1 Data Structures

1.1 Basic Types

The following are the basic types, out of which others are built, and which will be referred to. There is generally, but not always, a direct relationship to a C++ primitive.

Name	Closest C++ Equivalent	Description
Integer	int	Whole number
Float	float	Floating point number
Queue	std::list	FIFO queue
List(type)	std::list⟨type⟩	
String	std::string	String object that provides operations to manipulate itself
File	std::iostream	Abstract type to represent simple I/O operations
Map(KeyType → ValueType, DEFAULT: DefaultValue)	std::unordered_map⟨KeyType, ValueType⟩	A map to translate values of type KeyType to values of type ValueType. If the key isn't present, returns DefaultValue

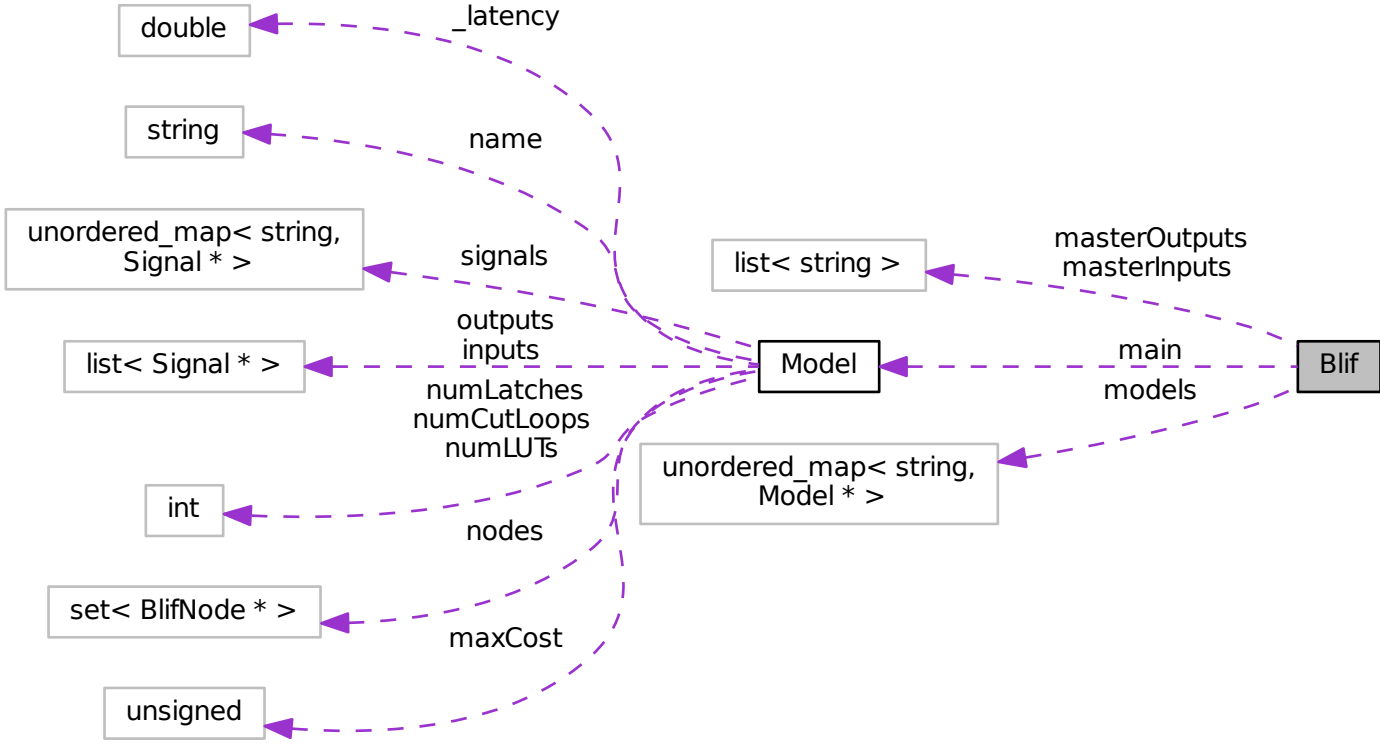
The following are complex types, which are further defined below. This is merely a quick description of each.

