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
BlifModel	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 BlifModel, representing a set of edges with common source

```
Signal
{
    string name;
    BlifNode* source;
    list<BlifNode*> sinks;
}
```

2 Algorithm

Types marked with an * are defined in the previous section (Data Structures). We're given a blif file as input. In line 11 we partition the input circuit into a number of sub circuits, each in a separate file. Then in lines 12-13 we iterate over all the partitions, and transform them into a triplicated partition with three copies and a voter circuit. Then 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.

Algorithm 2 Main Algorithm

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

varMain

main

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**

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
$circuit$	BlifModel*	Main circuit from input file, represented as DFG
$partition$	BlifModel*	Circuit, which we are adding nodes to, to make our partition
$queue$	Queue	FIFO queue of nodes to visit
$visited$	Map(BlifNode*→ bool)	Map of whether a BlifNode is visited
$signal$	Signal*	
$circuit.outputs$	List of 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 of file	
$numPartitions$	int	Counter of number of partitions
$signalName$	string	Name of a Signal*
$node.inputs$	List of string	List of names of signals which are inputs to this node
$model.signals$	Map(string → Signal*)	Map from signal name to Signal* representing it in that BlifModel*

varPart

Table 1: Variables for Partition

main

Algorithm 3 Partition

```
procedure PARTITION(file)
    blif ← new Blif(file)                                          ▷ Read in file
    circuit ← blif.main                        ▷ The actual circuit within the blif file
    partition ← new BlifModel                                      ▷ Empty Circuit
    queue ← new Queue                                              ▷ Empty Queue
    visited ← new Map(BlifNode → bool, DEFAULT: false)
    for all signal ∈ circuit.outputs do
        queue.Enqueue(signal.source)
    end for
    while queue.size > 0 do
        node ← queue.Dequeue()
        if visited[node] = true then
            continue                           ▷ Handle each node once and only once
        end if
        visited[node] ← true
        partition.AddNode(node)
        if partition.RecoveryTime() > targetRecoveryTime then
            partition.RemoveNode(node)
            CutLoops(partition)
            file ← partition.WriteToFile()
            files ← files + file
            numPartitions ← numPartitions + 1
            partition ← new BlifModel                              ▷ Empty Circuit
        end if
        for all signalName ∈ node.inputs do
            signal ← model.signals[signalName]
            queue.Enqueue(signal)
        end for
    end while
    if partition.size > 0 then
        CutLoops(partition)
        file ← partition.WriteToFile()
        files ← files + file
    end if
    return files
end procedure
```

Lines 2-6 are setting up our variables with initial values. We read a blif file in to memory, where it is represented as a DFG with a number of properties as described in Reference. In line 3, circuit refers to the main circuit of a blif file. As we only support non-heirarchical blif files, this will always be the only circuit. In lines 7-8 we push our outputs onto the queue, to start traversing. Line 11 pops the node from the front of the queue. Next, in lines 12-15 we check if this node is already marked as visited. If so, we skip it as we only add each node to exactly one partition. Otherwise, we mark it as visited and proceed to partition it. In lines 16-17 we add the node to the current partition, and test if we're still within our recovery time. If not, then in lines 18-20 we remove the current node from the partition, cut cycles within the partition, and write the partition out to a file. One file per partition. Then in 21-22 we update our collection of output files and increment our counter for the number of partitions, and finally, in line 23-24 we create a new empty circuit for our next partition, and add the node to it. Then, we add the inputs to this node to our queue, and continue traversing and partitioning until we've reached every node. Lastly, in lines 31-35 we check if our current partition has anything in it. If so, cut loops and write it out.

Algorithm 5 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

varPart
main

procedure RECOVERYTIME($partition$)
    $latency \leftarrow frequency \times criticalpath$
    $detectionTime \leftarrow latency$
    $resynchronisationTime \leftarrow latency$
    $reconfigurationTime \leftarrow \max(numFF, numLUT)/10/15...morestuff$
    $communicationTime \leftarrow numPartitions \times latency \times morestuff$
    $recoveryTime \leftarrow detectionTime + resynchronisationTime + reconfigurationTime + communicationTime$
    return $recoveryTime$
end procedure

Algorithm 7 CutLoops

Variable	Type	Description
$partition$	BlifModel*	BlifModel* containing DFG representing partition to cut cycles in
$state$	Map(BlifNode* $\rightarrow$ int)	Map of whether a node is UNKNOWN, EXPLORING, or FINISHED
$signal$	Signal*	
$partition.outputs$	List of Signal*	List of Signal* representing primary outputs of circuit

varPart

main

1: **procedure** CUTLOOPS($partition$)
2: $\quad state \leftarrow Map(BlifNode* \rightarrow int, DEFAULT : 0)$
3: $\quad$ **for all** $signal \in partition.outputs$ **do**
4: $\quad\quad$ CutLoopsRecursive$(state, NULL, signal)$
5: $\quad$ **end for**
6: **end procedure**

Start recursing from outputs back to detect loops. Line 3 starts the recursive traversal for each output, with no parent.

Algorithm 9 CutLoopsRecursive

Variable	Type	Description
$partition$	BlifModel*	BlifModel* containing DFG representing partition to cut cycles in
$state$	Map(BlifNode* $\rightarrow$ int)	Map of whether a node is UNKNOWN, EXPLORING, or FINISHED
$parent$	BlifNode*	Parent node
$signal$	Signal*	
$node$	BlifNode*	
$signal.name$	string	Name of a signal, as it appears in the blif file
$node$	BlifNode*	
$node.inputs$	List(string)	A list of names of input signals
$signalName$	string	Name of a signal, as it appears in the blif file
$partition.signals$	Map(string $\rightarrow$ Signal)*	Map from signal name to signal with that name in that partition
$signal$	Signal*	

varPart
main

procedure CutLoopsRecursive($partition, state, parent, signal$)
    $node \leftarrow signal.source$
    if $state[node] = EXPLORING$ **then** ▷ Found a cycle
        ReplaceSignalName($parent.inputs, signal.name, "qqrin" + signal.name$)
    else if $state[node] = FINISHED$ **then** ▷ Already explored this path
        return
    else
        $state[node] = EXPLORING$
        for all $signalName \in node.inputs$ **do**
            $signal \leftarrow partition.signals[signalName]$
            CutLoopsRecursive($partition, state, node, signal$)
        end for
    end if
end procedure