended before applying the operator.
- **bitxnor**(*arg1*, *arg2*). Bitwise exclusive NOR. The result size is the larger of the two operand sizes, and the smaller operand is sign extended before applying the operator.
- **bitnot**(*arg1*). Bitwise NOT. The result size is the same as the operand size.
- **lshift**(*arg1*, *arg2*). Left shift with zero fill. The first operand is sign extended to at least 33 bits if its size is less than 33 bits, then left shifted with truncation to that size by the number of bit positions specified in the second operand, which must be non-negative.
- **rshift**(*arg1*, *arg2*). The first argument is right shifted with sign extension by the number of bit positions specified in the second operand, which must be non-negative. The return type is the same as that of the first operand.

2.1.3 list

The only aggregate type supported for hardware code generation is **list**, which is an array of elements whose type must be **list**, **int** or **bool**. **list**s are declared with two parameters:

- **type**. This is syntactically a *type* parameter, and it declares the type of the individual elements in the list. When **type** is itself a list, a multi-dimensional list is being declared. Multi-dimensional lists are implemented

as one-D physical memories in the hardware, and additional hardware is generated automatically to compute the one-D index from the indices in the source code for each dimension. This type must be completely resolved at compile time. Note that the list indices are zero based, i.e. they run from 0 up to one less than the declared list size.

list indices are zero based

- **size**. This is syntactically a *value* parameter which determines the number of elements in the list. The value must be a constant that can be evaluated at compile time.

The use of lists in the source is restricted as follows:

- **Actor state variables**. The only variables that can be declared of type **list** are actor state variables. The physical interpretation of a **list** in hardware is a statically allocated memory, so this is the only context where it makes sense to declare a **list**.
- **Initializer expressions**. All declared **list**s must be initialized (see below for an example), and the initializer expression is syntactically a CAL*list comprehension*. The **list** may be declared constant (implying a ROM in the hardware implementation) by using **=** to assign the initializer, instead of **:=**.
- **list access**. The only access to a **list** type variable permitted is to one underlying scalar element, through an indexer expression with fully specified indices (for example myOneD[i], or myTwoD[i][j], etc).
- **Assignment**. Assignments may only be made to one underlying scalar element (for example myOneD[i] := 10;, or myTwoD[i][j] := -1, etc).

The following example shows the declaration and initialization of several lists:

```
actor listDecl( int W, int H, int SZ ) int IN ==> int OUT

  // declare and initialize a one-d list
  list( type:int(size=SZ), size=W * H )
          oneD := [ 0 : for i in Integers(1,SZ) ];

  // a utility function for initializing a list
  function initList( v, size ) :
    [ v : for i in Integers(1,size) ]
  end

  // declare, initialize a two-d list using function
  list( type:list(type:int(size=SZ), size=W), size=H )
          twoD := initList( initList( 0, W), H);

  // Compute nth Fibonacci number using recursion
```

```
  // Note: recursion is not supported in hardware
  function fibonacci( n ) :
    if n = 0 then 0 else
      if n=1 then 1 else fibonacci(n-1)+fibonacci(n-2) end
    end
  end

  // create a ROM to return the i'th Fibonacci number
  // Note: initializer evaluates to a list of constants
  // so this can be implemented even if the function
  // used is not suitable for hardware implementation.
  list( type:int, size=10 ) fibROM =
            [ fibonacci(i) : for i in Integers(0,9) ];

  ...

end
```

2.2 Unsynthesizable types

It may be convenient to use unsynthesizable types in a number of situations:

- **Initial system modeling**. It is common to capture the behavior of a system as rapidly as possible, using the full scope of the CAL actor language, and then refine a working model to a realizable version in steps. The initial model could use floating point arithmetic, for example, whereas the synthesizable would have to be converted to **int**s. Also, the specific structure of the actor implementation can have a big impact on area and throughput, so Actor source code is likely to go through a number of refinement and refactoring steps during implementation.

- **test bench and debug actors**. There can be any number of actors needed to construct a complete simulation model that will never be realized in hardware. These actors can use the full capabilities of the underlying simulation host instead of the synthesizable subset of the language.

- **Initializer expressions**. It may be desirable to use nonsynthesizable constructs for intermediate expressions, that ultimately result in synthesizable code. A useful example of this might be a ROM sine function lookup table whose inital values are computed using the sine() function which is real-valued.

