UNIVERSITAT POLITÈCNICA DE CATALUNYA
DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS
MÀSTER EN COMPUTACIÓ

TESI DE MÀSTER

LOGIC SYNTHESIS OF HANDSHAKE COMPONENTS USING CLUSTERING TECHNIQUES

ESTUDIANT: FRANCISCO DE LA CRUZ FERNÁNDEZ
DIRECTOR: JOSEP CARMONA VARGAS

DATA: 25 DE JUNY DE 2007

Acknowledgments

I would like to thank my director Josep Carmona for answering all my questions, helping me with the writing and always being available.

I would also like to thank Jordi Cortadella for introducing me in research two years ago and encouraging me to do the Ph.D. in Software.

Contents

1 Introduction **7**

2 Basic Theory **9**
- 2.1 Asynchronous Circuits 9
- 2.2 Petri Nets 9
  - 2.2.1 Signal Transition Graphs 11
  - 2.2.2 State Graphs 12
  - 2.2.3 SI Implementability Conditions 13
- 2.3 Synthesis of SI Circuits from STGs 14
- 2.4 Handshake Circuits 16
  - 2.4.1 Handshake Protocol 17
  - 2.4.2 Handshake Components 17

3 State of the Art and Problem Definition **19**
- 3.1 Behavioral Tools 19
  - 3.1.1 Petrify 20
- 3.2 Hardware Description Languages 21
  - 3.2.1 Balsa 23
- 3.3 Structural Tools 23
  - 3.3.1 Moebius 25

4 Methodology and Strategies **27**
- 4.1 A new design flow for asynchronous circuits 27
- 4.2 breeze2stg and eqn2abs 28
- 4.3 Example of two synchronized buffers 34

5 Experimental Results and Evaluation **43**

6 Discussion and Conclusions **45**

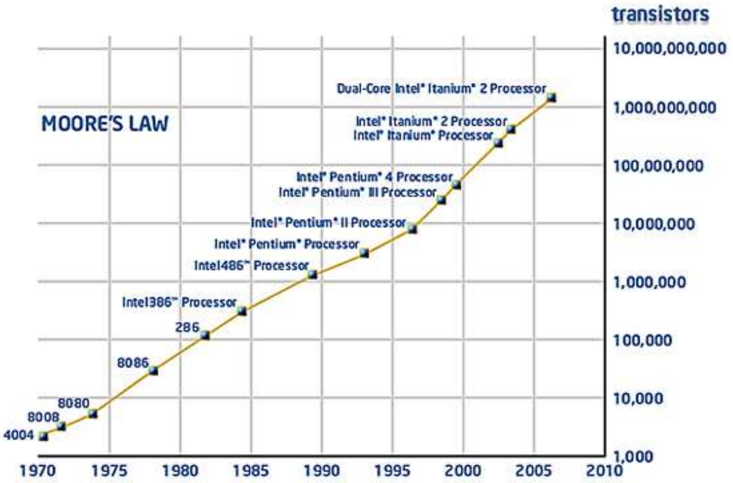
A Description of HCs behavior using STGs **47**
- A.1 Synch 49
- A.2 SequenceOptimised 50
- A.3 Concur 51
- A.4 DecisionWait 52
- A.5 Fork 55
- A.6 Call 56

B HC Nets and STGs of Balsa Examples 61
B.1 Arbiter Tree 62
B.2 Population Counter 66
B.3 Shifter 76
B.4 SSEM 80
B.5 Stack 88

Chapter 1

Introduction

The well known Moore's law predicts that the number of transistors on an integrated circuit doubles about every two years. In the near future, it will be possible to integrate all components of an electronic system into a single integrated circuit.

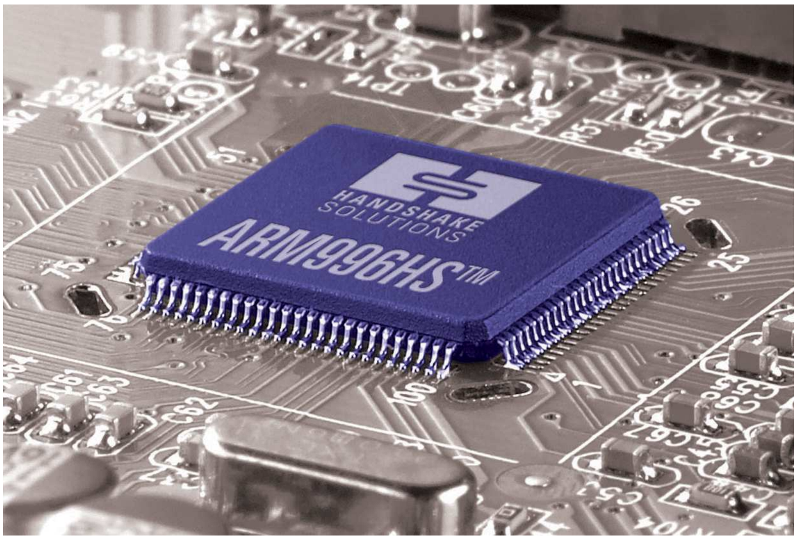


At present, synchronous circuits, which are synchronized by global clock signals, dominate the electronic industry. There are many design tools and much experience in this area.

However, the design cost of integrating all system components in an synchronous circuits is increasing. These components could have been designed for different clock periods. Therefore, a global clock period must be found to synchronize them. This search can be a very hard problem to solve.

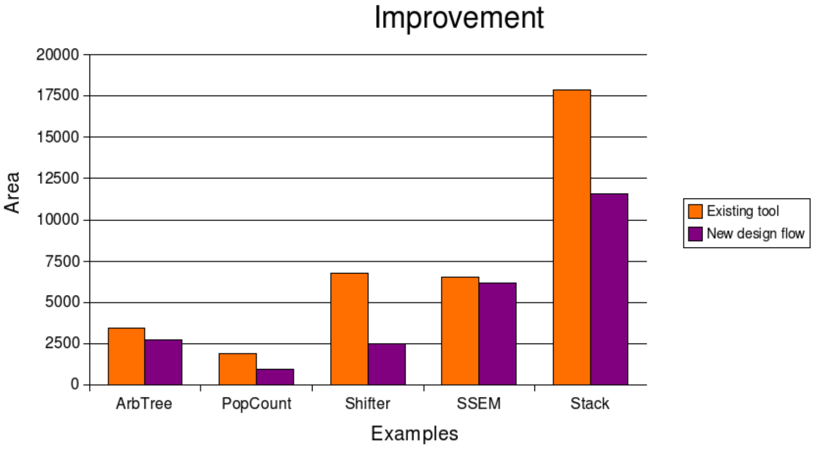
In contrast, asynchronous circuits, which have not global clock signals, are free from global synchronizations. Therefore, the power consumption is alleviated among other factors.

Asynchronous circuits are starting to be accepted by the electronic industry and companies are releasing their design tools. Handshake Solutions, a line of business of Philips, is the first company in commercially exploit them.



There are design tools to easily specify asynchronous circuits but their implementations can be improved. Also there are design tools to efficiently implement asynchronous circuits but they are difficult to design. However, in the last years, there have appeared new design flows to design asynchronous circuits.

In this document, a new design flow is proposed, which combines previous tools to easily specify and efficiently implement asynchronous circuits. Also two necessary applications are presented, which have been implemented to perform it. Some experimental results are showed, which proves the improvement in area on several examples.



The document is structured as follows: chapter 2 introduces some basic theory; chapter 3 describes the existing design tools; chapter 4 presents the new design flow; finally chapter 5 shows the experimental results.

Chapter 2

Basic Theory

This chapter presents some basic theory necessary to understand the following chapters. Section 2.1 describes asynchronous circuits and speed-independent circuits. Section 2.2 describes Petri nets, signal transition graphs, state graphs and implementability conditions. Section 2.3 describes the synthesis of speed-independent circuits from signal transition graphs. Section 2.4 describes handshake circuits, the handshake protocol and handshake components.

2.1 Asynchronous Circuits

Synchronous circuits are sequenced by one or more globally distributed periodic timing signals. In contrast, *asynchronous circuits* may be sequenced by other options than global periodic clock signals.

Asynchronous circuits can be classified into:

- *Delay-insensitive (DI)* circuits, which operate correctly regardless of the delays on their gates and wires.
- *Speed-independent (SI)* circuits, which operate correctly regardless of the delays on their gates. Their wires are assumed to have negligible delay.
- *Self-timed* circuits, which contain groups of self-timed elements. Each element is contained in an equipotential region, where wires have negligible or well-bounded delay. An element may be an SI circuit or a circuit whose correct operation relies on use of local timing assumptions. No timing assumptions are made on the communication between regions.

