The SA-C Compiler Dataflow Description

J. P. Hammes, R. E. Rinker, D. M. McClure, A. P. W. Böhm, W. A. Najjar
Colorado State University

June 21, 2001

1 Dataflow Graphs

Dataflow graphs are used internally by the SA-C compiler as a program representation that precedes the VHDL form and can be executed using a token-driven semantics. They represent a low-level, non-hierarchical and asynchronous program. The graphs take into account sequencing but not timing; however, the node functions have been created in such a way as to allow reasonably straightforward translations into the synchronous circuits that finally will be mapped onto reconfigurable hardware.

The functional elements of dataflow graphs are *nodes*, each of which has a node-type, one or more input ports, and one or more output ports. The nodes of a dataflow graph are connected by directed edges, each of which connects an output port to an input port. An output port can connect to any number (including zero) of input ports, but an input port can be fed by only one output port.

When a dataflow graph executes, nodes are "fired" and data are communicated via *tokens*. Every token is created by the output port of a dataflow node. (Data are brought in from outside the dataflow graph through special INPUT nodes.) Each token represents a single k-bit untyped entity, where k is specified by the output port of its source. Each node-type has a firing rule that specifies its behavior when it fires. When a node produces a value at one of its output ports, a token of that value is sent to every input port that is fed by that output. Every input port has a queue of unbounded size. The order of the tokens at an input is the same as the order in which they were produced – i.e., edges behave as queues.

A node may fire only when its firing rule is met. If multiple nodes are ready to fire, they may fire in any order or concurrently. The behavior of most nodes is very simple: it fires only if each of its inputs sees a token. The act of firing consumes one token from each input, and the value of its output is a function of only those inputs. If an input needs a constant value, it is fed by the constant rather than by an output port. The constant input may be viewed as producing a token of that value whenever such a token is needed to fire its node. Tokens have no type, but each has a size as a number-of-bits. The bit-size of an output must match the sizes of each input it feeds. However, the size of a node's input does *not* have to match the other inputs or the outputs of its node. The exact behavior of a node whose inputs and outputs are of various bit-sizes depends on its node-type.

2 Dataflow node descriptions

SA-C dataflow node functions can be classified into six kinds: Arithmetic, Bit, Selector, Generator, Reduction and I/O nodes. Arithmetic nodes perform common functions such as addition, comparison, and logical operations. Bit nodes perform shifts, sub-word selections, and width changing operations. Selector nodes involve choosing one of a number of inputs to pass to the output. Generator nodes take inputs and use them to specify sequences of output tokens. Reduction nodes take token sequences and reduce or store them. I/O nodes handle the interface between the dataflow graph and the outside world. Bit positions for an n-bit value are specified as b_0 (least significant bit) through b_{n-1} (most significant bit). A node has input ports designated I_0 through I_{p-1}, and output ports O_0 through O_{q-1}, where p is its number of inputs and q is its number of outputs. Some node types have a fixed number of inputs, but for some node types the number of inputs and/or outputs can vary.

2.1 Arithmetic nodes

Table 1 shows the names and descriptions of the dataflow nodes that perform arithmetic and bit-manipulation operations. While the tokens themselves have no type, the Arithmetic nodes implicitly treat their inputs as either signed or unsigned integers. The signed values are represented in twos-complement form.

The Arithmetic nodes have straightforward characteristics and semantics:

1. A node can fire only when a token is present on every one of its inputs.
2. The act of firing consumes exactly one token from each input.
3. Every node has exactly one output.
4. The sizes of a node's inputs are unified by extending the precisions of its inputs to match the largest input size. The new bits are placed to the left of the incoming bits. For unsigned values, the new bits are zeroes. For the signed operators, the value is *sign-extended*, that is the new bits are the same as the most-significant bit of the incoming token.
5. After the result value has been computed, the value's size is adjusted to match the specified size of the output port. If the size is reduced, the appropriate number of left-most bits are truncated. If the size is increased, the new bits are zeroes for operators with unsigned outputs, and sign-extended for operators with signed outputs.

The comparison nodes produce a zero if false, and a one if true; the output can be specified to be any bit-size, but only the least significant bit can be non-zero.

2.2 Multi-input Arithmetic nodes

Multi-input arithmetic nodes exist for some operators. They allow an arbitrary number of input values, each with an associated boolean mask value. Various implementations may

name	inputs	description
UADD	2	unsigned addition of values from I_0 and I_1
IADD	2	signed addition of values from I_0 and I_1
USUB	2	unsigned subtraction of values from I_0 and I_1
ISUB	2	signed subtraction of values from I_0 and I_1
NEGATE	1	negate the signed value from I_0
ULT	2	unsigned $<$ comparison of values from I_0 and I_1
ILT	2	signed $<$ comparison of values from I_0 and I_1
ULE	2	unsigned $\leq$ comparison of values from I_0 and I_1
ILE	2	signed $\leq$ comparison of values from I_0 and I_1
UGT	2	unsigned $>$ comparison of values from I_0 and I_1
IGT	2	signed $>$ comparison of values from I_0 and I_1
UGE	2	unsigned $\geq$ comparison of values from I_0 and I_1
IGE	2	signed $\geq$ comparison of values from I_0 and I_1
UEQ	2	unsigned equality comparison
IEQ	2	signed equality comparison
UNE	2	unsigned inequality comparison
INE	2	signed inequality comparison
BIT-AND	2	bit-wise AND of values from I_0 and I_1
BIT-OR	2	bit-wise OR of values from I_0 and I_1
BIT-EOR	2	bit-wise exclusive OR of values from I_0 and I_1
BIT-COMPL	1	bit-wise complement of value from I_0

Table 1: Arithmetic nodes

choose the order in which the values are reduced. The Multi-input arithmetic nodes are shown in Table 2.

2.3 Multi-input Comparison Nodes

Table 3 lists the multi-input comparison nodes. These nodes have input ports organized in clusters of $n + 2$ where n is the number of output values/ports. The first input port in each cluster is a comparison value, the next n ports are *captured* values, and the last input port in each cluster is a boolean mask. For each cluster with a boolean mask of true, it's comparison value is compared with all other comparison values to find the [*first* or *last*] [*max* or *min*] of all comparison values (depending on type of node). The output of the node is the n captured values associated with the selected comparison value.