When an expression or variable is untyped, or has a declared type that is not in the synthesizable set of the previous section, the type is ignored by the simulator and the hardware compiler. Instead, the type is inferred from the actual data according to the rules of the Java language. The following example shows the initialization of a lookup tabel with sine function values. In this

case the result of constant expresssion evaluation is synthesizable, although the intermediate expressions are not:

```
// Make the Java math library visible
import java.lang.Math;
actor sineGen( int A, int N, int BITS )
            int THETA ==> int(size=BITS) V

  // Initialize the lookup table. Both the angular
  // arguments and the result are promoted to floating
  // point (Java double).
  // The result is then converted to int.
  list( type:int(size=BITS), size=npoints) rom =
   [A * sin( 6.28318 * i / N ) : for i in Integers(0,N-1)];

 ...

end
```

In the following example, a Java library function is used to generate random floating point numbers:

```
// Random number generator is in the Java utility package
import all java.util;

// Actor to generate two streams of random tokens, one
// with a Gaussian distribution with mean and standard
// deviation parameterized, the other uniform
// integers in the range [ 0, UNI_RANGE-1 ]
actor myRandom( GAUSS_SD, GAUSS_MEAN, UNI_RANGE )
           ==> GAUSSIAN, UNIFORM

  // Create a random number generator object
  r = Random( );

  // Produce the random token streams
  action ==>
     GAUSSIAN:[r.nextGaussian()*GAUSS_SD + GAUSS_MEAN],
      UNIFORM:[r.nextInt( UNI_RANGE )]
  end
end
```

Some care is required when using types inherited from the underlying Java. For example, the following two code fragments result in different values being assigned to the variable ‘a’:

```
  a := 0;
```

```
a := a + 1;
a := a / 2;   // result is 0

a := 0.;
a := a + 1;
a := a / 2;   // result is 0.5
```

Chapter 3

Debug

While the CAL simulator framework provides a number of debug features accessible through a graphical interface (described elsewhere), some programmers prefer to diagnose problems with the 'printf' style debugging commonly practiced when programming in more conventional languages. This chapter documents several features and utility functions that enable this more textual style of debugging that usually involves adding debug code to the source.

3.1 Utility functions and procedures

3.1.1 println

The **println** procedure echoes its single argument to the console, usually the command shell from which SYSTEMBUILDER was invoked. Its argument is converted to a string according to Java language conventions.

For example, this code fragment:

```
a := -2;
count := 5;
println("The value of a is " + a + " at count " + count);
```

would display the following message to the console:

```
The value of a is -2 at count 5
```

List types cannot be used in an expression as shown above, and must appear as a simple argument. For example, use the following form to print the value of a list along with other text:

```
a := Integers( 0, 3 );
count := 5;
println("The value of a at count " + count + " is");
println( a );
```

The resulting console message is:

```
The value of a at count 5 is
[ 0, 1, 2, 3 ]
```

3.1.2 SOP

The **SOP** function takes a single argument, which it displays on the console and uses as its return value. This is useful for inspecting some value in a context where a procedure call cannot be used, for example in a guard expression. Since the value is echoed to the console whenever the expression is evaluated, the resultant display can be confusing. Remember that the specific schedule of execution of a concurrent program can be implementation-dependent so the ordering of printouts to the console may be unexpected!

Here is an example of the SOP function used to monitor a guard condition:

```
myThing: action IN:[x] ==> OUT:[ 2 * x ]
  guard SOP(x) > 0
  do
    println( "myThing just fired!" );
  end
```

For a partial input sequence (... -2, -10, 7, -3, ...) on the input port IN this code fragment might produce an output such as:

```
-2
-10
7
myThing just fired!
-3
... etc ..
```

This illustrates the use of the SOP function to gain insight into the behavior of an actor - in this case why an action is not firing.

Keep in mind that the output to console is unpredictable since it depends on when the simulator chooses to evaluate the guard condition for the action. This complication not withstanding, there are situations where SOP can help to locate subtle problems with program execution.

3.2 Magic Variables

There are a number of magic variables that can be used to ineract with the simulator from within the CAL source code.