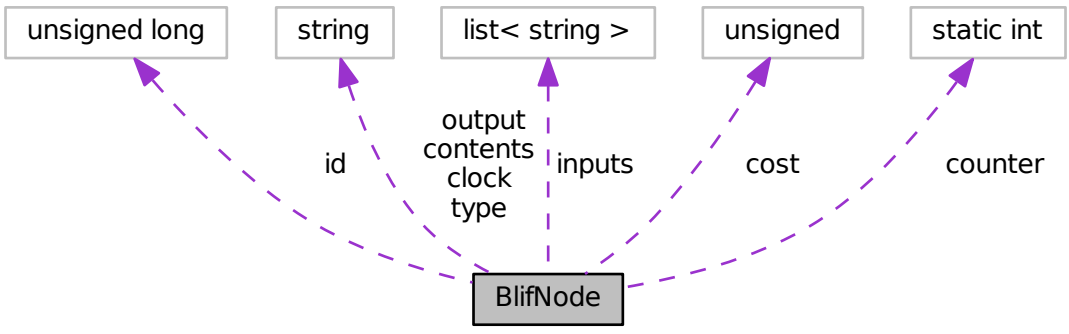
Name	Description
Blif	Parent object, contains all information about a BLIF file and provides useful operations
Model	Represents a circuit within a BLIF file, and provides methods to manipulate said circuit
BlifNode	A circuit element, or node in the DFG representing the circuit
Signal	A signal within a specific circuit, or Model, representing a set of edges with common source