An example may help illustrate the functionality of these nodes. Figure 1 shows an example VAL-AT-FIRST-UMAX-MANY node. For this example there are $n = 2$ captured values per comparison value, so the input ports appear in clusters of $n + 2 = 4$. Since this is a VAL-AT-FIRST-UMAX-MANY node, all comparison values with a boolean mask of true will be compared to find the leftmost maximum among them. The output of the node is the two captured values associated with the selected comparison value.

name	inputs	description
USUM-MANY	var	sum the unsigned input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
ISUM-MANY	var	sum the signed input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
UMIN-MANY	var	'min' the unsigned input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
IMIN-MANY	var	'min' the signed input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
UMAX-MANY	var	'max' the unsigned input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
IMAX-MANY	var	'max' the signed input values; each input pair represents a value and a boolean mask; the arithmetic is performed at a bit width equal to the output port's bit width
UMEDIAN-MANY	var	find the median of the unsigned input values; each input pair represents a value and a boolean mask
IMEDIAN-MANY	var	find the median of the signed input values; each input pair represents a value and a boolean mask
AND-MANY	var	'and' the input values; each input pair represents a value and a boolean mask
OR-MANY	var	'or' the input values; each input pair represents a value and a boolean mask

Table 2: Multi-input Arithmetic nodes

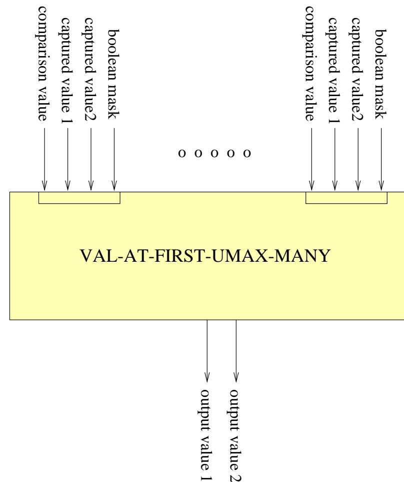

Figure 1: Example Multi-Input Comparison node (a VAL-AT-FIRST-UMAX-MANY node with n=2)

2.4 Bit nodes

The Bit nodes perform shifts, concatenates, sub-word selects and size-changing operations. Table 4 describes the Bit nodes, which have the following characteristics and semantics:

name	inputs	description
VAL-AT-FIRST-IMAX-MANY	var	return the signed *captured* value or values at the first max among many *comparison* values (see Figure 1). Input ports are organized in clusters of $n+2$ where n is the number of *captured* values per cluster (equal to the number of output ports/values). Input port I_0 in each cluster is the comparison value; ports I_1 to I_n in each cluster are the *captured* values; port I_{n+1} in each cluster is a boolean mask. Output ports O_0 to O_{n-1} carry the captured values at the first max comparison value with a boolean mask of true.
VAL-AT-FIRST-UMAX-MANY	var	the same as VAL-AT-FIRST-IMAX-MANY but with **unsigned** comparison and captured values.
VAL-AT-FIRST-IMIN-MANY	var	the same as VAL-AT-FIRST-IMAX-MANY, but the output ports carry the captured values at the first **min** comparison value with a boolean mask of true.
VAL-AT-FIRST-UMIN-MANY	var	the same as VAL-AT-FIRST-IMIN-MANY, but with **unsigned** comparison and captured values.
VAL-AT-LAST-IMAX-MANY	var	the same as VAL-AT-FIRST-IMAX-MANY, but the output ports carry the captured values at the **last** max comparison value with a boolean mask of true.
VAL-AT-LAST-UMAX-MANY	var	the same as VAL-AT-LAST-IMAX-MANY, but with **unsigned** comparison and captured values.
VAL-AT-LAST-IMIN-MANY	var	the same as VAL-AT-FIRST-IMIN-MANY, but the output ports carry the captured values at the bf last min comparison value with a boolean mask of true.
VAL-AT-LAST-UMIN-MANY	var	the same as VAL-AT-LAST-IMIN-MANY but with **unsigned** comparison and captured values.

Table 3: Multi-input Comparison nodes

1. A node can fire only when a token is present on every one of its inputs.
2. The act of firing consumes exactly one token from each input.
3. Every node has exactly one output.
4. After the result value has been computed, the value's size is adjusted to match the specified size of the output port. If the size is reduced, the appropriate number of leftmost bits are truncated. For the CHANGE-WIDTH-SE node, if the size is increased, the most significant (leftmost) bit is sign-extended. For all other nodes, the size is increased by padding zeroes to the left of the value.

2.5 Selector node

Currently, there is only one selector node, called SELECTOR, that corresponds to the switch of a high-level language. Its first input, I_0, takes the key value that is used to choose among the input values. Each case value uses two consecutive inputs, the first as the match value and the second as the chosen expression. The default expression is the last input, and it must be present. The bit width of every key compare input is guaranteed to equal the width of key input (I_0), so an absolute compare of the bit strings can be made without the

name	inputs	description
L-SHIFT	2	left-shift the value from I_0 by the distance from I_1; zeroes enter from the right
R-SHIFT	2	right-shift the value from I_0 by the distance from I_1; zeroes enter from the left
BIT-SELECT	3	I_0 and I_1 specify a range of bits to be taken from the value on I_2
BIT-CONCAT	2	I_0 and I_1 are concatenated to form a result whose width is the sum of the two input widths
CHANGE-WIDTH	2	change the bit-width of the value on I_0 to the *constant* size on I_1
CHANGE-WIDTH-SE	2	change the bit-width of the value on I_0 to the *constant* size on I_1 by sign-extending it if the size is increased

Table 4: Bit nodes

need for considering sign extension. Also, the bit width of every value input is guaranteed to equal the bit width of the output. The following node corresponds to the SA-C switch expression (assume that "key" is a `int7`):

```
switch (key) {
    case -1   : return (E0)
    case 3,-7 : return (E1)
    case 2    : return (E2)
    default   : return (E3) }
```

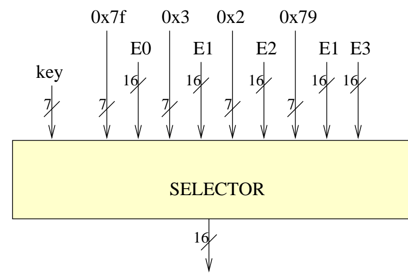

Figure 2: Selector node example.

Except for the default expression, which must be the last input, the order of the <value,expression> pairs does not matter. Duplicate case values are not allowed.

2.6 Generator nodes

Generator nodes produce streams of scalar tokens. There are four basic types of generator nodes:

1. Scalar generators produce sequences of integers.

2. Element generators produce streams of values drawn from memory.

3. Window generators produce streams of values drawn from specified windows of a source array read from memory.

4. Slice generators produce streams of values drawn from specified slices of a source array read from memory.

Slice generators use only 2D arrays as source arrays; the other generator nodes can use 1-, 2- or 3-D source arrays, with the potential to use arrays of up to 8-D if that functionality should be needed. The various generator nodes are listed in Table 5. [1]

An example 2-D window generator node is shown in Figure 3. Input port 0 carries the start address of the source array from which windows of values will be read. The input ports then come in clusters of three for each dimension of the window – with one port each carrying the window extent, step size, and source array extent in that dimension. This is followed by $(dims - 1)$ ports carrying the image offsets, where $dims$ is the number of dimensions of the window (2-D for the example in Figure 3). The last input port carries the the number of dummy iterations. There are $(num_els + dims + 2)$ output ports, where num_els is the number of elements in a single window. Output ports 0 to $(num_els - 1)$ carry the values read from the current window in the source array memory, the next two ports carry the indices of that window in the source array, while the last two ports carry the *dummy* and *done* signals, indicating whether the current iteration is a dummy and when all iteration have been executed respectively.