3.2.1 _CAL_traceOutput

This variable can be used to turn on and off reporting of action firing information to the console. This information details each individual action firing and the reason for its selection by the simulator's action scheduler.

```
// Example of selective tracing of action firing
actor dummy() IN ==> OUT

  // magic variable declared as actor state variable
  _CAL_traceOutput := false;

  int count := 0;

  myThing: action IN:[ x ] ==> OUT:[ 2*x ]
  do
    count := count + 1;

    // Enable firing report while count < 10
    _CAL_traceOutput := count < 10;
  end

end
```

3.2.2 _CAL_tokenMonitor

This variable can be used to turn on the creation of test bench data by recording all actor token traffic. As with _CAL_traceOutput, it must be declared as an assignable actor state variable. Token traffic will be written into a subdirectory called 'monitor' which must exist in the directory from which SYSTEMBUILDER was invoked. This subdirectory will not be created, so the two conditions for recording of test bench data are that the subdirectory exists and the _CAL_tokenMonitor is assigned a value of 'true'.

The process of turning recorded token traffic into an HDL simulation testbench is described elsewhere.

3.3 Test Benches

This section describes several techniques for creating test bench actors.

3.3.1 File Reading

The following example actor shows how a stream of tokens can be read from a file and used to stimulate the rest of an actor network. It uses two utility functions built into the simulation environment:

- **openFile**. Opens the file whose pathname is specified as a string argument. Relative path names are with respect to the directory in which SYSTEMBUILDER was invoked. It returns a file descriptor. It the specified file cannot be located the simulation will fail to start, probably by throwing an exception message identifying the function call.

- **readByte**. Reads the next byte from the file pointed to by the file descriptor argument. If the read succeeds, the return value will be in the range 0 to 255. Negative return values indicate a problem, including end-of-file.

File reader actions are typically used to inject test inputs into the model, and to provide a stream of 'golden' results tokens for comparison against the data actually produced in the actor network under test.

```
// File reader actor

// The parameter "filename" is a string with a path
// to the data file to be read. It should be absolute,
// or relative to the directory from which
// SYSTEMBUILDER was invoked.
actor fread( fileName ) ==> int(size=9) OUT

  // Initialize the file descriptor, get first byte
  fd = openFile(fname);
  int nextc := readByte( fd );

  // Keep providing output until EOF detected
  action ==> OUT:[ c ]
  guard
    nextc >= 0
  var
    int c
  do
    c := nextc;
    // prefetch the next byte
    nextc := readByte( fd );
  end

  // Actor hangs at EOF

end
```

3.3.2 Video Viewer

When debugging actor networks that process video data, it is useful to be able to view the resulting video sequence. This can be accomplished with the Picture and Frame objects provided by the simulator environment.

- **Frame**. Creates a pop-up widow to contain the display.
- **Picture**. A pixel display object.

The use of these objects and some relevant methods is illustrated in the following RGB viewer example:

```
// Display a series of R,G,B pixels that arrive in scan
// order with image size fixed at compile time.
actor RGBDisplay ( width, height, title ) PIXELS ==>

  picture = Picture( width, height);
  frame = JFrame( title );

  // Use an initialization action to execute procedural
  // code before starting simulation.
  initialize ==>
  do
    frame.getContentPane().add(picture);
    frame.pack();
    frame.setVisible(true);
    readFrame();
  end

  x := 0;
  y := 0;

  action PIXELS:[ r, g, b ] ==>
  do
    picture.setPixel( y, x, r, g, b );
    // Update display for each pixel drawn
    picture.displayImage();
    x := x + 1;
    if x >= width then
      x := 0;
      y := y + 1;
      if y >= height then
        y := 0;
        // Put displayImage() here to update
        // screen only once per frame.
      end
    end
  end

end
```

3.3.3 Plotting

The simulation environment contains a predefined plotting actor that can be used to display plots of token values. The actor name is **PtPlot**. Its ports are

- **Data**. The data input can receive a single series of tokens, or multiple interleaves streams. The token values can be either simple numerical values or pairs identifying a series name and data value. In XY plot mode, pairs of numerical tokens are provided to PtPlot. For example:
  - Unlabeled series: `1` 1 2 3 5
  - Single labeled series: `[`"fib",1] ["fib",1] ["fib",2] ["fib",3] ["fib",5]
  - Two labelled series: `[`"even",2] ["odd",1] ["even",4] ["odd",3] ["even",6] ["odd",5]
  - XY plot series: `[`-1,1] [1,-1] [1,1] [-1,-1]

  When there is more than one series present, they will all be plotted on the same grid with different colors or line styles. It is not necessary to present the data points for the different series in any particular relative order, but how the points display relative to each other will depend on the 'time' parameter (see below).
- **Redraw**. The arrival of a token on this input triggers a redraw of the plot. If unconnected, this behavior is determined by a parameter.