2.2 Petri Nets

Petri Nets (PN) are a graphical and mathematical model for describing and studying concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic systems.

A PN is a 4-tuple, $N = (P, T, F, m_0)$ where:

- P is a finite set of places,
- T is a finite set of transitions,

- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation,
- $m_0 \in \mathbb{N}^{|P|}$ is the initial marking.

A PN is represented as a directed and bipartite graph consisting of two kinds of nodes, places and transitions, where arcs are from a place to a transition or from a transition to a place. Places are drawn as circles and transitions as bars or boxes.

A marking assigns to each place a nonnegative integer. If a marking assigns to place p a nonnegative integer k, we say that p is marked with k tokens and we place k dots in place p. A marking, denoted by m, is a $|P|$-vector where the pth component, denoted by $m(p)$, is the number of tokens in place p.

For example, figure 2.1(a) shows a PN with three places $p1$, $p2$ and $p3$ and one transition t. The places $p1$ and $p2$ are input places and the place $p3$ is an output place of the transition t. The initial marking assigns one token to the places $p1$ and $p2$.

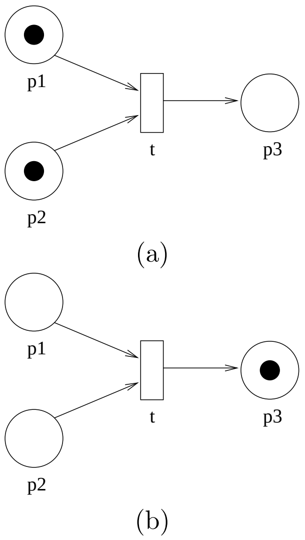


Figure 2.1: A marking (a) before and (b) after a transition firing

A marking in a PN is changed according to the firing rule:

1. A transition t is enabled if each input place p of t is marked.
2. An enabled transition may or may not fire.
3. A firing of an enabled transition t removes one token from each input place p of t, and adds one token to each output place p of t.

For example, figure 2.1(a) shows the marking before firing the enabled transition t, which is enabled because the input places $p1$ and $p2$ are marked. Figure 2.1(b) shows the marking after firing t, where the tokens of the places $p1$ and $p2$ have been removed and the token of the place $p3$ has been added.

A marking m_n is reachable from a marking m_0 if there is a sequence of firings $\sigma = t_1 t_2 \dots t_n$ that transforms m_0 to m_n, denoted by $m_0[\sigma\rangle m_n$. The set of all possible markings reachable from m_0 is denoted by $[m_0\rangle$. The reachability graph can be obtained considering the set of reachable markings as the set of states and the transitions among these markings as the transitions between the states.

2.2.1 Signal Transition Graphs

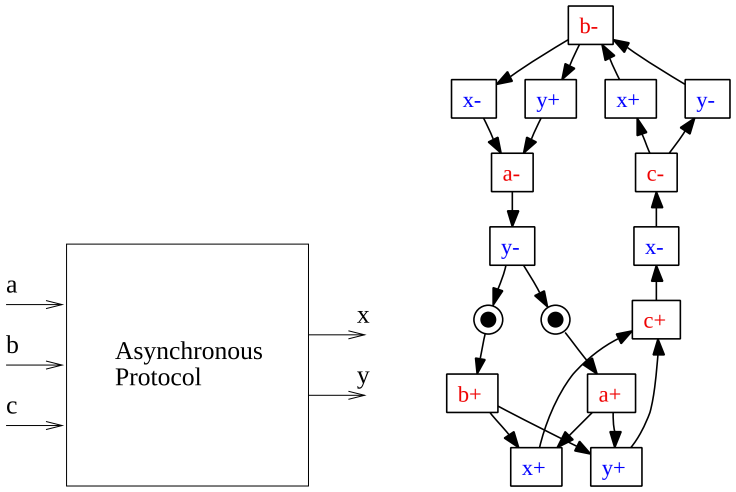
Transitions in a PN can represent signal changes of an asynchronous circuit.

A *Signal Transition Graph (STG)* is a triple (N, Σ, Λ) where:

- $N = (P, T, F, m_0)$ is a PN,
- Σ is the set of signals,
- $\Lambda : T \rightarrow \Sigma \times \{+, -\}$ is the labeling function which maps rising and falling signal transitions to transitions in the PN.

Signals in a STG are splitted into: inputs, outputs, internals and dummies. Input and output signals are in the interface of an asynchronous circuits. Output and internal signals must be synthesized. And dummy signals do not represent any event in the behaviour of an asynchronous circuits.

For example, figure 2.2(a) shows the interface of an asynchronous controller with three input signals a, b and c and two output signals x and y. Figure 2.2(b) shows the STG specifying its behavior, where implicit places (with one input transition and one output transition) are not drawn for simplicity.



(a) (b)

Figure 2.2: An asynchronous controller (a) interface and (b) STG

The behavior described in the STG is the following:

At the beginning, transitions a+ and b+ are enabled because their input places have a token. After their concurrent firing, x+ and y+ are enabled. When they fire concurrently, sequence c+ x- c- can fire. x+ and y- are enabled after firing c-. b- can fire when x+ and y- fire concurrently. After its firing, x- and y+ are *enabled. When they fire concurrently, sequence a- y- can fire. The marking is the same that at the beginning when y- fires.*

2.2.2 State Graphs

The *State Graph (SG)* is the encoded reachability graph of a STG.

A SG is a 5-tuple $A = (S, \Sigma, T, s_{in}, \lambda)$ where:

- S is the set of states,
- Σ is the set of signals,
- $T \subseteq S \times \Sigma \times \{+, -\} \times S$ is the labeled transition relation among source states, rising and falling signal transitions and destination states,
- s_{in} is the initial state,
- $\lambda : S \to \mathbb{B}^{|\Sigma|}$ is the encoding function which maps signal values to states in the SG.

$\lambda_x(s)$ is the value of signal x in state s.

For example, figure 2.3 shows the SG of the previous asynchronous controller, where the initial state is drawn as a double circle and the order of signal values in the encoding function is *abcxy*.

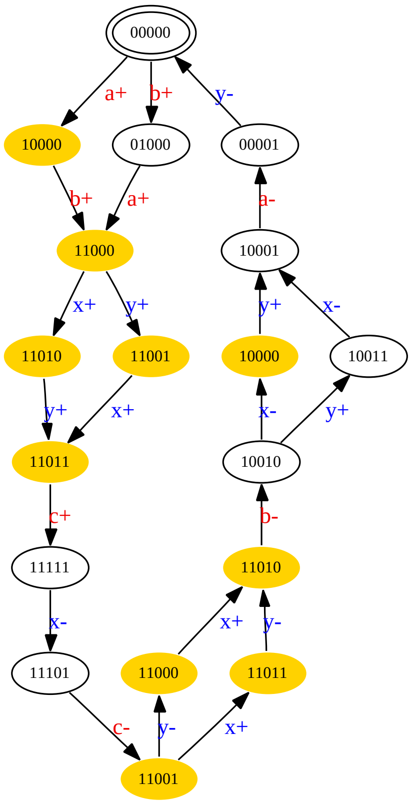


Figure 2.3: The previous asynchronous controller SG

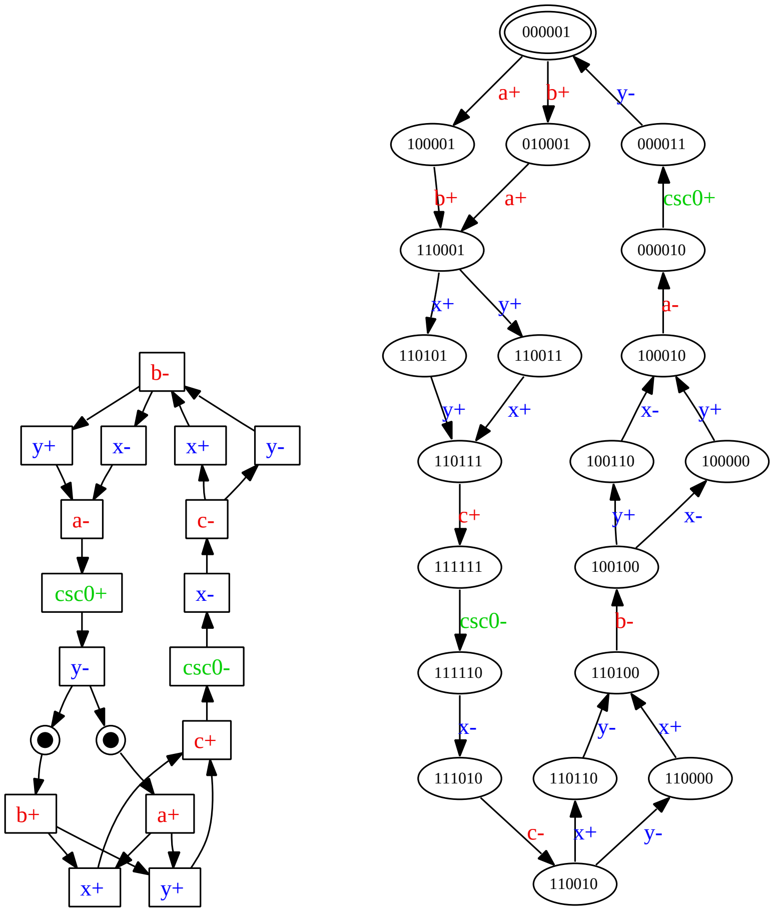
2.2.3 SI Implementability Conditions

An asynchronous circuit specification must satisfy the following properties to be implementable as a SI circuit [6]:

- *Boundedness*: the set of states in the SG must be finite.
- *Consistency*: the rising and falling transitions of each signal alternate in any trace.
- *Complete State Coding (CSC)*: there are not two different states with the same signal encoding and different behavior for an output or internal signal.
- *Persistency*: no signal transition can be disabled by another signal transition, unless both signals are inputs.

For example, the STG of the previous asynchronous controller, showed figure 2.2(b), is not implementable because it does not satisfy the CSC property. In its SG, showed in figure 2.3, there are different states with the same signal encoding and different behavior for an output or internal signal, which are indicated with filled circles.

Some logic synthesis tools enforce the CSC property by inserting internal signals to force a different encoding on this pair of states. For example, figure 2.4(a) shows the STG of the previous asynchronous controller with a new internal signal *csc*0. Figure 2.4(b) shows its SG now without CSC conflicts.



(a) (b)

Figure 2.4: The previous asynchronous controller (a) STG and (b) SG without CSC conflicts

2.3 Synthesis of SI Circuits from STGs

If a specification of an asynchronous circuit satisfy the implementability conditions, a logic equation can be automatically derived for each output and internal signal x.

The states of the SG can be classified for signal x into four regions:

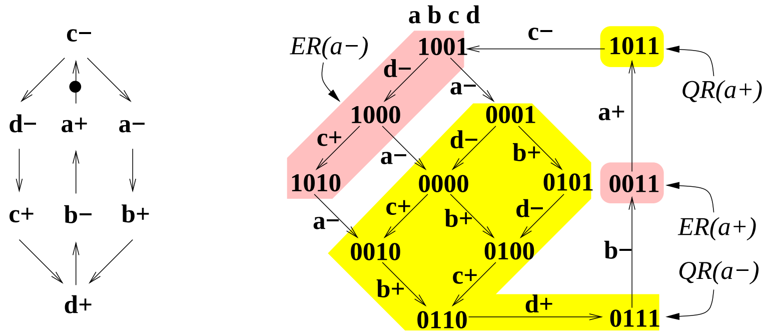
- the *positive excitation region*, ER($x+$), which includes states where a rising transition of x is enabled;

- the *negative excitation region*, ER($x-$), which includes states where a falling transition of x is enabled;
- the *positive quiescent region*, QR($x+$), which includes states where value of x is 1 and $x-$ is not enabled;
- the *negative quiescent region*, QR($x-$), which includes states where value of x is 0 and $x+$ is not enabled.

The next-state function for signal x can be computed by the excitation and quiescent regions:

- in the ER($x+$), value of x must change from 0 to 1;
- in the ER($x-$), value of x must change from 1 to 0;
- in the QR($x+$), value of x is 1 and it must not change;
- in the QR($x-$), value of x is 0 and it must not change;.

For example, figure 2.5(a) shows a consistent and persistent STG of an asynchronous circuit with two inputs b and c and two outputs a and d. Figure 2.5(b) shows its SG with CSC. Excitations and quiescent regions for signal a are indicated.



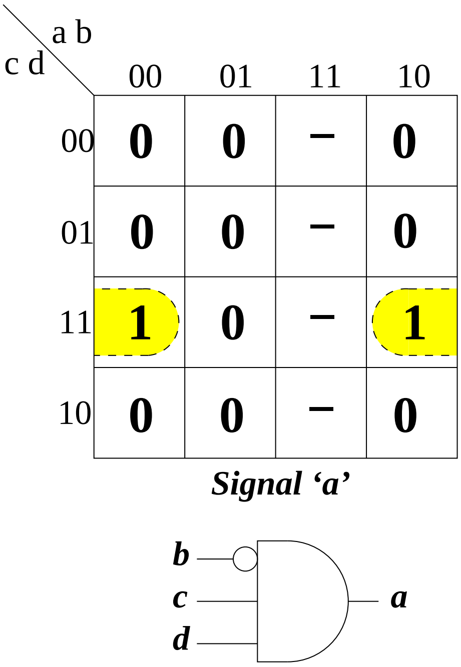
(a) **(b)**

Figure 2.5: Another asynchronous controller (a) STG and (b) SG

The logic equation for signal x can be derived, for example, by using a Karnaugh map. The next-state function can be used to fill its cells and nonreachable states can be marked as *don't cares*.

For example, figure 2.6 shows the Karnaugh map of the previous example and the logic equation after performing boolean minimization.

If CSC property is not satisfied, there are two states with the same encoding and different behavior for an output or internal signal. Therefore, the cell of this encoding in the Karnaugh of this signal contains the values 0 and 1 and no logic equation can be derived. This is the reason why CSC property is a necessary condition for logic synthesis.



c d \ a b	00	01	11	10
00	**0**	**0**	**–**	**0**
01	**0**	**0**	**–**	**0**
11	**1**	**0**	**–**	**1**
10	**0**	**0**	**–**	**0**

Figure 2.6: Karnaugh map and logic equations of the previous asynchronous controller

2.4 Handshake Circuits

Asynchronous circuits also can be specified by several computer languages. These specifications can be translated into *Handshake circuits*, which are asynchronous circuits composed of handshake components and channels. These handshake components are synchronization components which communicate through channels using a handshake protocol. This handshake protocol is a synchronization protocol composed of a request and a acknowledge. Figure 2.7 shows a handshake circuit composed of two handshake components and one channel.

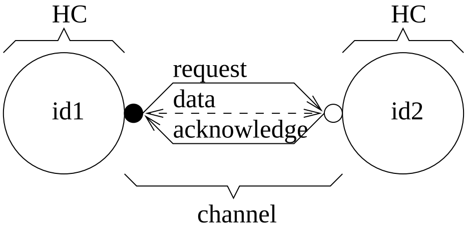


Figure 2.7: A handshake circuit

2.4.1 Handshake Protocol

The *handshake protocol* synchronizates two components through a channel. When the *active* component needs to synchronizate with the *passive* component, it sends a *request*. When the passive component receives this request, it can perform some computation and replies with an *acknowledge*. If the active (passive) component also needs to communicate some data to the passive (active) component, then data is sent with the request (acknowledge). The behavior of the protocol is correct independently of the request and acknowledge delays and the computation time.

The handshake protocol can be a two phases or a four phases protocol. If it is a two phases protocol (showed in figure 2.8(a)), changes in value of signals are interpreted as requests or acknowledges. In contrast, if it is a four phases protocol (showed in figure 2.8(b)), rising transitions in signals are interpreted as requests or acknowledges and, after handshakes, falling transitions return signals to their initial state.

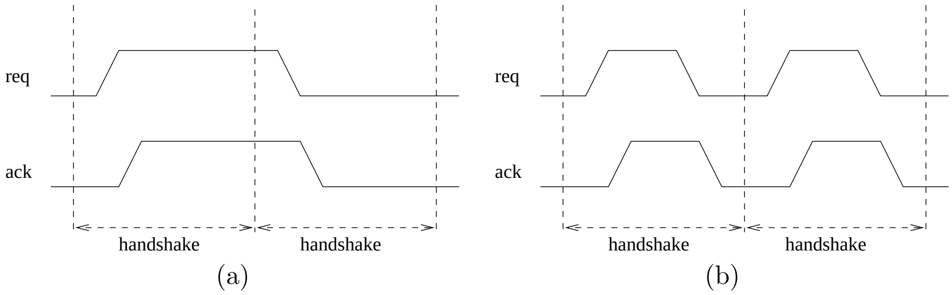


Figure 2.8: The handshake protocol: (a) 2 phases and (b) 4 phases

2.4.2 Handshake Components

Every *Handshake Component (HC)* has a function and is composed of ports. Its function is determined by its behavior through its ports. If a port starts (ends) a communication, it is an *active* (*passive*) port. If a port only synchronizates, it is a *synchronization* port. In contrast, if a port sends (receives) some data, it is a *output* (*input*) port. Figure 2.9 shows a HC (drawn as a big circle), composed of a passive synchronization port 0 (drawn as a little no filled circle with a line), an active output port 1 (drawn as a little filled circle with an output arrow) and a passive input port 2 (drawn as a little no filled circle with an input arrow). Its function is indicated with an identifier (drawn in the middle of the big circle).