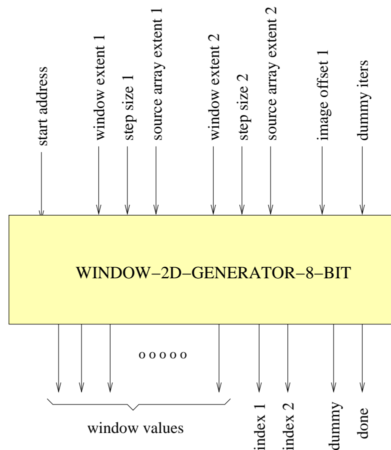

Figure 3: Generator node example.

[1] In several of the tables that follow, entire familes of nodes are represented by using node names with one or more parameter placeholders. For the ELEMENT GENERATOR nodes in Table 5, for example, β represents bit-size (which can take on values of 1, 2, 4, 8, 16, or 32), and δ represents dimensionality of the source array (which can take on values of 1, 2 or 3). Consequently, ELEMENT-δD-GENERATOR-β-BIT represents a family of $3 \times 6 = 18$ related nodes. As this node-naming convention is used in the tables that follow, each parameter's meaning and its possible values will be specified.

name	ins	outs	description
SCALAR-GENERATOR	$3\delta+1$	$\delta+2$	generate δ streams of integer tokens for a generator with $rank = \delta$ ($\delta = 1$, 2 or 3); the δ input triples carry the start value, end value, and step size used to generate the output stream, and the last input port carries the dummy iterations; output ports O_0 to $O_{\delta-1}$ carry the output stream, and the last two output ports have the dummy and done signal.
ELEMENT-δD-GENERATOR-β-BIT	$3\delta+2$	$\delta+3$	generate a stream of scalar tokens constituting a δ-dimensional array ($\delta = 1$, 2 or 3) read from memory in β-bit units ($\beta = 1$, 2, 4, 8, 16 or 32); I_0 carries the starting address of the array in source memory, the next δ input pairs carry the step size and the array's extent in each of the δ dimensions, the next $(\delta - 1)$ input ports carry the image offsets, last two input ports carry image size in words and dummy iterations; ouput O_0 produces the stream of values constituting the array, outputs 0_1 to 0_δ carry indices of the current element in the δ-dimensional array, the last two output ports have the dummy and done signal.
WINDOW-δD-GENERATOR-β-BIT	$4\delta+2$	$n+\delta+2$	generate n streams of scalar tokens, for the n elements in that part of a source array currently under a moving δ-dimensional window ($\delta = 1$ or 2 or 3), those elements to be read from memory in β-bit units ($\beta = 1$, 2, 4, 8, 16 or 32); input port I_0 carries the starting address of the source array, the next δ input triples carry the window size, step size, and array extent in one of the window's δ dimensions, the next $(\delta - 1)$ input ports carry the image offsets, last two input ports carry image size in words and dummy iterations; output ports O_0 to O_{n-1} carry the scalar elements of that part of the source array currently under the moving window, the next δ output ports carry indices of the top-left element of the current window in the δ-dimensional array, the last two output ports have the dummy and done signal.
SLICE-2D-GENERATOR-ROW-β-BIT	7	$n+3$	generates n streams of scalar tokens for the n β-bit elements in a specified row slice of a 2D array read from memory. Input I_0 carries the start address of the source array, I_1 carries the step size between slices to be generated, I_2 and I_3 carry the extents of the 2D source array, I_4 carries the image offset, while I_5 and I_6 carry image size in words and dummy iterations; outputs O_0 to O_{n-1} carry the values of the array elements in the current slice, O_n carries the index of the current row slice in the 2D source array, and the last two output ports have the dummy and done signal.
SLICE-2D-GENERATOR-COL-β-BIT	7	$n+3$	same as SLICE-2D-GENERATOR-ROW-β-BIT but the slice elements are from a column (rather than a row);

Table 5: Generator nodes

2.7 Array Reference nodes

Array reference nodes return the value of a single array element. The input ports of an ARRAYREF node carry the information needed to locate an array element in memory (start address, array extents, and indices witin the array of the element being referenced); while the single output port carries the value of the array element thus referenced. ARRAYREF nodes are currently implemented for 1-, 2- and 3-D arrays, but could be implemented for arrays of up to 8-D should that functionality be needed. The currently implemented

ARRAYREF nodes are listed in Table 6, the single entry of which represents a family of $3 \times 6 = 18$ nodes (see footnote 1, above).

name	ins	outs	description
ARRAYREF-δD-β-BIT	3δ+1	1	generate a token with the value of a β-bit element ($\beta = 1, 2, 4, 8, 16$ or 32) of a δ-D array ($\delta = 1, 2$ or 3); input port I_0 carries the start address of the source array, ports I_1 to I_δ carry the indices of the referenced element in the source array, ports $I_{\delta+1}$ to $I_{2\delta}$ carry the extents of the source array in its δ dimensions, ports $I_{2\delta+1}$ to $I_{3\delta-1}$ carry the image offsets, while port $I_{3\delta+1}$ carries the image size in words; output port O_0 carries the value of the array element being referenced;

Table 6: Array Reference nodes.

2.8 Reduction nodes

Reduction nodes handle values produced by a loop body, reducing those values by a corresponding arithmetic function. Currently implemented reduction nodes are listed in Tables 7 and 8. An example reduction node is shown in Figure 4. The functionality of this node is as follows: a stream of *value*s from a source array enters the node (on input port I_0), along with a *label* (integers beginning from 0, on port I_1) identifying from which region of the source array the current value is drawn, and a boolean *mask* (on port I_3) indicating whether the current value should be considered when determing the MIN for its particular region. The Accum-Umin-Values node determines the MIN of all non-duumy iteration values for each region of the source array, and writes to memory a 1-D array holding those minimum values (extent of output array == number of labeled regions in the input array).

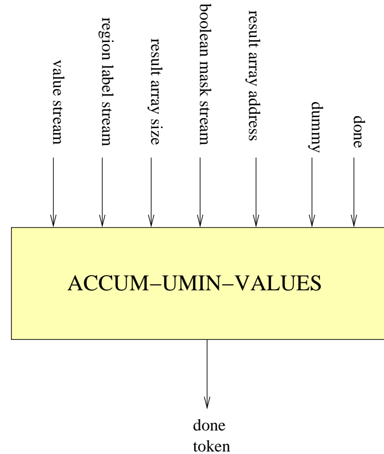

Figure 4: Reduction node example.