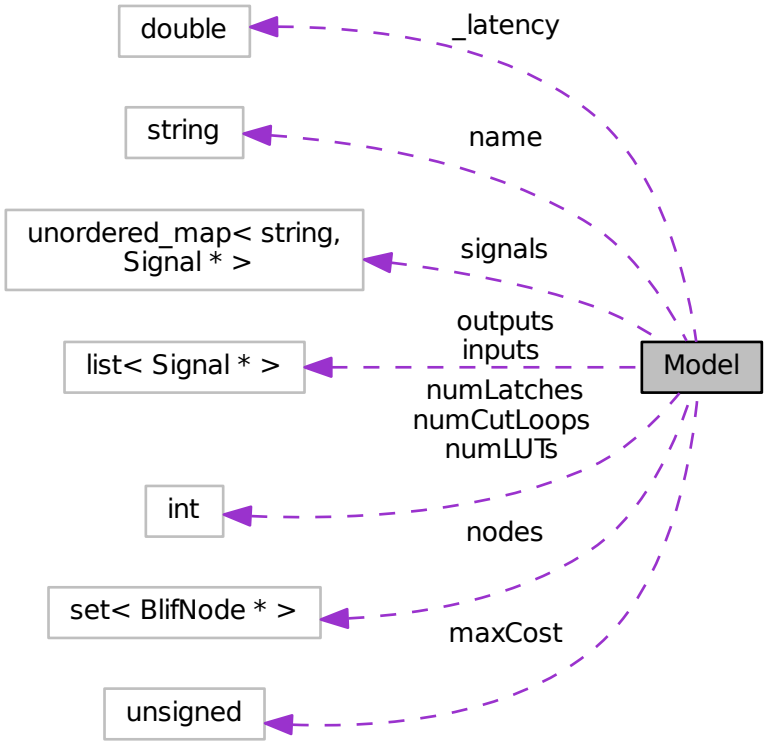
1.2 Blif

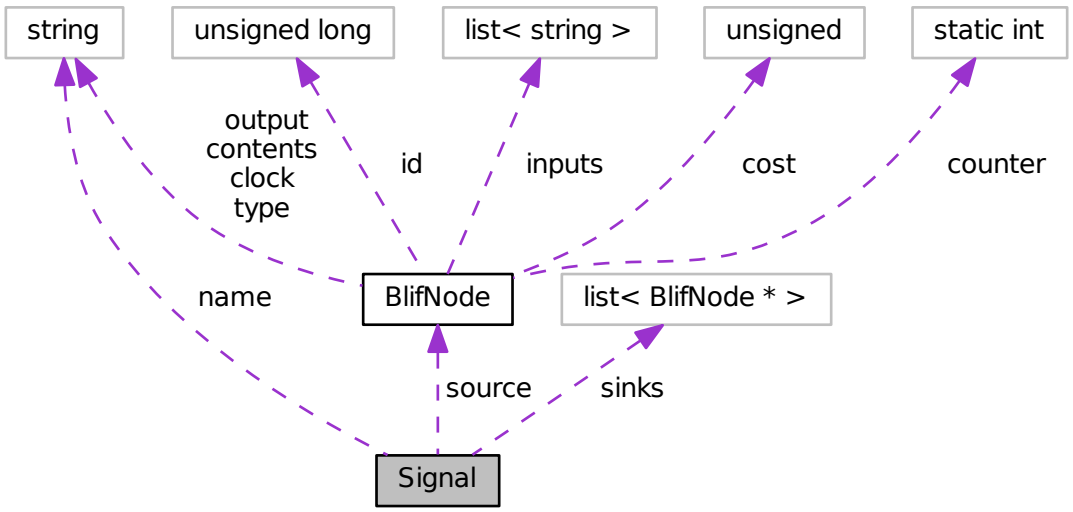
1.3 Model

Represents the circuit as a DFG. Contains a list of nodes, map of signal name → Signal*, and lists of primary inputs and outputs for the circuit. Each node contains the names of its input and output signals, allowing the Signal* to be looked up, then the Signal* contains pointers to its source and sink nodes. This allows the DFG to be traversed by going from node, to signal, to node, etc. A BlifNode represents the information in a circuit element declaration within a **BLIF!** (**BLIF!**) file, which includes only the name of its input and output signals. The actual Signal itself is a separate circuit specific construct designed to allow for ease of traversal of the circuit as a **DFG!** (**DFG!**). As such, we don't directly point to signals from a BlifNode, as the Signal depends on the circuit context. TODO: Image showing DFG traversal, and example of blif file and class contents !


double
_latency
string
name
unordered_map< string, Signal * >
signals
list< string >
masterOutputs
masterInputs
outputs
inputs
list< Signal * >
Model
main
Blif
numLatches
numCutLoops
numLUTs
models
unordered_map< string, Model * >
int
nodes
set< BlifNode * >
maxCost
unsigned


unsigned long
string
list< string >
unsigned
static int
output
contents
clock
type
id
inputs
cost
counter
BlifNode


double
_latency
string
name
unordered_map< string, Signal * >
signals
outputs
inputs
list< Signal * >
Model
numLatches
numCutLoops
numLUTs
int
nodes
set< BlifNode * >
maxCost
unsigned


string
unsigned long
list< string >
unsigned
static int
output
contents
clock
type
id
inputs
cost
counter
name
BlifNode
list< BlifNode * >
source
sinks
Signal

2 Algorithm

Types marked with an * are custom types defined previously in section 1.

2.1 Main

Partition, Triplicate, Join and Flatten are all implemented in separate programs. Main is responsible for taking an input file and running it through our toolchain to produce a TMR'd output file. We're given a

Algorithm 1 Main Algorithm

Variable	Type	Description
$input$	**File**	Input blif file
$targetRecoveryTime$	**float**	Per partition recovery time (in seconds)
$files$	**List(File)**	circuit partitions, one per file
$file$	**File**	
$header$	**string**	string containing the first three lines of the input file
$output$	**File**	output file