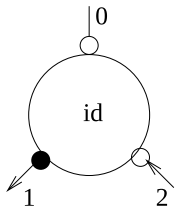


Figure 2.9: A handshake component

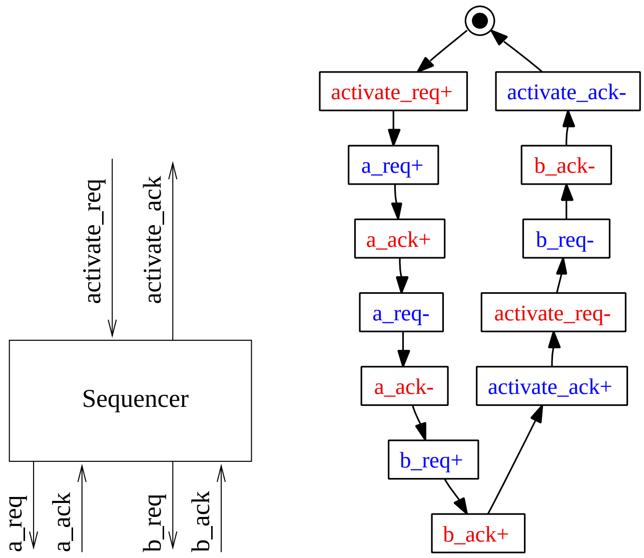
Chapter 3

State of the Art and Problem Definition

This chapter presents several methodologies for the design of asynchronous circuits. Section 3.1 describes behavioral tools like Petrify, Assassin and Minimalist. Section 3.2 describes hardware description languages like Tangram and Balsa. Section 3.3 describes structural methods like implemented in Moebius.

3.1 Behavioral Tools

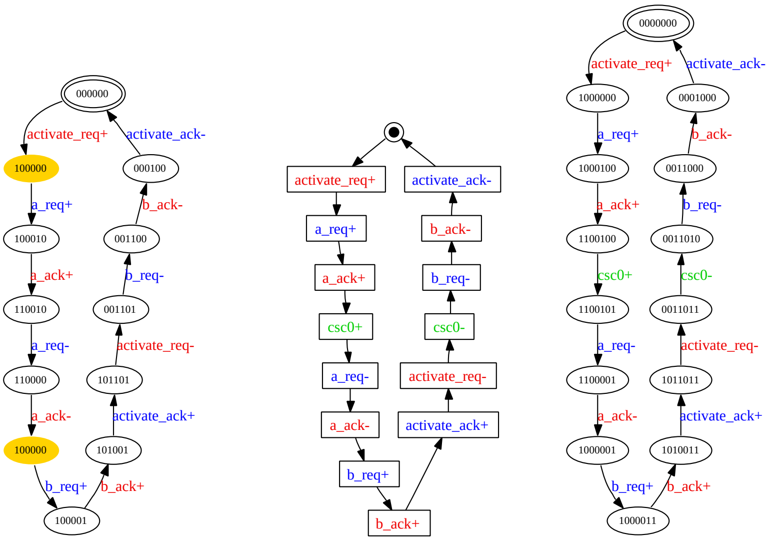
As it is described in section 2.2, an asynchronous circuit can be specified by a STG. For example, figure 3.1 shows the interface and the STG of the *Sequencer* asynchronous protocol.



(a) (b)

Figure 3.1: The *Sequencer* asynchronous protocol (a) interface and (b) STG

An asynchronous circuit can be implemented by behavioral tools. Given a STG, they enforce the CSC, minimize the logic and construct a SI circuit. For example, figure 3.2(a) shows the SG with a CSC conflict of the *Sequencer* asynchronous protocol. Two different states, which are indicated with filled circles in the figure, have the same encoding *100000*, which is the value of the signals, but the output behavior expected in each state is different from the other. Figure 3.2(b) shows the STG of the *Sequencer* with a new internal signal *csc0* inserted by behavioral tools. Figure 3.2(c) shows the SG without CSC conflicts of the *Sequencer*. The previous two different states with different behavior have now different encoding.



(a) (b) (c)

Figure 3.2: The *Sequencer* asynchronous protocol (a) SG with a CSC conflict, (b) STG with signal *csc0* and (c) SG without CSC conflicts

Several behavioral tools have been implemented, as Petrify [5], Assassin [12] and Minimalist [8].

3.1.1 Petrify

Petrify is a tool for the synthesis and optimization of asynchronous circuits. Given a STG, it produces an optimized net-list of an asynchronous circuit in the target gate library while preserving the specified input-output behavior.

The design flow of Petrify is showed in figure 3.3. It translates a STG into a SG and performs state assignment by solving the CSC problem. State assignment is coupled with logic minimization and SI technology mapping. The synthesis technique is based on the theory of regions.

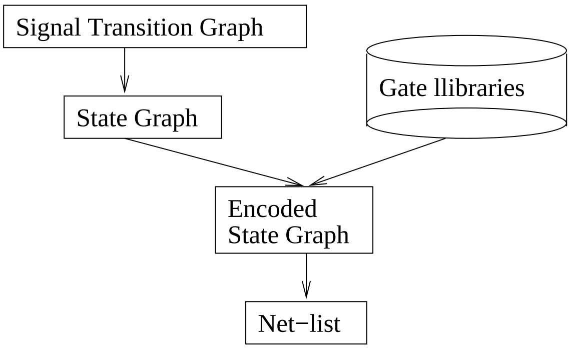


Figure 3.3: The design flow of Petrify

3.2 Hardware Description Languages

It is difficult to specify large asynchronous circuits using STGs. How can we specify, for example, a simple processor? Asynchronous circuits also can be specified by *Hardware Description Languages (HDLs)*, which are computer languages for formal description of the temporal behavior and spatial structure of electronic systems. Handshake circuits can be obtained by *Syntax Directed Translation (SDT)*, which is a linear method of translating specifications into nets of HCs.

For example, the *ssem*, a simple processor, can be specified with a hundred of lines. Figure 3.4 shows the net of HCs resulting from the syntax-directed compilation.

Tangram [15] and Balsa [7] are HDLs for specifying asynchronous circuits. New flow designs have been proposed [4, 11, 14] to improve the implementation of handshake circuits control part.