name	ins	outs	description
USUM-VALUES	5	1	sum a stream of unsigned tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
ISUM-VALUES	5	1	sum a stream of signed tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
UMIN-VALUES	5	1	min a stream of unsigned tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token done;
IMIN-VALUES	5	1	min a stream of signed tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
UMAX-VALUES	5	1	max a stream of unsigned tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
IMAX-VALUES	5	1	max a stream of signed tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
AND-VALUES	5	1	'and' a stream of tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
OR-VALUES	5	1	'or' a stream of tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the address to store the result, I_3 and I_4 are the dummy and done signals; O_0 emits a TRUE token when done;
HIST-VALUES	6	1	create a histogram from a stream of tokens; I_0 is the value token stream, I_1 is the mask value stream, I_2 is the result array size, I_3 is the result array start address, I_4 and I_5 are the dummy and done signals; O_0 emits a TRUE token when done;
ACCUM-UMIN-VALUES	7	1	write to a result array in memory the minimum unsigned value in each of several labeled regions of a source array streamed into port 0 (result array size = number of labeled regions in source array); input I_0 is the stream of values from the source array, I_1 is the region number for the current value on I_0, I_2 is the result array size (number of labeled regions in source array), I_3 is a boolean mask stream, I_4 is the address to which the result array should be written, I_5 and I_6 are the dummy and done signal; output O_0 emits a TRUE token when the node's processing is complete;
ACCUM-IMIN-VALUES	7	1	same as ACCUM-UMIN-VALUES but with signed values;
ACCUM-UMAX-VALUES	7	1	same as ACCUM-UMIN-VALUES but with maximum values;
ACCUM-IMAX-VALUES	7	1	same as ACCUM-UMAX-VALUES but with signed values;
ACCUM-USUM-VALUES	7	1	same as ACCUM-UMIN-VALUES but sum the values in each region of the source array rather than finding their min;

Table 7: Reduction nodes

name	ins	outs	description
ACCUM-ISUM-VALUES	6	1	same as ACCUM-USUM-VALUES but with signed values;
ACCUM-AND-VALUES	6	1	same as ACCUM-UMIN-VALUES but AND the values in each region of the source array rather than finding their min;
ACCUM-OR-VALUES	6	1	same as ACCUM-AND-VALUES but OR the values in each region of the source array;
ACCUM-HIST-VALUES	7	1	same as ACCUM-UMIN-VALUES, but create a histogram for each labeled region of the source array; input I_0 is the stream of values; I_1 is the stream of region labels in the source array; I_2 and I_3 are the 2-D result array's extents; I_4 is a stream of boolean mask values; I_5 is the result array start address; I_6 is the number of values in the incoming stream; output port O_0 emits a TRUE token when the node's processing is complete;

Table 8: Reduction nodes (continued)

2.9 Circulate nodes

CIRCULATE nodes are associated with loops in the source code. Since these loops are unrolled in a dataflow graph, communcating altered values from one iteration of a loop to the next could be problematic. CIRCULATE nodes solve this problem by delivering such accumulation or *nextified* variables from the exit of a loop, back to the point within the loop where the variable is to be used in the next iteration. In while-loops, the circulate nodes are broken down as two nodes: MERGE and STEER nodes. CIRCULATE nodes and STEER nodes are the only node types that generate backward-directed edges in SA-C dataflow graphs.

CIRCULATE node specifics are given in table 9 while MERGE node and STEER node specifics are given in table 10. A data flow graph containing a CIRCULATE node, and the SA-C code that generated it, are given in Section 4 below.

name	ins	outs	description
CIRCULATE	4	2	circulate an accumulation (or *nextified*) variable from the exit point of an unrolled loop back to the point within the loop where the variable is next used; input port I_0 receives the current value of the nextified variable after each iteration, I_1 holds the initial value of the nextified variable, I_2 and I_3 hold the dummy and done signals; the final value of the nextified variable is delivered on output port O_0 upon completion of the loop, a back edge from port O_1 delivers the current value of the nextified variable back to the point within the loop where it is next used every iteration;

Table 9: Circulate node.

name	ins	outs	description
MERGE	2	1	select between the current iteration value and the initialization value; input port I_0 receives the current value of the nextified variable after each iteration, I_1 holds the initial value of the nextified variable; O_0 delivers the current value of the nextified variable to the STEER node and also for the calculation of the termination condition of the while-loop (if required).
STEER	2	2	steers the current value of the nextified variable as the final value or on the backedge to be used by the next iteration; input port I_0 receives the current iteration value from a MERGE node, I_1 receives the done signal; the final value of the nextified variable is delivered on output port O_0 upon completion of the loop, a back edge from port O_1 delivers the current value of the nextified variable back to the point within the loop where it is next used every iteration;

Table 10: Merge and Steer nodes.

2.10 Write nodes

There are four basic types of write nodes – WRITE-SCALAR nodes, WRITE-TILE nodes, WRITE-ARRAY nodes and WRITE-VALS-TO-BUFFER.

2.10.1 Write-Scalar nodes

Currently, WRITE-SCALAR nodes are always associated with CIRCULATE nodes (though this could change in the future). In the current implementaion, when the final value of a *nextified* variable (see previous section) is to be returned by a dataflow graph, a CIRCULATE node delivers that value to a WRITE-SCALAR node upon completion of the loop, and the WRITE-SCALAR node writes the value to the appropriate location in memory.

WRITE-SCALAR node specifics are given in Table 11.

name	ins	outs	description
WRITE-SCALAR	2	1	write to memory the final value of a *nextified* variable upon completion of a loop; input port I_0 carries the value to be written; port I_1 carries the address to be written to; output port O_0 delivers a boolean TRUE token when writing is complete;

Table 11: Write-Scalar node.

2.10.2 Write-Tile nodes

WRITE-TILE nodes write results of m-dimensional loops to target memory at each non-dummy iteration. The result per iteration is an n dimensional subarray that is tiled. (Thus, the rank of the output array written to memory is equal to the greater of m and n.) To fully specify a WRITE-TILE node, both the dimensionality of the loop and the rank of the

tile produced must be specified. Currently implemented WRITE-TILE nodes are listed in Table 12, each entry of which represents a family of $3 \times 6 = 18$ related nodes.

Figure 5 shows an example WRITE-TILE node for which the original loop is 2-dimensional and the tile to be written for each element in that array is 3-dimensional. Since the loop is two-dimensional, two input ports carry the loop iterations (ports 9-10). Since the tile to be written for each element has 3 dimensions, three input ports carry the tile's extents (ports 11-13) and two input ports carry the image offsets (ports 14-15). For this example the tile has an extent of 2 in each of the three dimensions, thus requiring that each tile have eight elements. Thus eight input ports carry the incoming stream of tile values (ports 0-7). Port 8 carries the start address of the output array being written while the last two ports carry the dummy and done signal. Output port 0 emits a TRUE token when all the tiles have been written to memory (that is, it receives the done token).