1: **procedure** MAIN($input$, $targetRecoveryTime$)
2: $files \leftarrow$ Partition($input$)
3: **for all** $file \in files$ **do**
4: $file \leftarrow$ Triplicate($file$)
5: **end for**
6: $header \leftarrow input.lines[0 \rightarrow 3]$
7: $file \leftarrow$ Join($files, header$)
8: $output \leftarrow$ Flatten($output$)
9: **end procedure**

blif file as input. In line 11 we partition the input circuit into a number of sub circuits, each in a separate file, as further expanded in Algorithm 2. Then in lines 12-13 for each partition file, we read it in as a black box, triplicate it, insert voting logic, and write it back out. Next in line 14 we extract the original header, which provides the name, inputs and outputs of the original circuit. We then, in line 15, join all the partitions together with the original name, inputs and outputs (in the same order), as the original circuit, and finally line 16 flattens the circuit, i.e. transforms the generated heirarchical netlist into a flat netlist with only one main model, or circuit, and no submodels.

Variable	Type	Description
$file$	**File**	input file
$targetRecoveryTime$	**float**	maximum per partition recovery time (in seconds)
$blif$	**Blif***	In-memory representation of input blif file
$circuit$	**Model***	Main circuit from input file, represented as DFG
$partition$	**Model***	Circuit, which we are adding nodes to, to make our partition
$queue$	**Queue**	FIFO queue of nodes to visit
$visited$	**Map(BlifNode*$\rightarrow$ bool)**	Map of whether a BlifNode is visited
$signal$	**Signal***	
$circuit.outputs$	**List(Signal*)**	List of output Signal* of a circuit
$signal.source$	**BlifNode***	Node which drives this Signal*
$queue.size$	**integer**	Number of nodes in queue
$node$	**BlifNode***	
$file$	**File**	
$files$	**List(File)**	
$numPartitions$	**int**	Counter of number of partitions
$signalName$	**string**	Name of a Signal*
$node.inputs$	**List(string)**	List of names of signals which are inputs to this node
$model.signals$	**Map(string $\rightarrow$ Signal*)**	Map from signal name to Signal* representing it in that Model*

Table 1: Variables for Partition

2.2 Partition

Given an input file, Partition reads it in, and splits it into a number of smaller subcircuits, each of which has a maximum recovery time of our target recovery time or less. Each subcircuit is then output to its own separate file, each of which is a valid **BLIF!** and circuit on its own.

Algorithm 2 Partition

```
1: procedure PARTITION(file)
2:     blif ← new Blif(file)                                   ▷ Read in file
3:     circuit ← blif.main                                     ▷ The actual circuit within the blif file
4:     partition ← new Model                                   ▷ Empty Circuit
5:     queue ← new Queue                                       ▷ Empty Queue
6:     visited ← new Map(BlifNode → bool, DEFAULT: false)
7:     for all signal ∈ circuit.outputs do
8:         queue.Enqueue(signal.source)
9:     end for
10:    while queue.size > 0 do
11:        node ← queue.Dequeue()
12:        if visited[node] = true then
13:            continue                                        ▷ Handle each node once and only once
14:        end if
15:        visited[node] ← true
16:        partition.AddNode(node)
17:        if partition.RecoveryTime() > targetRecoveryTime then
18:            partition.RemoveNode(node)
19:            MakeIOList(partition, circuit)
20:            file ← partition.WriteToFile()
21:            files ← files + file
22:            numPartitions ← numPartitions + 1
23:            partition ← new Model                           ▷ Empty Circuit
24:            partition.AddNode(node)
25:        end if
26:        for all signalName ∈ node.inputs do
27:            signal ← model.signals[signalName]
28:            queue.Enqueue(signal)
29:        end for
30:    end while
31:    if partition.size > 0 then
32:        MakeIOList(partition, circuit)
33:        file ← partition.WriteToFile()
34:        files ← files + file
35:    end if
36:    return files
37: end procedure
```