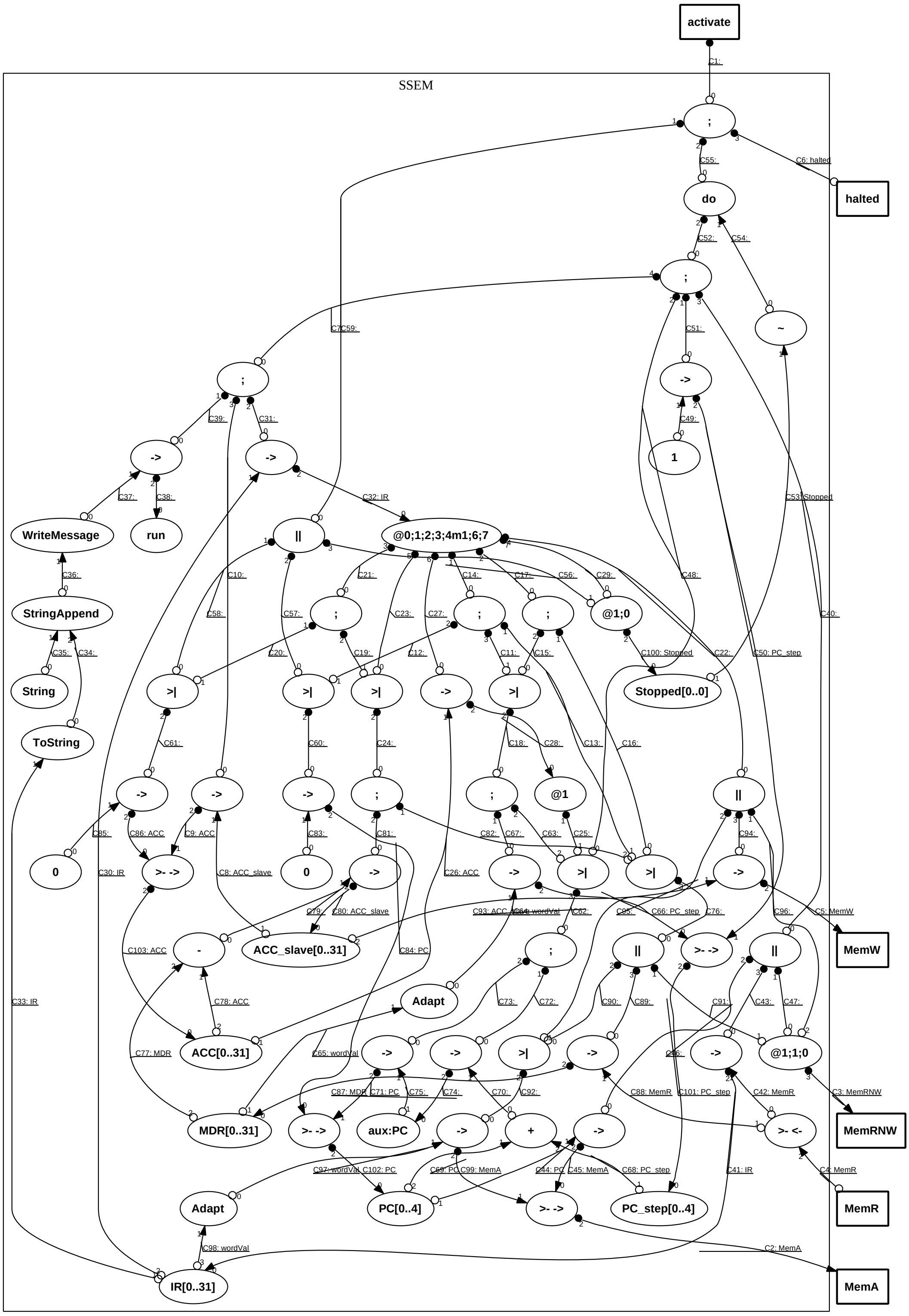


Figure 3.4: The *ssem* net of handshake components

3.2.1 Balsa

Balsa is a framework for sinthesizing and a HDL for specifying asynchronous circuits. The approach adopted, as in Tangram, is the SDT into HCs. There is a one-to-one mapping between the language and the HCs.

The design flow of Balsa framework is showed in figure 3.5. *balsa-c* compiles a Balsa specification into a Breeze specification (HC netlist) and *balsa-netlist* produces a netlist appropriate to the target technology from a Breeze description. Balsa framework also have tools to manage projects, simulate designs, etc.

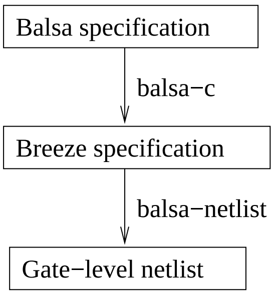


Figure 3.5: The design flow of Balsa framework

3.3 Structural Tools

Behavioral tools suffer the state explosion problem: they construct the SG to synthesize the logic. For example, figure 3.6 shows the interface, the STG and the SG with CSC conflicts of the *Parallelizer* asynchronous protocol. The number of states is exponential with respect the number of transitions and places.

Structural tools can enforce the CSC without constructing the SG. For example, figure 3.7(a) shows the STG of the *Concur* asynchronous protocol with the new internal signals *csc_1* and *csc_2* inserted by structural tools. Figure 3.7(b) shows the SG without CSC conflicts of the *Concur*.

Several structural methods [2, 3] have been implemented in Moebius. Other techniques to check the CSC without constructing the SG have been proposed [9, 10].

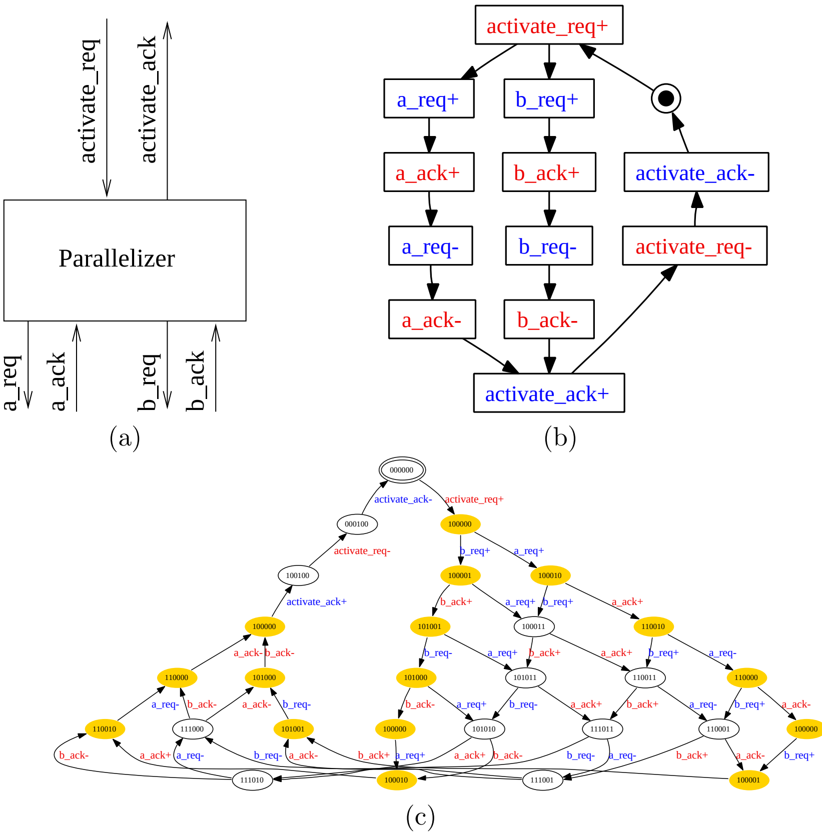


Figure 3.6: The *Parallelizer* asynchronous protocol (a) interface, (b) STG and (c) SG with CSC conflicts

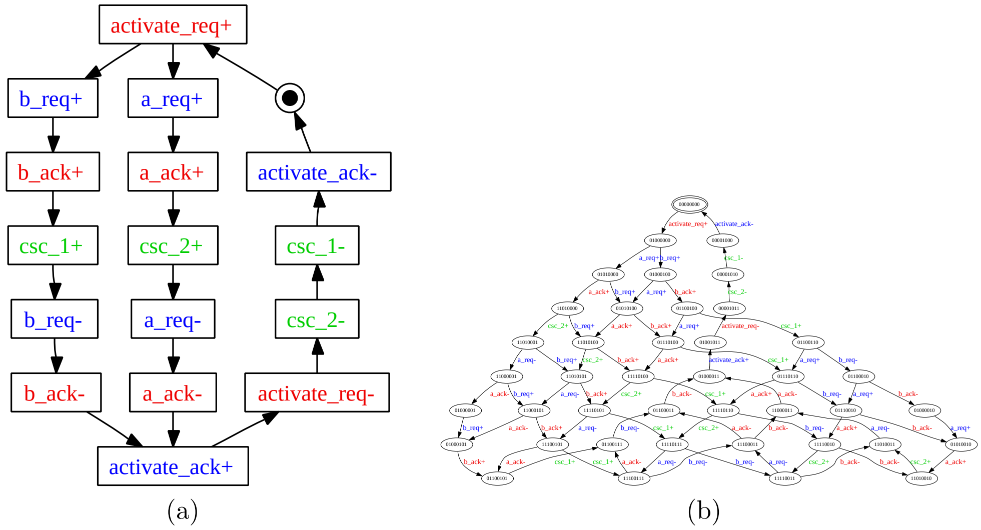


Figure 3.7: The *Parallelizer* asynchronous protocol (a) STG and (b) SG without CSC conflicts

3.3.1 Moebius

Moebius is a tool for the synthesis of asynchronous circuits. Given a STG, it produces the projections of the noninput signals into their support. It implements Integer Linear Programming (ILP) methods for checking the CSC and calculating the support of noninput signals.

The support of a signal is usually a small subset of all the signals. The greater part of the signals in a projection are labeled as dummy. Therefore, the SG of a projection has not much states and behavioral tools can synthesize its signal.