name	ins	outs	description
WRITE-TILE-δD-1D-β-BIT	$n + 2\delta + 2 + l$	1	for each iteration of a δ-dimensional loop ($\delta = 1$, 2 or 3), write a 1-dimensional tile of β-bit tokens ($\beta = 1$, 2, 4, 8, 16 or 32) to memory; I_0 to I_{n-1} carry the n values in the current tile, I_n is the start address, I_{n+1} to $I_{n+\delta}$ carry the iteration counts of the δ-dimensional loop, $I_{n+\delta+1}$ carries the extents of the 1D tile, the next ports carry $l(= max(\delta, 1) = \delta) - 1$ image offsets, the dummy and done signals; output port O_0 emits a TRUE token when writing is complete;
WRITE-TILE-δD-2D-β-BIT	$n + \delta + 4 + l$	1	for each iteration of a δ-dimensional loop ($\delta = 1$, 2 or 3), write a 2-dimensional tile of β-bit tokens ($\beta = 1$, 2, 4, 8, 16 or 32) to memory; inputs I_0 to I_{n-1} (where $n = x \times y$, with x and y the dimensions of the tile) carry the values in the current tile, I_n is the start address, I_{n+1} to $I_{n+\delta}$ carry the iteration counts, $I_{n+\delta+1}$ and $I_{n+\delta+2}$ are the extents of the 2D tile, the next ports carry $l(= max(\delta, 2)) - 1$ image offsets, the dummy and done signals; output port O_0 emits a TRUE token when writing is complete;
WRITE-TILE-δD-3D-β-BIT	$n + \delta + 5 + l$	1	for each iteration of a δ-dimensional loop ($\delta = 1$, 2 or 3), write a 3-dimensional tile of β-bit tokens ($\beta = 1$, 2, 4, 8, 16 or 32) to memory; inputs I_0 to I_{n-1} (where $n = x \times y \times z$, with x, y and z the dimensions of the tile) carry the values in the current tile, I_n is the start address, I_{n+1} to $I_{n+\delta}$ carry the iteration counts, $I_{n+\delta+1}$ to $I_{n+\delta+3}$ are the extents of the 3D tile, the next ports carry $l(= max(\delta, 3)) - 1$ image offsets, the dummy and done signals; output port O_0 emits a TRUE token when writing is complete;

Table 12: Write-Tile nodes.

2.10.3 Write-Array nodes

WRITE-ARRAY nodes write results of m-dimensional loops to target memory at each non-dummy iteration. The result per iteration is an n dimensional subarray that is not tiled. (Thus, the rank of the output array written to memory is equal to the sum of m and n.) The WRITE-ARRAY node structure is identical to that of a WRITE-TILE. They differ in the way they produce the output array. Currently, WRITE-ARRAY-1D-1D, WRITE-ARRAY-

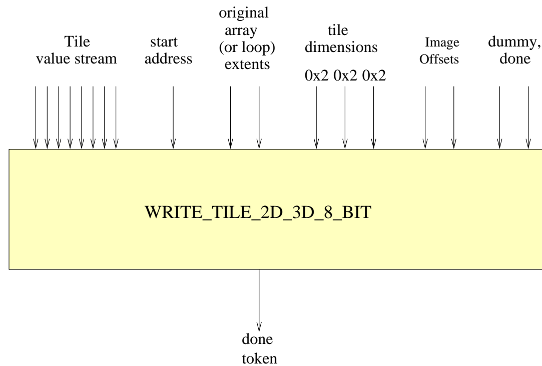

Figure 5: Write-Tile node example.

1D-2D and WRITE-ARRAY-2D-1D exists at the DFG level. Note that for a Write-Array node $l = m + n$.

2.10.4 Write-Vals-To-Buffer nodes

WRITE-VALS-TO-BUFFER nodes write results of one-dimensional loops when the output array size is unknown at run-time. The non-masked values are written into a buffer (in the memory). When the buffer is full, the host is signalled to empty the buffer. A counter stores the number of values currently written in the buffer. (This counter is reset when the buffer is emptied).

WRITE-VALS-TO-BUFFER node specifics are given in Table 13.

name	ins	outs	description
WRITE-VALS-TO-BUFFER	$2*n+5$	1	write to buffer the non-masked values every iteration of a 1D loop; n pairs of inputs contain values and their masks, input port I_{2n} carries the buffer size, I_{2n+1} carries the start address of the buffer, I_{2n+2} has the counter address in the memory while I_{2n+3} and I_{2n+4} are the dummy and done signals; output port O_0 delivers a boolean TRUE token when writing is complete;

Table 13: Write-Vals-To-Buffer node.

2.11 I/O nodes

I/O nodes handle the interface with the outside world. Each node specifies a *channel* (input or output channel, as appropriate), designated by an input constant. I/O nodes are listed in Table 14.

name	ins	outs	description
INPUT	0	1	get a value from the input channel *constant* specified by the corresponding input port of the DFG wraping node.
OUTPUT	1	0	take a value token from I_0; A TRUE token indicates completion of execution of the DFG.

Table 14: I/O nodes

2.12 LoopInfo node

LoopInfo nodes provide the iteration counts for each dimension of the loop. They also provide the provide the product of iteration count of the current dimension to the innermost dimension. For an n-dimensional loop have $2n$ input ports: I_0 to I_{n-1} carry the iterations per dimension of the loop while I_n to I_{2n-1} carry the multiplied iterations. ($I_{n+i} = \prod_{k=i}^{n-1} I_k$). LoopInfo node is listed in Table 15.

name	ins	outs	description
LOOPINFO	$2n$	0	Provide iteration counts for an n-dimensional loop; I_0 to I_{n-1} carry the iterations per dimension of the loop while I_n to I_{2n-1} carry the multiplied iterations.

Table 15: LoopInfo node

3 Dataflow file format

To allow easy inspection and interfacing of dataflow graphs with the tools and programs that use them, a text representation is used. This representation is identical to the DDCF format.
The zero-th node of the DFG is a DFGWRAP node. It is a compound node that contains the actual DFG graph to be executed. The inputs to this wrapper node are compile-time constants that correspond to the addresses in the source memory where the run-time constants are stored. The DFGWRAP node contains a single output node. An arrival of a TRUE token on the input port of this node (during simulation) indicates completion of execution of the DFG.
For simulation, the DFG (in the DDCF format) is converted into internal DFG representation. The $0th$ DFGWRAP node is removed and all the node numbers are shifted by one. Also the input nodes get one input port carrying the compile-time constant address location corresponding to the run-time constant they must provide.

4 Example Dataflow Graphs

4.1 DFG with Unrolled Loop

Consider the following SA-C program:

```
int16[:,:] main (uint8 Image[:,:]) {

   int16 H[3,3] = {  {-1, 0, 1},
                     {-1, 0, 1},
                     {-1, 0, 1}} ;

   int16 R[:,:]  =   for window W[3,3] in Image {
                        int16 iph =  for h in H dot w in W
                                     return( sum(h*w) );
                     } return( array(iph) );
} return(R);
```

This code performs the convolution of a 3×3 mask over over a large input `Image` array, as one might see in an edge detection routine – e.g. the Prewitt edge detection algorithm.

As shown in Figure 6, both loops of this nested structure are converted to a single dataflow graph by the SA-C compiler. The Window-Generator node near the top of this graph reads elements from a 3×3 window of the `Image` array at each iteration, and as this data flows through the graph, the necessary convolution is performed. Notice the multiplies explicit in the code have been removed by the compiler, replaced with either *no-ops* (for multiplication by 0) or *negate* operations (for multiplication by -1). Thus, the nine multiplies and eight additions explicit in the code at each iteration of the for-loop, have been replaced with three negation operations and five additions at each iteration.