Line 2 reads a **BLIF!** into memory, transforming from the **BLIF!** format described in TODO: Reference to the one represented by TODO: Reference. Lines 12-15 ensure that we visit each node only once, and thus that each node is in exactly one partition, by checking if a node has been visited before and if so, skipping it, otherwise marking it as visited and coontinuing. Lines 16/24 inserts the current node into the open partition, cutting any created cycles and updating values such as critical path length as outlined in Algorithm 5. Line 17 tests if the current partition recovery time is greater than our specified limit, with the algorithm used to calculate the recovery time located at Algorithm 4. If the recovery time is greater we execute lines 18-24, where we remove the just added node to bring our recovery time back under the limit, and then write the partition to a file. Line 18 calculates which signals are primary inputs or outputs for the partition, and promotes them accordingly, with more detail given in Algorithm 3. Writing the partition to a file simply involves outputting the name, inputs, outputs, and a list of every node in the partition in **BLIF!** format. RemoveNode, on line 19, merely removes the node from the partition's list of nodes rather than fully reversing everything AddNode does. As WriteToFile simply serialises the inputs, outputs and node list this is all that's required with the caveat that some cycles may be cut unecessarily. Lines 31-34 write out the final partially full partition, if there is one. Again, WriteToFile simply outputs the circuit name, list of inputs, outputs and clocks, and list of nodes, with no further processing required.

2.3 MakeIOList

Given an original partition and a subpartition, promote any signals which are sourced or sunk outside of the subpartition to a primary input or output of the subpartition. We iterate through every signal in our

Algorithm 3 MakeIOList

Variable	Type	Description
$partition$	**Model***	Partition to create list of primary inputs and outputs for
$originalCircuit$	**Model***	Original model
$signal$	**Signal***	
$signal.source$	**BlifNode***	The driver for the signal
$partition.inputs$	**List(BlifNode*)**	List of primary inputs for the circuit
$partition.signals$	**Map(string $\rightarrow$ Signal*)**	Map from signal name to Signal*
$originalCircuit.signals$	**Map(string $\rightarrow$ Signal*)**	Map from signal name to Signal*
$signal.sinks$	**List(BlifNode*)**	List of sinks for the signal

```
1:  procedure MAKEIOLIST(partition, originalCircuit)
2:      for all signal ∈ partition.signals do
3:          if signal.source = NULL then                ▷ If this signal has no driver
4:              partition.inputs.Add(signal)
5:          end if
6:          otherSignal ← originalCircuit.signals[signal.name]   ▷ Get the corresponding signal in the original circuit
7:          if otherSignal.sinks − signal.sinks > 0 then ▷ If the signal has more sinks in the original circuit than it does in this partition
8:              partition.outputs.Add(signal)
9:          end if
10:     end for
11: end procedure
```

partition. For each one we check if we have a source (line 4), if not it must be a primary input. Similarly, on line 7 we check if we have a sink which is not represented within our partition. If so, promote it to a primary output of the partition.

2.4 RecoveryTime

For a given partition, calculate its error recovery time. The derivation of this algorithm and the values used

Algorithm 4 RecoveryTime

Variable	Type	Description
$latency$	**float**	Circuit latency (i.e. time for input to completely propagate to output) in seconds
$clockFrequency$	**Integer**	Operating frequency of the circuit, in seconds
$criticalPath$	**Integer**	Maximum number of steps between an input and an output
$numFF$	**Integer**	Number of Latches in circuit
$numLUT$	**Integer**	Number of look up tables in circuit
$resynchronisationTime$	**Float**	Time, in seconds, that it takes to resynchronise circuit
$detectionTime$	**Float**	Time, in seconds, that it takes to detect an error
$ReconfigureTime$	**Float**	Time, in seconds, that it takes to reconfigure circuit
$communicationTime$	**Float**	Time, in seconds, that it takes to transmit reconfiguration request to controller

1: **procedure** RECOVERYTIME($partition$)
2: $latency \leftarrow frequency \times (criticalpath + 1)$
3: $detectionTime \leftarrow latency$
4: $resynchronisationTime \leftarrow latency$
5: $reconfigurationTime \leftarrow \max(numFF, numLUT)/10/15...morestuff$
6: $communicationTime \leftarrow numPartitions \times latency \times morestuff$
7: $recoveryTime \leftarrow detectionTime + resynchronisationTime + reconfigurationTime + communicationTime$
8: return $recoveryTime$
9: **end procedure**

is fully discussed in Section TODO: Reference. The criticalpath is a measure of the maximum number of latches on a path from input to output. The +1 is to account for the contribution of combinational logic, which may be up to one additional clock cycle of latency.