The design flow of Moebius is showed in figure 3.8. First, it encodes a STG with structural methods and forces CSC with ILP. Then, it optimizes the STG and checks CSC with ILP. After that, it projects the noninput signals into their support and calculates them with ILP. Finally, the projections of the noninput signals can be synthesized with state-based tools, as Petrify.

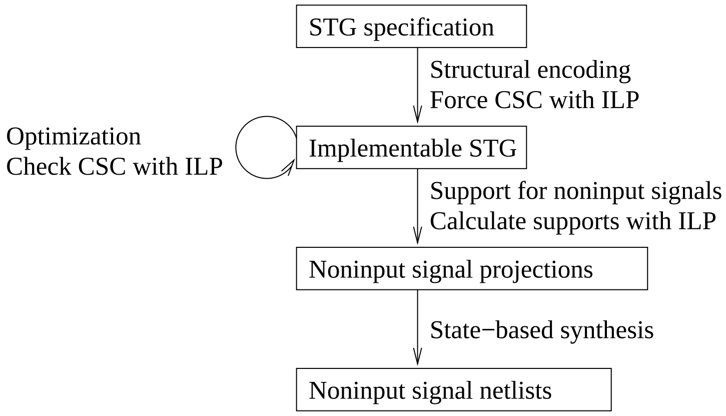


Figure 3.8: The design flow of Moebius

Figure 3.9(a) shows the projection STG into the signal *a_req* support (*a_req*, *activate_req* and *csc_2*) of the *Concur* asynchronous protocol. The number of nondummy signals is fewer than in the previous STG. Figure 3.9(b) shows the projection SG into the signal *a_req* support of the *Concur*. The number of states is fewer than in the previous SG. The STGs of the projections now can be synthesized by behavioral tools.

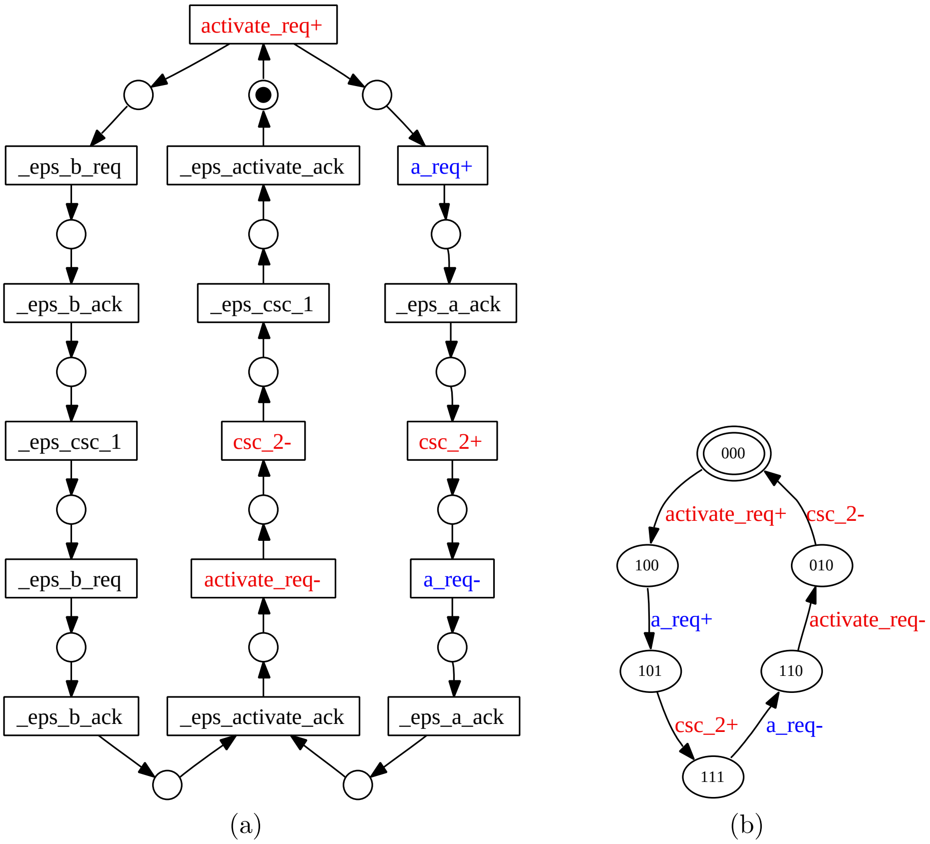


Figure 3.9: The signal *a_req* projection (a) STG and (b) SG of the *Concur* asynchronous protocol. Dummy transitions are denoted by the prefix "_eps"

Chapter 4

Methodology and Strategies

This chapter presents methodologies and strategies to benefit from advantages and avoid from disadvantages of behavioral tools, hardware description languages and structural tools.

HDLs are easy tools for specifying asynchronous circuits. However, HDL implementations can be improved by behavioral tools. But behavioral tools can suffer the state explosion problem. Therefore, structural tools can be used to minimize the state explosion problem.

Section 4.1 describes a new design flow for asynchronous circuits. Section 4.2 describes the necessary tools which have been implemented. Section 4.3 shows a simple design example.

4.1 A new design flow for asynchronous circuits

We propose a new design flow, showed in figure 4.1. Balsa is used to specify asynchronous circuits because it is easier than using STGs. First, *balsa-c* compiles a Balsa specification and produces a Breeze specification, which is a net of HCs. Then, some control HCs are clustered, obtaining the clusters of control HCs and a net of control, data and cluster HCs. After that, the control HCs in the clusters are specified using STGs, which are composed to specify the clusters of control HCs. Next, Moebius is used to enforce the CSC and split this complex STGs into the projection STGs of the noninput signals, which are simpler. Later, Petrify is used to synthesize this simple STGs, obtaining the noninput signal equations. Then, this equations are efficiently mapped into a basic gate library and they are translated into the Abs specifications of the control HC clusters, which are necessary to implement this new cluster HCs. Finally, *balsa-netlist* is used to implement the asynchronous circuit, obtaining a gate-level netlist.

Clustering is only performed on some control HCs, because data HCs are implemented efficiently by Balsa. Moreover, there are several control HC STGs which violate the implementability properties and are not clustered.

If we use Moebius, we obtain simple STGs which can be sinthesized by Petrify, reducing the state explosion problem in logic synthesis. Therefore, logic optimization is performed on some control HCs and Balsa control part implementations can be improved.

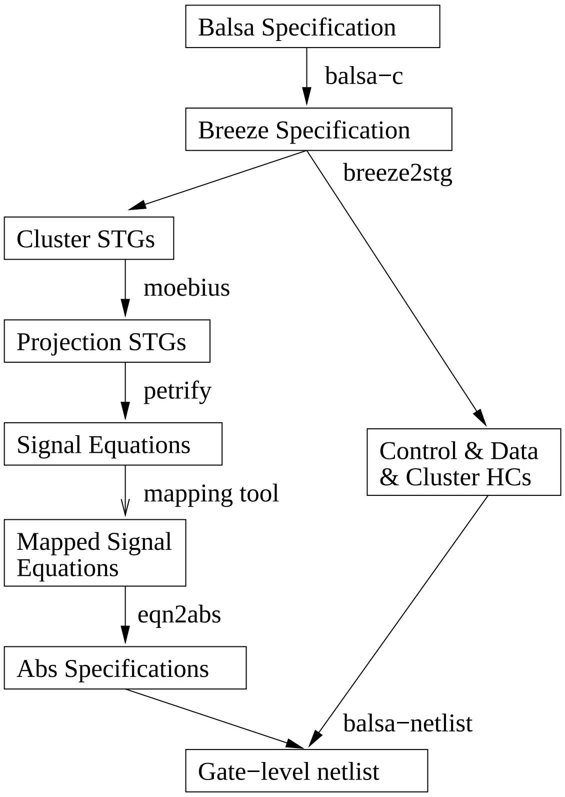


Figure 4.1: A new design flow for asynchronous circuits

4.2 breeze2stg and eqn2abs

breeze2stg is an application implemented to be used in the new design flow for asynchronous circuits. It clusters some control HC in a Breeze specification and describes their behavior with STG specifications.

Figure 4.2 shows the *breeze2stg* flow: first, *breeze2ast* reads a Breeze specification and constructs an Abstract Syntax Tree (AST). Informally, an AST can be defined as a forest with trees. Each tree is called *part definition*. For each part definition in the AST, *ast2graph* constructs a graph with channels and components as vertexs. After that, *cluster-graph* clusters some control components in the graphs, replaces them with cluster components and constructs cluster graphs. Next, *graph2ast* replaces channels and components in the part definitions AST with them in the clustered graphs. Later, for each cluster graph, *graph2stg* describes the cluster behavior with a STG and *SRR* performs simple reduction rules on it. Finally, *ast2breeze* writes the clustered AST on a clustered Breeze specification and *stg2g* writes the cluster STGs on cluster STG specifications. The detailed definition of each step is described below.