The relevant parameters of the PtPlot actor are:

- **title**. A string.
- **impulse**. 'true' or 'false'. Display each data point as an impulse.
- **connected**. 'true' or 'false'. Connect the points in the series.
- **marks**. 'points'
- **autoredraw**. An integer specify the number of input tokens between redraws of the plot.
- **mode**. 'xy' or omitted. Enable XY plotting instead of series.
- **time**. 'true' or 'false'. Plot points according to arrival time in simulation, or just in order of arrival.

Chapter 4

Language Coverage

Language converage for synthesis is detailed in the following table. The element names correspond to the syntactic elements define in Appendix A of the CAL Language Reference Manual (CLR). The CLR column references the manual section that discusses each element in detail.

As several examples in this guide show, it is often useful to use unsynthesizable constructs in initializer expressions. As long as these constructs can be evaluated to constants at compile time, the actor can be realized in hardware.

Element	Support	Comment	CLR
Actor	yes	type parameters and time clause not supported.	3.
TypePar	no		3.
ActorPar	yes	Actor parameters must be bound to known constant values at compile time	3.
IOSig	yes		3.
PortDecl	yes	keyword **multi** not supported. Port type must be a synthesizeable scalar type	3.
TimeClause	no		3.2
Import	yes	Required for proper evaluation of compile time constants involving external definitions	3.1
Type	yes	Only elements with one of the types discussed in Chapter 2 can be used in synthesizable constructs. Other types may be used if their related constructs evaluate to initializers which are compile time constants	4.
TypeAttr	yes	See Chapter 2 for the required type attributes	4.
VarDecl	yes	Keyword **mutable** not supported.	5.1

continued on next page

(continued)

Element	Support	Comment	CLR
Expression	yes	Non-synthesizable expressions that evaluate to constants known at compile time may be used as initializers	6.
PrimaryExpression	yes	Object access (*objName.memberName*) not supported	6.
SingleExpression	yes	No support for closures. Limited support for comprehensions as discussed below.	6.
ExpressionLiteral	yes	Only integer and boolean literals are supported	6.1
IfExpression	yes		6.7
LetExpression	yes		6.8
LambdaExpression	no	No support for closures.	6.9.2
FormalPar	yes		6.9.3
ProcExpression	no	No support for closures.	6.9.2
FuncDecl	yes	Functions are inlined, implying replication of hardware at each point of application.	6.9.3
ProcDecl	yes	Procedures are inlined, implying replication of hardware at each point of invocation.	6.9.3
SetComprehension	no		6.10
ListComprehension	yes	List comprehensions may only be used as initializers for actor state variables of type **list**, and must evaluate to a list of constant values known at compile time.	6.10
MapComprehension	no		6.10
Mapping	no		6.10
Generator	yes	Only suppported in context of a constant **list** initializer.	6.10.2
TypeAssertionExpr	no		6.11
Statement	yes	choose and foreach not suppported	7.
AssignmentStmt	yes	Assignment to an object (*objName.memberName* := ...) not supported. Assignments to a **list** element must access a scalar element (not a sub-**list**, for example).	7.1
Index	yes	All indexers must refer to a scalar element.	7.1.3
FieldRef	no		7.1.3
CallStmt	yes	The procedure body is in-lined at this point.	7.2
BlockStmt	yes		7.3
IfStmt	yes		7.4
WhileStmt	yes		7.5
ForEachStmt	no		7.6
ForEachGenerator	yes	Supported in context of a list comprehension used as a **list** initializer.	6.10.2

continued on next page

(continued)

Element	Support	Comment	CLR
ChooseStmt	no		7.7
ChooseGenerator	no		7.7
Action	yes	Initializer action not supported.	8.
ActionTag	yes		9.1
ActionHead	yes	delay clause not supported.	8.3
InputPattern	yes	The repeat clause is not supported. Actions which read multiple tokens are automatically transformed into multiple actions that read single tokens with additional actor state variables. It may be more efficient to avoid the use of multi-token reads in the source code.	8.1
ChannelSelector	no		8.1.2
RepeatClause	no		8.
OutputExpression	yes	Repeat clauses and channel selectors not supported. Multi-token outputs will require multiple clock cycles to complete.	8.2
InitializationAction	no	Future support likely.	8.5
InitializerHead	no		8.5
ActionSchedule	yes	The regular expression form is not supported.	9.2
ScheduleFSM	yes		9.2.1
StateTransition	yes		9.2.1
ActionTags	yes		9.1
ScheduleRegExp	no		9.2.2
RegExp	no		9.2.2
PriorityOrder	yes		9.3
PriorityInequality	yes		9.3