2.5 AddNode

Insert a node into an existing partition, or circuit, while updating appropriate parameters (i.e. maximum path length and signals) which are depended upon by other components (i.e. recovery time calculation and **DFG!** traversal respectively). Additionally, detect any newly created cycles and cut them. This ensures that the circuit is always (except within this method) an acyclic graph with every node reachable. Lines 4-6 check if one of the inputs signals for this node has been cut to remove a cycle. If so, we rename the signal appropriately, to what the new primary input is called. Additionally, lines 3-11 update the appropriate signals so that this node is reachable within the **DFG!**. Lines 12-19 then update the maximum path length (or latency in clock cycles) while detecting and cutting any newly created cycles.

Algorithm 5 AddNode

Variable	Type	Description
$partition$	**Model***	Model* containing DFG representing partition to add node to
$node$	**BlifNode***	Node to add
$signal$	**Signal***	
$signalName$	**string**	
$partition.cutLoops$	**Map[string → string]**	For signals which have been cut, a map of the old to the new name
$partition.signals$	**Map[string → Signal***	Map of signal name to Signal*
$signal.sinks$	**List(BlifNode*)**	List of sinks for a Signal*
$signal.source$	**BlifNode***	Source, or driver, for a Signal*
$inCost$	**int**	Maximum number of critical path steps to reach node, not counting the node itself
$explored$	**Map(BlifNode* → boolean)**	Whether a node has been reached yet in the current iteration

1: **procedure** ADDNODE($partition, node$)
2: $nodes.insert(node)$
3: **for all** $signalNameinnode.inputs$ **do**
4: **if** $partition.cutLoops[signalName] \neq$ "" **then**
5: $signalName \leftarrow cutLoops[signalName]$ ▷ If this signal has been renamed already to avoid a cycle, rename this occurance of it.
6: **end if**
7: $signal \leftarrow partition.signals[name]$
8: $signal.sinks.Add(node)$
9: **end for**
10: $signal \leftarrow partition.signals[node.output]$
11: $signal.source \leftarrow node$
12: $inCost \leftarrow 0$
13: **for all** $signalName \in node.inputs$ **do**
14: $source \leftarrow partition.signals[signalName].source$
15: **if** $partition.costs[source] > inCost$ **then**
16: $inCost \leftarrow partition.costs[source]$
17: **end if**
18: **end for**
19: UpdateCostsAndBreakCycles($partition, node, NULL, node, inCost, explored, costs$)
20: **end procedure**

2.6 UpdateCostsAndBreakCycles

Recursively traverse our network to update maximum path lengths to account for our new node and additional paths. While traversing the network, detect and break any cycles we encounter. This turns a possibly cyclic **DFG!** with partially computed path lengths, into an acyclic **DFG!** with fully computed path lengths.

We care about two things. One, the maximum cost to reach a node, and two, detecting and removing any cycles. Given an existing **DAG!** (**DAG!**) Directed Acyclic Graph which we insert a new node into, then 1. The new node is the root node of a subtree within the **DAG!**. 2. Nodes which are not within the subtree cannot have the maximum cost to reach them change (as nothing has changed in any path to them). 3. Any cycles must pass through the new root node, as there are no cycles elsewhere in the graph. 4. Without any cycles, the root node will only be reached once at the start. Using this information we develop our traversal algorithm. Line 2 demonstrates an optimisation, in that once a path has been checked we need not recheck it unless we have found a more expensive path to it as otherwise nothing will change. Lines 5-8 check if we have detected a cycle. If so, cut it through cutting the signal, detailed in algorithm 7.