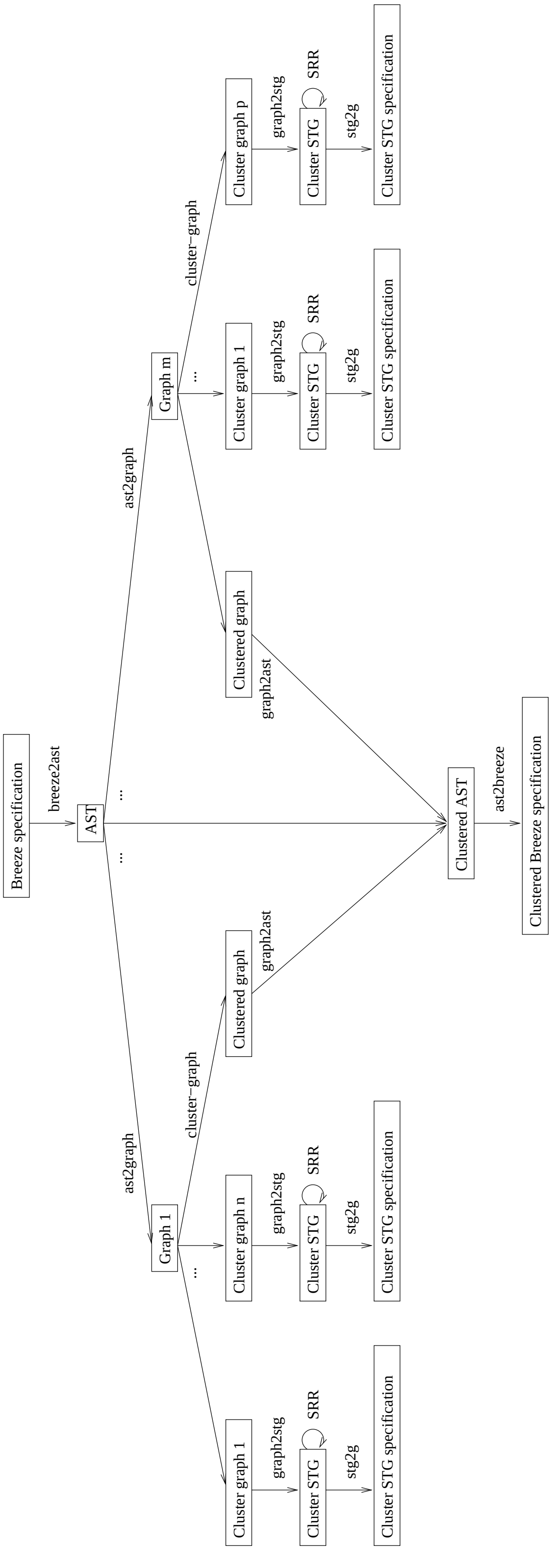


Figure 4.2: The *breeze2stg* flow

breeze2ast reads a Breeze specification and constructs an AST. A *breeze file* has zero or more *part definitions* which have a set of *channels* and a set of *components*. Each *component* has an *identifier* and a set of *channel numbers*.

ast2graph constructs a graph with channels and components as vertexs from a part definition AST. First, it adds a vertex for each *channel*. Then, it adds a vertex for each *component*. Finally, it adds an edge for each *channel number* between its *component* and the indicated *channel*. Channels are processed as vertexs, instead of edges, because the construction of the graph is guided by the channel numbers in component definitions. The order of a channel connection in a component is stored in the edge between the channel and the component.

cluster-graph clusters some control components in a graph, replaces them with cluster components and constructs cluster graphs. This is the pseudocode of the clustering algorithm:

```
procedure cluster(graph)
  for each component ∈ vertexs(graph) do
    if is_clustering(component) and not is_splitting(component)
      if cluster(graph, component, cluster_graph)
        // graph2stg //

function cluster(graph, cluster, cluster_graph)
  for each channel ∈ neighbours(cluster) do
    if not is_splitting(channel)
      for each component ∈ neighbours(channel) do
        if is_clustering(component) and not is_splitting(component)
          remove(graph, component)
          connect(cluster, component)
          add(cluster_graph, component)
          cluster(graph, cluster, cluster_graph)
          return true
  return false
```

Procedure *cluster* searches components which satisfy properties to be clustered and are not selected components to split big clusters. For each component founded, it tries to cluster this component with others. If it achieves this clustering, *graph2stg* is called.

Function *cluster* searches channels which are cluster neighbours and are not selected channels to split big clusters. For each channel founded, it searches components which are channel neighbours, satisfy properties to be clustered and are not selected components to split big clusters. For each component founded, it removes this component from the *graph*, connects the *cluster* with this component neighbour channels, adds this component to the *cluster_graph* and calls itself recursively.

This clustering algorithm performs the biggest possible clusters by selecting components as clusters and trying to expand them as much as possible. If clusters obtained are too big, they can be splitted by selecting some channels and/or components which can not be clustered.

graph2ast replaces channels and components in a part definition AST with them in a clustered graph. First, it removes *channels* in the part definition AST and, for each channel vertex in the clustered graph, it adds a *channel* in the AST part definition. Then, it removes *components* in the part definition AST and, for each component vertex in the clustered graph, it adds a *component* in the AST part definition. Finally, for each edge between a channel and a component in the clustered graph, it adds the *channel number* indicated by this channel in this *component* AST.

graph2stg describes the behavior of a cluster graph with a STG. For each component vertex, it describes the behavior of the component with a STG. Channel vertexs are used to label signal transitions of component STGs. Therefore, components STGs are composed automatically by sharing signal transitions.

Only a few components have been clustered and described with STG. Some statistics have been performed to select this components. Table 4.1 shows the components frequency in Balsa examples. All control components without data ports have been selected (marked with boldface in the table), except *Loop* component which has a non-consistent STG specification.

The behavioral descriptions of selected HCs using STGs can be founded in appendix A.

Component	Frequency
Fetch	155
FalseVariable	69
Synch	**51**
SequenceOptimised	**42**
BinaryFunc	36
Concur	**34**
CallMux	26
DecisionWait	**24**
ContinuePush	19
Variable	17
Loop	17
Combine	15
Case	13
Fork	**13**
Constant	9
UnaryFunc	8
Call	**8**
CombineEqual	4
Arbiter	4
BinaryFuncConstR	4
Encode	4
Adapt	3
CallDemux	3
WireFork	2
PassivatorPush	2
Continue	1
While	1
NullAdapt	1

Table 4.1: The components frequency in Balsa examples

SRR performs Simple Reduction Rules [13] on a cluster STG to hide dummy transitions and reduce the complexity of the synthesis.

Fusion of Series Places (FSP) as figure 4.3(a) shows.

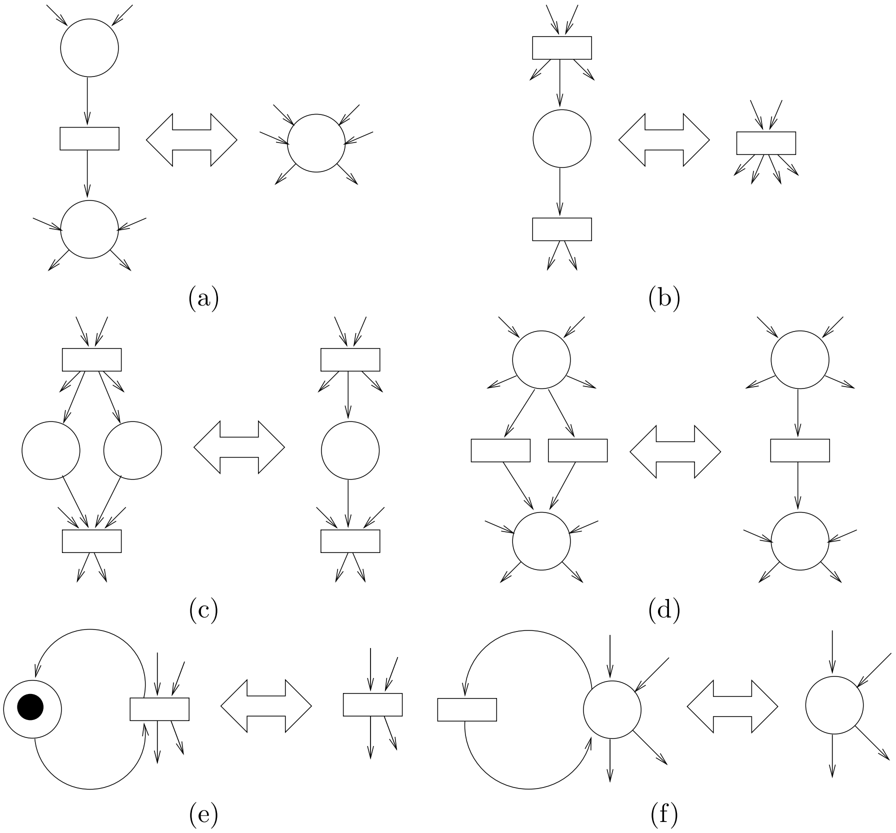
Fusion of Series Transitions (FST) as figure 4.3(a) shows.