Chapter 5

Tuning for Performance

This chapter aims to give the CAL programmer some insights into the behavior of the hardware code generator and the impact of coding choices on quality of result (QoR).

The final result in hardware is affected both by the HDL code generated by SYSTEMBUILDER and the results of conventional synthesis, place and route in the FPGA tool flow. Consequently, it can be difficult to predict exactly which optimizations at the source code level will make a real difference in QoR, but an understanding of the automatic code generation strategies at work can be useful.

5.1 Hardware implementation template

This section introduces the generate hardware template that the SYSTEMBUILDER code generator maps a CAL actor description onto. The following actor outline is used to illustrate the template:

```
// Two input, two output actor example
actor sample() int I1, int I2 ==> int O1, int O2

// Two actor state variables
int S1 := ... ;
int S2 := ... ;

// read I1, write O1, modify S1 and S2
A1: action I1:[ x ] ==> O1:[ f1( x, S1, S2 ) ]
  guard f2( S2 )
  do
    ...
    S1 := f3( x, S1, S2 );
    S2 := f4( x, S1, S2 );
  end
```

```
// read I2, write O1 and O2, modify S2
A2: action I2:[ x ] ==> O1:[ f5( x, S2 ) ], O2:[ f6( x, S2 ) ]
  guard f7( x )
  do
    ...
    S2 := f8( x, S1, S2 );
  end

  schedule FSM state1:
    state1 (A1) --> state2;
    state2 (A2) --> state1;
  end

end
```

Figure 5.1 illustrates the various hardware processes that will be generated to implement this sample actor. The major data paths are shown, but control signals have been omitted for the sake of clarity.

Action Scheduler

The action scheduler controls the execution of the actor by causing individual actions to fire (ie. perform an atomic computation, consuming input tokens, producing output tokens, and modifying the actor state). The scheduler must consider the condition of any action guards, action priorities, the actor state and the availability of input tokens before selecting an action to fire.

The current code generator only permits one action to execute at a time, so when an action is fired the scheduler must wait for completion before firing another action. For some actor descriptions this is an overly strict interpretation of the CAL formalism. Future implementations will seek to increase the amount of concurrent execution without violating the guarantee of atomicity inherent in the programming model.

The action scheduler maintains the FSM state variable if FSM control is specified in the source program, and issues the necessary command strobes to execute actions and manage inputs, outputs and state variables. When multiple actions can write to the same resource (state variable or output port), the appropriate source is selected by a multiplexer under the control of the scheduler.

Guards

Hardware to evaluate guard expressions must operate independently of the action that declares the guard, since multiple guard evaluations may be required by the scheduler to determine which action to fire next. When a guard expression uses local variables of an action, these variable values are shared with the action hardware to prevent redundant calculations.