Algorithm 6 UpdateCostsAndBreakCycles

Variable	Type	Description
$partition$	**Model***	Model* containing DFG representing partition to add node to
$root$	**BlifNode***	Newly added node
$node$	**BlifNode***	Node we just came from
$node$	**BlifNode***	Current node
$costToReach$	**int**	Maximum number of critical path steps to reach node, not counting the node itself
$explored$	**Map(BlifNode* → boolean)**	Whether a node has been reached yet in the current iteration
$partition.signals$	**Map[string → Signal***	Map of signal name to Signal*
$partition.signals[parent.output]$	**BlifNode***	Signal driven by the prior node i.e. the graph edge we most recently travelled.
$node.cost$	**int**	1 for latches, 0 for LUTs
$costs$	**Map(BlifNode* → int)**	Map of the cost to reach each node
$node$	**BlifNode***	
$signal.sinks$	**List(BlifNode*)**	List of sinks for a Signal*
$cost$	**int**	Number of critical path steps to reach node, including the node itself

1: **procedure** UPDATECOSTSANDBREAKCYCLES($partition, root, parent, node, costToReach, explored$)
2: **if** $explored[node] = true$ and $costs[node] \geq costToReach$ **then** ▷ Already expanded this path, and it can't change
3: return
4: **end if**
5: **if** $parent \neq NULL$ and $node = root$ **then** ▷ We have a cycle, as all newly created cycles must go through the new node, and the new node should only ever be reached once at the start without cycles
6: CutSignal($partition, partition.signals[parent.output]$)
7: return
8: **end if**
9: $cost \leftarrow costToReach + node.cost$
10: **if** $cost > costs[node]$ **then**
11: $costs[node] = cost$
12: **else**
13: $cost = costs[node]$
14: **end if**
15: **for all** $node \in partition.signals[node.output].sinks$ **do**
16: UpdateCostsAndBreakCycles($root, node, child, cost, explored$)
17: **end for**
18: $explored[node] = true$
19: **end procedure**

2.7 CutSignal

Given a signal, cut it, by splitting it into two signals. One of which is a primary input with the sinks of the previous signal. One of which is a primary output with the source of the previous signal.

Algorithm 7 CutSignal

Variable	Type	Description
$partition$	**Model***	Model* containing DFG representing partition to add node to
$partition.cutLoops$	**Map(string → string)**	Map from old to new name of a cut signal. Note that only the input signal is renamed. The output signal retains the same name.
$signal$	**Signal***	The signal we're cutting
$newSignal$	**Signal***	The new signal we created
$signal.source$	**BlifNode***	The source, or driver, of a signal
$partition.signals$	**Map(string → Signal*)**	A map from signal name to signal
$partition.outputs$	**List(Signal*)**	The primary outputs for the circuit
$node$	**BlifNode***	
$node.inputs$	**List(string)**	names of input signals to a node

1: **procedure** CUTSIGNAL($partition, signal$)
2: $newSignal \leftarrow newSignal(name)$ ▷ Create a new signal with the same name, but currently no sources or sinks
3: $newSignal.source \leftarrow signal.source$
4: $partition.signals[name] \leftarrow newSignal$
5: replace($partition.outputs, signal, newSignal$) ▷ Replace any occurances of the old signal in the circuit outputs with the new
6: $signal.name \leftarrow$ "$qqrin$" $+ name$
7: $signal.source = NULL$
8: $signals[signal.name] \leftarrow signal$
9: $cutLoops[newSignal.name] \leftarrow signal.name$
10: **for all** $node \in signal.sinks$ **do**
11: replace($node.inputs, name,$ "$qqrin$" $+ name$)
12: **end for**
13: **end procedure**

Cutting a signal consists of splitting an existing signal into two. One with the same source but no sinks as a primary output. One with the same sinks but no source as a primary input. We also need to update existing references to point to the correct signal, and record the change to update any future references as nodes get added. To that end, lines 3-5 create a new signal with the same source, no sinks, and replace all existing instances with the new signal. We now in lines 6-12 rename the previous signal to a reserved name, set it to have no source, and update all its sinks to refer to the correct, renamed signal.