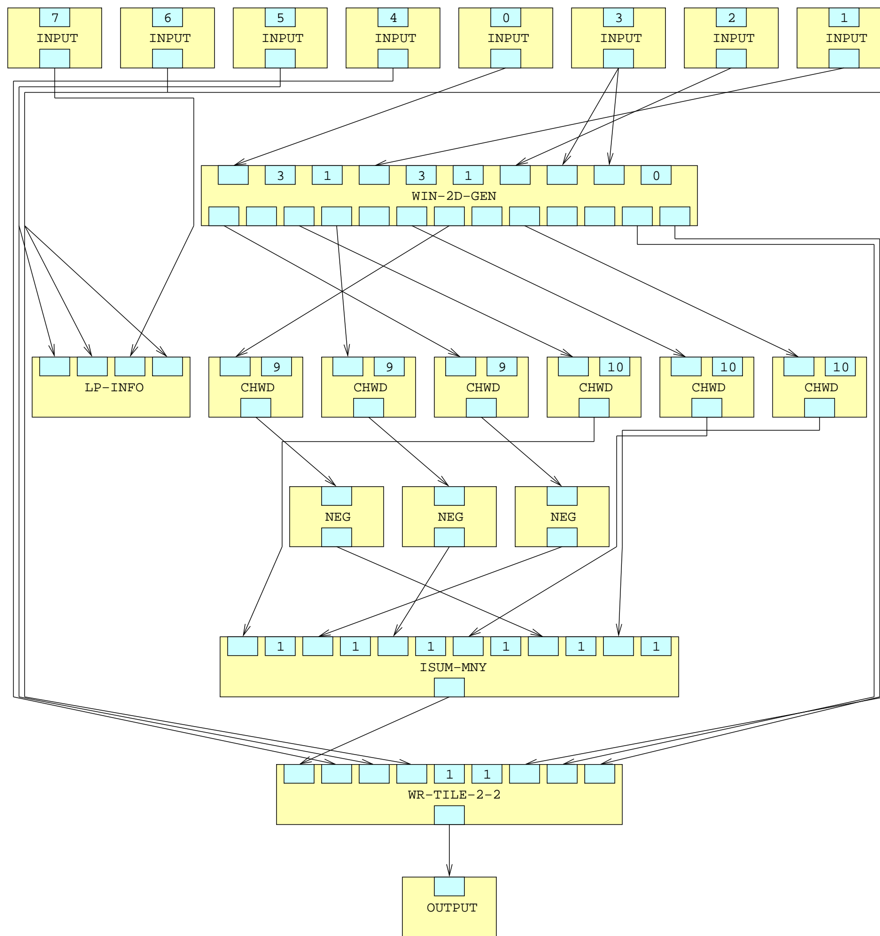

Figure 6: Dataflow Graph with unrolled loop.

4.2 DFG with Circulate Nodes and Back-edges

The following SA-C routine uses a *nextified* variable `count`, and thus generates a dataflow graph containing a CIRCULATE node – the only node type in SA-C dataflow graphs, recall, which generate back-edges. Nextified variables allow for the accumulation of some value from one iteration of an unrolled loop to the next.

```
uint16 main (uint8 Image[:,:], uint8 val) {
   uint16 count = 0;

   uint16 occurrence  =
```

```
            for el in Image {
               next count = ( el == val ? count + 1 : count );
            } return( final(count) );
         } return(occurrence);
```

This routine simply counts the number of occurrences of the value `val` in the input array `Image`.

Figure 7 shows the dataflow graph generated from this code. The current value of the nextified variable `count` is circulated back up the graph at each iteration of the loop via back-edges from the CIRCULATE node's output port O_1. When the loop terminates, the final value of this variable leaves output port O_0 and is written to memory by the WRITE-SCALAR node. The SELECTOR node serves as a conditional statement: if the comparison perfomed by the UEQ node is false (not equal to the 1 on the SELECTOR node's inport I_1), the current value of `count` is selected and passed to the CIRCULATE node; if the comparison is true, an incremented version of `count`, entering inport I_2 from the UADD node above, is selected and passed to the CIRCULATE node.

4.3 DFG with VAL_AT_FIRST_UMAX_MANY node

The following SA-C code implements a part of the *dilation* edge smoothing algorithm. A small `kernel` array is passed over a larger `src` image array, overlapping elements of the two arrays are summed, and the max sum within the current boundary of the kernel is selected. A 2×2 kernel was used for this example to keep the DFG generated to a manageable size; a more typical kernel size is 5×5.

```
uint8[:,:] main(uint8 src[:,:], uint8 kernel[2,2])
{
   uint16 r, uint16 c = extents(kernel);
   uint8 result[:,:] =
      for window win[r,c] in src
      {
         uint8 maxvals[:] =
            for elem1 in win dot elem2 in kernel
            {
               bool b = (elem2 != 0);
            } return(vals_at_first_max(elem1+elem2,{elem1},b));
         uint8 maxval = maxvals[0];
      }return(array(maxval));
}return(result);
```

Figure 8 shows the dataflow graph generated from this code. Notice the four additions required for each position of the kernel are done in parallel by the four UADD nodes on level 3 of the DFG (for a 5×5 kernel, of course, twenty-five additions would be done in parallel). The first (leftmost) maximum among these sums is selected by the VAL-AT-FIRST-UMAX-MANY node on level 4, and written to memory by the WRITE-TILE node below.