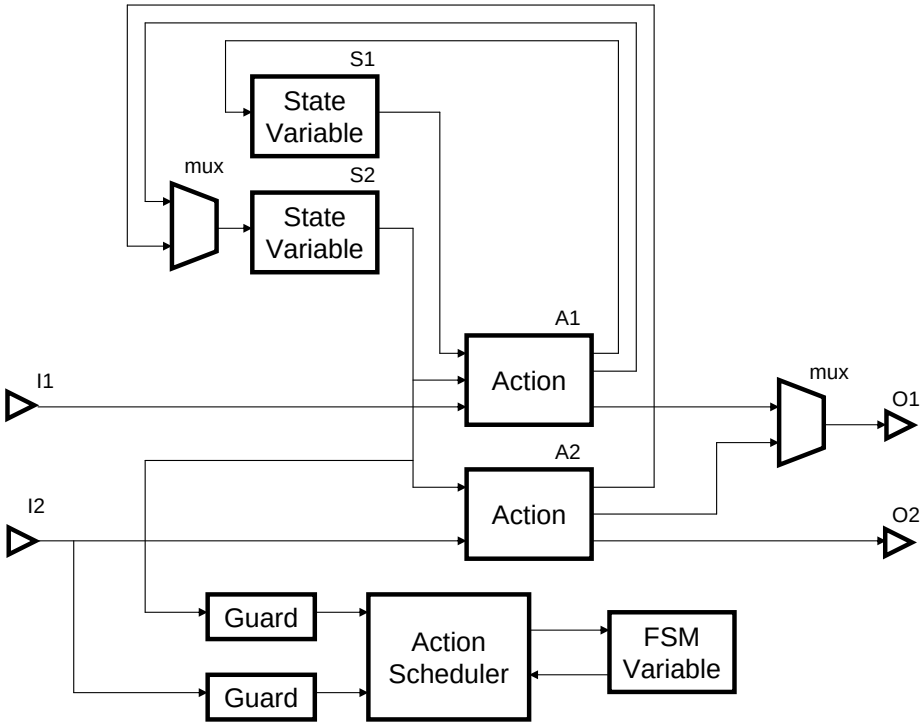

Figure 5.1: Sample actor implementation

Since guard expression evaluation is guaranteed by the language to be side-effect free, the actor state will not be affected by this evaluation even if the related action does not fire at a particular time step.

Ports

Input port values are available for inspection whenever there is a token waiting. The actual consumption of an input token occurs when a consuming action completes its execution. Ouput ports that have more than one writer must be driven by a multiplexer under the control of the action scheduler.

Actor State

Actor state variables must persist between action firings, so the generated hardware includes storage registers or memories to represent these variables. Scalar values (variables of type **int** or **bool**) are held in registers and **list** type variables are stored in independent addressable memories. Registers are updated once at the end of action execution, an memories are written or read immediately, as the source code requires.

Actions

Hardware to implement the computation in individual action bodies, including the calculation of output token values, is under control of the action scheduler.

5.2 Area

The CAL code generator tends to produce HDL that is more or less 'WYSIWYG' - there is no reuse of hardware operators, common subexpression elimination or similar compiler analysis and transformation of the source code. With this in mind, here are a few suggestions for minimizing the size of the generated hardware.

5.2.1 Common subexpressions

When an action has common subexpressions that appear in more than one place, use local variables to hold the subexpression result for reuse. Introducing additional local variables causes no hardware overhead and can save area.

5.2.2 Type sizing

In general it is possible to save FPGA area by maintaining only the required number of bits in an expression. Make sure that state variables, input ports and output ports are sized to be as small as possible for the application. SYSTEMBUILDER does some conservative analysis to detect and remove redundant bits within an expression based on the width of the result destination, so it is

not necessary to specify all sizes throughout the program. The default size of **int**s is 32 bits, so if there are a lot of 32 bit registers or operators being reported after final HDL synthesis, this may be a sign that more explicit sizing would help reduce the circuit size. Any explicit size annotations in the source help the bit-width analysis minimize area.

5.2.3 Reuse common operations

If there is an operation that is performed in more than one action, and which requires a significant amount of hardware to implement, it may be a good idea to put this operation into its own action and then use the FSM capability to execute the operation in more than one context. This also applies to procedures and function calls since they are in-lined, implying replication of hardware.

For example, consider the following code snippet, which invokes some expensive function twice:

```
...

a1: action  some IO signature
var x = expensive_function( args)
do
  ...
end

a2: action  some other IO signature
var x = expensive_function( other args)
do
  ...
end

...

schedule fsm
  ...
  state_x ( a1 ) --> state_y;
  state_p ( a2 ) --> state_q;
  ...
end

...
```

This could be transformed as follows to invoke the expensive function in only one place. This would result in area savings at the expense of introducing extra clock cycles, some multiplexing and more actor state to hold the required inputs and function result.

```
...
```

```
int shared_x;
int shared_arg_1, ... ;

// break a1 into what happens before, after function call
a1_pre: action  some I signature  ==>
do
  ...
end

a1_post: action ==>  some O signature
do
  ...
end

// break a2 into what happens before, after function call
a2_pre: action  some other I signature  ==>
do
  ...
end

a2_post: action ==>  some other O signature
do
  ...
end

op: action
do
  shared_x = expensive_function( shared args );
end

...

schedule fsm
  ...
  state_x      ( a1_pre  ) --> state_x_op;
  state_x_op   ( op      ) --> state_x_post;
  state_x_post ( a1_post ) --> state_y;
  state_p      ( a2_pre  ) --> state_p_op;
  state_p_op   ( op      ) --> state_p_post;
  state_p_post ( a2_post ) --> state_q;
  ...
end

...
```

This example shows how, with some coding effort, it is possible to make the expensive function call appear only once in the source, for a potential area savings.