2.8 TMR

Given a file containing a partition, read it in as a black box, triplicate it, add voter logic, write back out to file.

Algorithm 8 TMR

Variable	Type	Description
$file$	**File**	File to triplicate
$circuit$	**string**	Contents of the circuit file
$header$	**string**	String containing the circuit name, inputs and outputs as they appear in the **BLIF!**
$voter$	**string**	Contents of the voter circuit file
$subcktDefintionN$	**string**	String representing a subcircuit in **BLIF!** format
$voterDefintionN$	**string**	String representing a subcircuit in **BLIF!** format

1: **procedure** TMR($file$)
2: $circuit \leftarrow file.read()$
3: $header \leftarrow circuit.header$
4: $voter \leftarrow voter.read()$
5: $subcktDefinition1 \leftarrow MakeSubcktDefinition(circuit)$
6: $subcktDefinition2 \leftarrow MakeSubcktDefinition(circuit)$
7: $subcktDefinition3 \leftarrow MakeSubcktDefinition(circuit)$
8: $voterDefinition \leftarrow MakeSubcktDefinition(voter)$
9: $subcktDefinition.ConnectInputs(circuit.inputs)$
10: $voterDefinition.ConnectInputs(subcktDefinition1.outputs, subcktDefinition2.outputs, subcktDef$
▷ Connect the outputs of the partitions to the inputs of the voter
11: $voterDefinition.ConnectOutputs(circuit.outputs)$
12: $file.write(header)$
13: $file.write(voterDefinition)$
14: $file.write(subcktDefinition1)$
15: $file.write(subcktDefinition2)$
16: $file.write(subcktDefinition3)$
17: $file.write(voter)$
18: $file.write(circuit)$
19: **end procedure**

This method operates on the **BLIF!** in a low level way, dealing manipulating the actual file contents, rather than operating on an abstract circuit representation, as we transform a flat circuit, into a heirarchical circuit, in which out original flat circuit remains untouched but we insert voting and similar logic around it. We read in our partition circuit and voter circuit. We now create three partition subcircuit and one voter subcircuit definitions. We match up our signal names between them appropriately, and then write out our subcircuit definitions, followed by our partition and voter subcircuits.

This transforms a file from format:

```
1 .name partition
2 .inputs ...
3 contents
```

Into one in format:

```
.name TMR
.inputs ...
.subckt partition
.subckt partition
.subckt partition
.subckt voter
.end

.name voter
...
.end
.name partition
...
.end
```

2.9 BlifJoin

Given a list of blif files, concatenates them all together, creates subcircuit definitions to connect them all together, and writes them to a file

This transforms a set of files in format:

```
.name partitionN
.inputs ...
contents
```

Into one file with format:

```
.name TMR
.inputs ...
.subckt partition1
.subckt partition2
.subckt partition3
...
.end

.name partition1
...
.end
.name partition2
...
.end
...
```

2.10 Flatten

Given a heirarchical blif file, run it through abc to flatten it, and postprocess if necessary.

Algorithm 9 Flatten

Variable	Type	Description
$file$	**File**	File to flatten

1: **procedure** FLATTEN($file$) ▷ Flattening is currently performed by abc (link), called with parameters:
2: ./abc -o output -c echo file ▷ Due to bug in abc, clock information is stripped from latches, so we then call grep and sed to fix the output file
3: $latch$ ←split(grep -m 1 'latch' file)
4: **if** $latch$ **then**
5: sed -ri 's/(\.latch.+)(2)/\1 '+latch[3] + ' ' + latch[4] + ' 2/' output
6: **end if**
7: **end procedure**

./abc is provided an input file, given the command to echo the current file, and told to output everything to output. grep is called to search for latches, and return the latch information if there is one. If there is, replace the faulty latch information with the correct information. This assumes that there is only one global clock, all latches are triggered on the same signal (e.g. rising edge, falling etc), and all latches have initial state don't care, which holds true for all provided benchmarks.