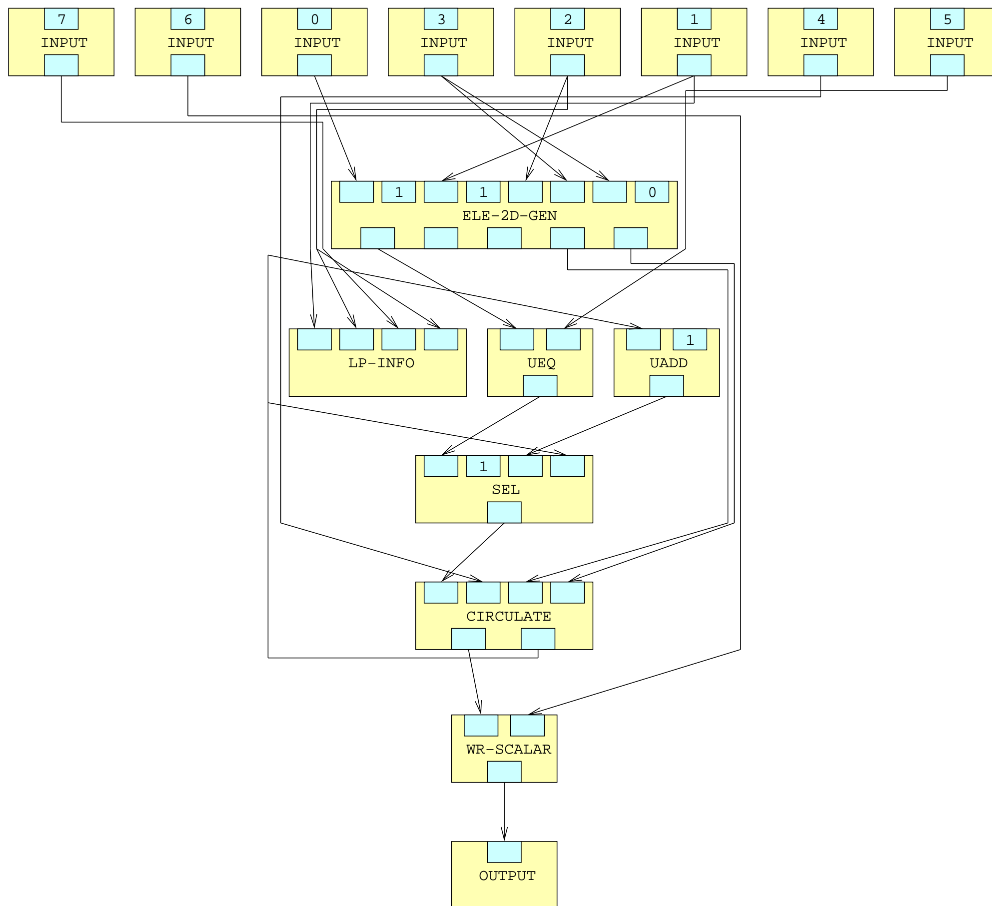

Figure 7: Dataflow Graph with CIRCULATE node and back-edges.

5 A Complete Example

Our final example demonstrates the transformation of a SA-C program from source code to data dependency and control flow (ddcf) graph, and the RC_COMPUTE node within that ddcf graph which contains the dataflow graph (DFG). Consider the following SA-C program:

```
uint8[:] main (uint12 A[:], uint12 x) {
    uint12 R[:] =
        for window W[4] in A
```

```
        return (array (x + (uint12) array_sum (W)));
} return (R);
```

The compiler will unroll the implicit loop created by "array_sum", and enclose it in an RC_COMPUTE node that contains the loop and the host-RCS interface. This is shown in Figure 9. The RC_FORALL graph is then converted to a DFG, as shown in Figure 10. The text representation of this DFG is shown in Figure 11 and 12.

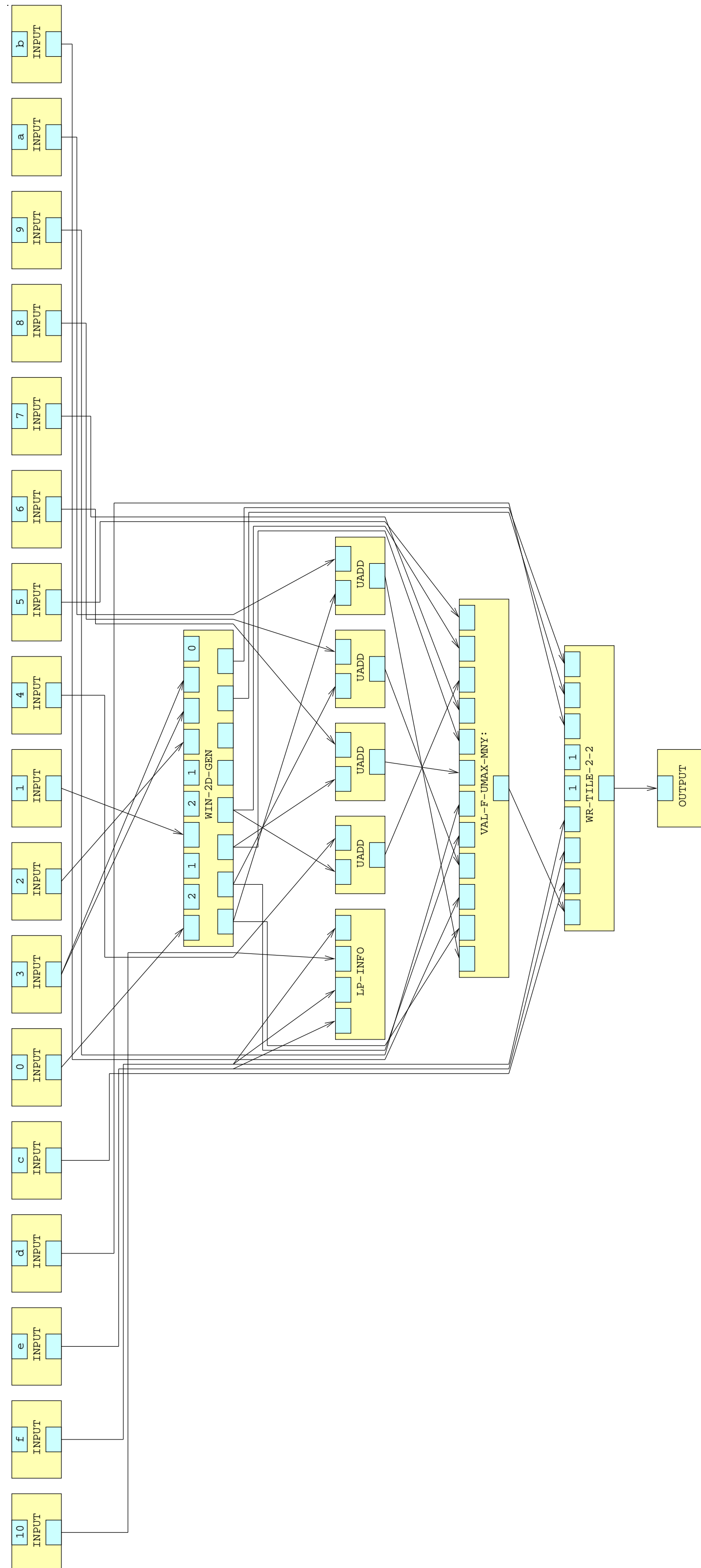

Figure 8: Dataflow Graph with VAL-AT-FIRST-MAX node.