Look for future implementations of SYSTEMBUILDER to support implementation of procedure or function calls as a shared hardware resource.

5.2.4 Use language control constructs

The programmer should try to move as much of the program control specification into the native CAL actor control constructs as possible. FSM specifications, guards and priority statements used in preference to **if** statements will help reduce the size of the resultant hardware.

Prioiry statements are useful in simplifying guard expressions. For example, if there are two actions with complementary guards, use priority to remove one of the guard expressions entirely:

```
...
a1: action signature
guard f1( args)
do
  ...
end

a2: action signature
guard not f1( args)
do
  ...
end
...
```

This can be simplified as:

```
...
a1: action signature
guard f1( args)
do
  ...
end

a2: action signature
do
  ...
end
...
priority
  a1 > a2;
end
```

. . .

The priority statement has no hardware cost associated with it. One word of caution, though. The lower priority action must consume at least as many tokens as the higher-priority one to avoid making the actor behavior dependent on implementation choices and the context in which it is used. This is because lack of tokens disqualifies an action from consideration in the action scheduler, and our goal in this example is to maintain complementary guard conditions without explicitly coding them. There are valid uses for non-determinism, but in general it is desirable to code actors in a way that makes their behavior insensitive to implementation, timing etc.

non-determinism alert!

5.3 Throughput

This section gives a few suggestions for coding practices that may result in increased performance. Ultimately, though, experimenting with trial implementations will result in the highest QoR.

Higher actor performance can be achieved by increasing the maximum clock rate for an actor, and minimizing the number of action firings needed to process tokens. Dataflow networks can be constructed with multiple actor clocks, but it is often convenient to clock the entire network from a single source. In such cases, the slowest execution path through any action will determine the maximum clock frequency.

5.3.1 Actor granularity

The actor schedulers that are currently synthesized by SYSTEMBUILDER are limited to a single action executing at any one time. This restriction usually means that it is beneficial to keep actors small, and to separate processing out into different actors to maximize concurrency. Actions that do not need to access shared state or inputs should be contained in separate actors.

This is particularly true in video processing systems which are often characterized by complex decision-making at the block level and simple processing at the pixel level. If the block-level control is factored out into its own actor, then multi-cycle decision calculations can procede concurrently with pixel-rate processing in another actor which can achieve throughputs of one or more pixels per action firing (ie clock cycle).

5.3.2 Action execution

Actions take at least one clock cycle to execute. Most expressions execute combinatorially, which limits cycle time but does not add clock cycles. The evaluation of these expressions occurs concurrently to the extent that data dependencies will permit, and the evaluation is eager.

The following activities add additional clock cycles to action execution:

- Memory (ie. **list** variable) read.
- Memory write.
- Multi-token writes - one additional cycle per excess token.
- Division introduces a data-dependent cycle count.
- **while** loop - at least one cycle per iteration.
- Guard evaluation. If any guard expression includes a multi-cycle operation this will add add clock cycles to every action firing in the actor.

5.3.3 Clock frequency

Clock frequency is determined by the longest combinatorial path through any action. Activities that can contribute to combinatorial delay are:

- Guard evaluation. Guard expressions usually evaluate combinatorially and the action scheduler cannot make a decision until all guards have evaluated, so the worst case guard evaluation delay affects all actions.
- Long dependency chain through expressions. A chain of variable dependencies through a number of statements can result in longer delays for action evaluation. This chain ends with the calculation of a modified state variable value or an output token.

  The solution is to break a more complex action into simpler actions. Of course, while this reduces action evaluation time it increases the number of action firings to process tokens. In the video processing example above, this is a worthwhile tradeoff in the case of block rate processing. If the offending action is handling pixel rate processing, it may be necessary to move part of the computation into a separate action. In effect, this is **manual pipelining** of the computation.
- Address calculation. There is usually a combinatorial expression that must be evaluated to determine the address to use in a memory reference. Memory sizes that are powers of two can streamline address calculation hardware.