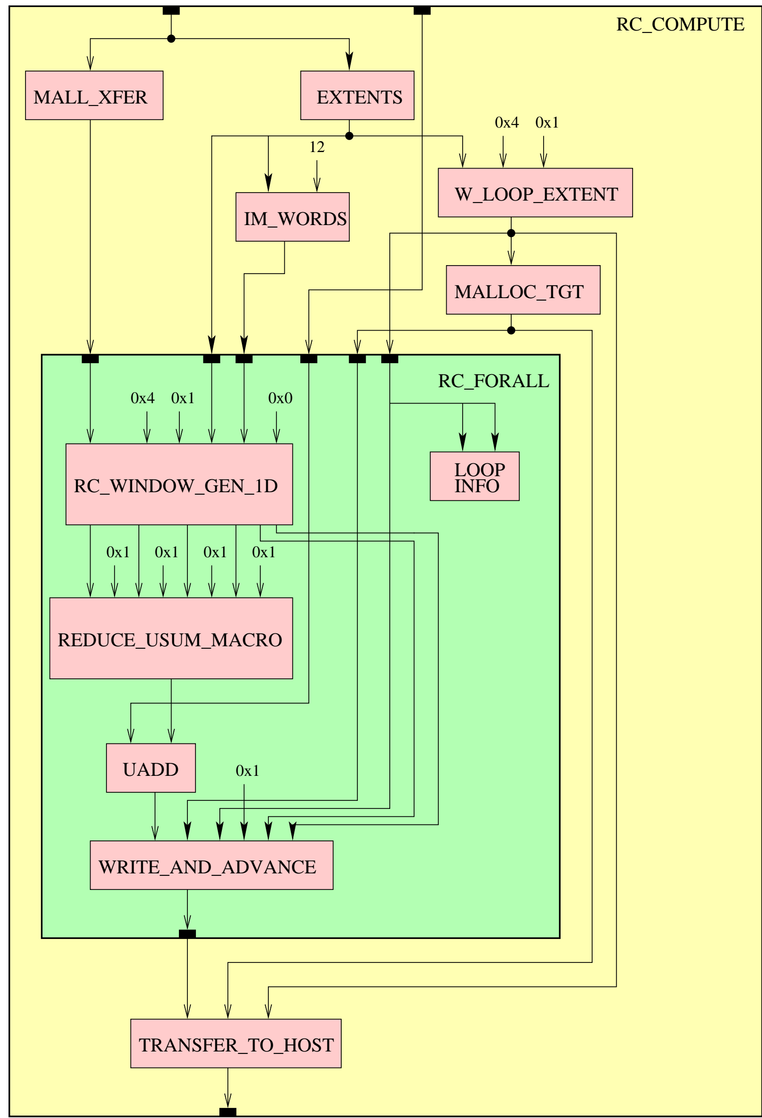

Figure 9: DDCF graph of converted loop with interface.

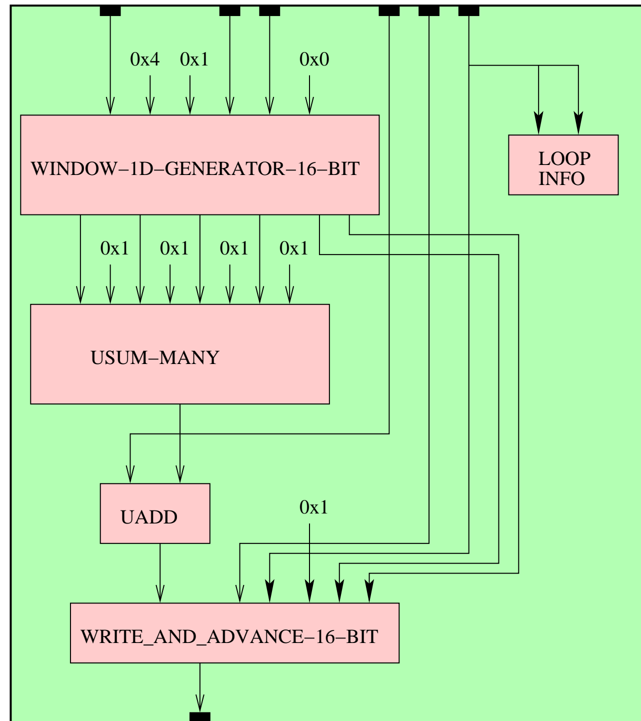

Figure 10: DFG converted loop.

```
dfg "tst_0"
  0 ND_DFG_WRAP (my nodes: 1 2 3 4 5 6 7 8 9 10 11) <"tst.sc", "main", 3>
    6 inputs: (nodes  4 6 5 9 3 2)
      port 0 <bits32> value "0x0"
      port 1 <bits32> value "0x1"
      port 2 <bits32> value "0x2"
      port 3 <bits32> value "0x3"
      port 4 <bits32> value "0x4"
      port 5 <bits32> value "0x5"
    1 outputs: (nodes 12)
      port 0 <bits32>
;
  1 ND_LOOPINFO <"tst.sc", "main", 3>
    2 inputs:
      port 0 <bits32>
      port 1 <bits32>
    0 outputs:
    ;
  2 ND_G_INPUT  (input 5 for graph node 0) <"tst.sc", "main", 3>
    0 inputs:
    1 outputs:
      port 0 <bits32>  11.2  1.0  1.1
    ;
  3 ND_G_INPUT  (input 4 for graph node 0) <"tst.sc", "main", 3>
    0 inputs:
    1 outputs:
      port 0 <bits32>  11.1
    ;
  4 ND_G_INPUT  (input 0 for graph node 0) <"tst.sc", "main", 3>
    0 inputs:
    1 outputs:
      port 0 <bits32>  7.0
    ;
  5 ND_G_INPUT  (input 2 for graph node 0) <"tst.sc", "main", 3>
    0 inputs:
    1 outputs:
      port 0 <bits32>  7.4
    ;
  6 ND_G_INPUT  (input 1 for graph node 0) <"tst.sc", "main", 3>
    0 inputs:
    1 outputs:
      port 0 <bits32>  7.3
    ;
  7 ND_RC_WINDOW_GEN_1D <"tst.sc", "main", 3>
    6 inputs:
      port 0 <bits32>
      port 1 <bits3>  value "0x4"
      port 2 <bits1>  value "0x1"
      port 3 <bits32>
      port 4 <bits32>
      port 5 <bits1>  value "0x0"
```

Figure 11: Text representation of DFG.

```
  7 outputs:
    port 0 <bits12>  8.0
    port 1 <bits12>  8.2
    port 2 <bits12>  8.4
    port 3 <bits12>  8.6
    port 4 <bits32>
    port 5 <bits1>  11.4
    port 6 <bits1>  11.5
  ;
8 ND_REDUCE_USUM_MACRO <"tst.sc", "main", 4>
  8 inputs:
    port 0 <bits12>
    port 1 <bits1>  value "0x1"
    port 2 <bits12>
    port 3 <bits1>  value "0x1"
    port 4 <bits12>
    port 5 <bits1>  value "0x1"
    port 6 <bits12>
    port 7 <bits1>  value "0x1"
  1 outputs:
    port 0 <bits12>  10.1
  ;
9 ND_G_INPUT  (input 3 for graph node 0) <"tst.sc", "main", 4>
  0 inputs:
  1 outputs:
    port 0 <bits12>  10.0
  ;
10 ND_UADD <"tst.sc", "main", 4>
  2 inputs:
    port 0 <bits12>
    port 1 <bits12>
  1 outputs:
    port 0 <bits12>  11.0
  ;
11 ND_WRITE_TILE_1D_1D <"tst.sc", "main", 4>
  6 inputs:
    port 0 <bits12>
    port 1 <bits32>
    port 2 <bits32>
    port 3 <bits32>  value "0x1"
    port 4 <bits1>
    port 5 <bits1>
  1 outputs:
    port 0 <bits1>  12.0
  ;
12 ND_G_OUTPUT  (output 0 for graph node 0) <"tst.sc", "main", 3>
  1 inputs:
    port 0 <bits1>
  0 outputs:
  ;
```

Figure 12: Text representation of DFG (contd).