Fusion of Parallel Places (FPP) as figure 4.3(a) shows.

Fusion of Parallel Transitions (FPT) as figure 4.3(a) shows.

Elimination of Self-loop Places (ESP) as figure 4.3(a) shows.

Elimination of Self-loop Transitions (EST) as figure 4.3(a) shows.



(a) (b) (c) (d) (e) (f)

Figure 4.3: The SRR: (a) FSP, (b) FST, (c) FPP, (d) FPT, (e) ESP and (f) EST

eqn2abs is another application implemented to be used in the new design flow for asynchronous circuits. It basically translates the cluster signal equations obtained with a mapping tool into the cluster Abs specification which *balsa-netlist* uses to implement the cluster.

4.3 Example of two synchronized buffers

In this section, two synchronized buffers are designed. Their Balsa specification is the following:

```
import [balsa.types.basic]
procedure seq_par (input i1 : byte; output o1 : byte; input i2 : byte; output o2 : byte) is
  variable x1 : byte
  variable x2 : byte
begin
  loop
        [i1 -> x1
      ||
        i2 -> x2]
    ;
        [o1 <- x1
      ||
        o2 <- x2]
  end
end
```

The two buffers concurrently read some data from the input and write this data to the output.

First, *balsa-c* compiles the Balsa specification to obtain the following Breeze Specification:

```
(import ”balsa.types.builtin”)
(breeze-part ”seq_par”
   (ports
      ...
   )
   (attributes
      ...
   )
   (channels
      (sync (at 6 3 ”seq-par.balsa” 1)) ; 1
      (pull 8 (at 7 10 ”seq-par.balsa” 1) (name ”i1”)) ; 2
      (push 8 (at 11 10 ”seq-par.balsa” 1) (name ”o1”)) ; 3
      (pull 8 (at 9 9 ”seq-par.balsa” 1) (name ”i2”)) ; 4
      (push 8 (at 13 9 ”seq-par.balsa” 1) (name ”o2”)) ; 5
      (pull 8 (at 13 15 ”seq-par.balsa” 1) (name ”x2”)) ; 6
      (sync (at 13 12 ”seq-par.balsa” 1)) ; 7
      (pull 8 (at 11 16 ”seq-par.balsa” 1) (name ”x1”)) ; 8
      (sync (at 11 13 ”seq-par.balsa” 1)) ; 9
      (sync (at 12 7 ”seq-par.balsa” 1)) ; 10
      (push 8 (at 9 12 ”seq-par.balsa” 1) (name ”x2”)) ; 11
      (sync (at 9 12 ”seq-par.balsa” 1)) ; 12
      (push 8 (at 7 13 ”seq-par.balsa” 1) (name ”x1”)) ; 13
      (sync (at 7 13 ”seq-par.balsa” 1)) ; 14
      (sync (at 8 7 ”seq-par.balsa” 1)) ; 15
      (sync (at 10 5 ”seq-par.balsa” 1)) ; 16
   )
   (components
      (component ”$BrzVariable” (8 1 ”x1[0..7]” ””) (13 (8)) (at 3 3 ”seq-par.balsa” 0)) ; 0
      (component ”$BrzVariable” (8 1 ”x2[0..7]” ””) (11 (6)) (at 4 3 ”seq-par.balsa” 0)) ; 1
      (component ”$BrzLoop” () (1 16)) ; 2
      (component ”$BrzSequenceOptimised” (2 ”S”) (16 (15 10))) ; 3
      (component ”$BrzConcur” (2) (15 (14 12))) ; 4
      (component ”$BrzFetch” (8 ”false”) (14 2 13)) ; 5
      (component ”$BrzFetch” (8 ”false”) (12 4 11)) ; 6
      (component ”$BrzConcur” (2) (10 (9 7))) ; 7
      (component ”$BrzFetch” (8 ”false”) (9 8 3)) ; 8
      (component ”$BrzFetch” (8 ”false”) (7 6 5)) ; 9
   )
)
```

Then, *breeze2stg* steps are performed. First, *breeze2ast* step constructs the AST of the Breeze specification and *ast2graph* step constructs the graph of the part definition AST.

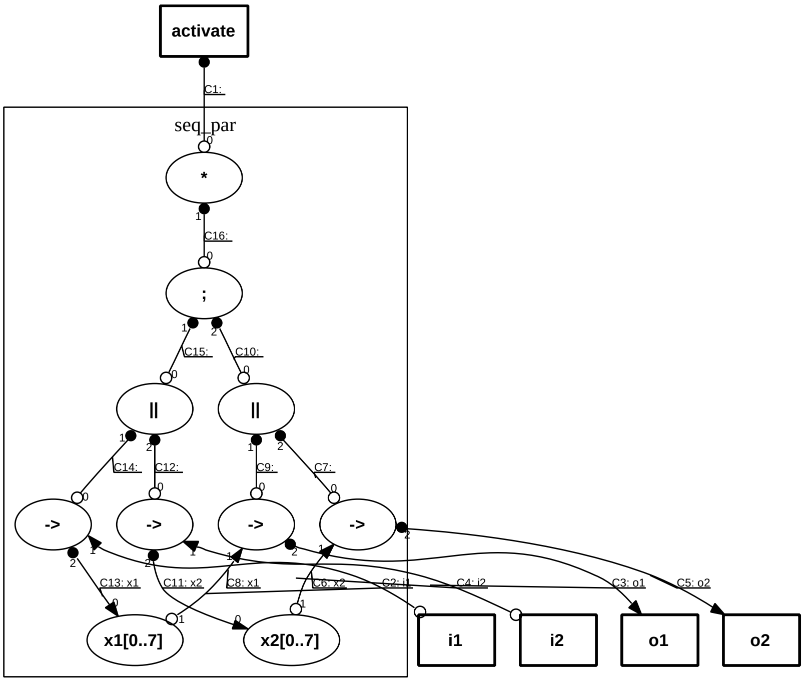
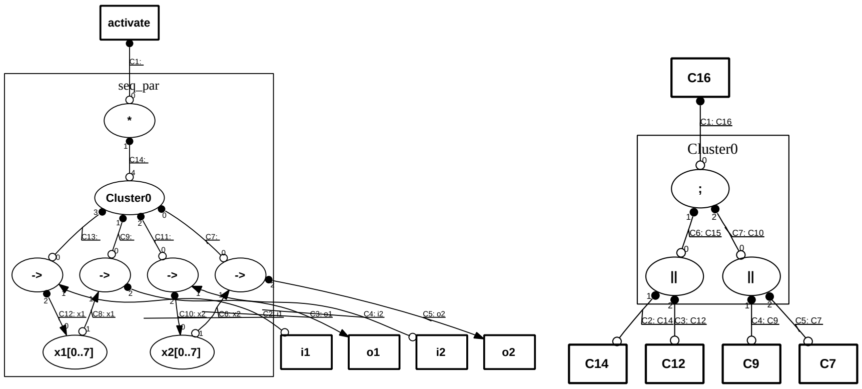


Figure 4.4: The constructed graph of the two synchronized buffers

After that, *cluster-graph* step clusters some control components to obtain the clustered graph and the cluster graph.



(a) (b)

Figure 4.5: The two synchronized buffers (a) clustered graph and (b) cluster graph

The *SequenceOptimised* (identified as ";") and the two *Concur* (identified as "||") components have been clustered.

Next, *graph2ast* step replaces channels an components of the original part definition AST with them of the clustered graph and *ast2breeze* step writes the following clustered Breeze specification:

```
( import ”balsa.types.builtin” )
( breeze-part ”seq_par”
  ( ports
...
  )
  ( attributes
...
  )
  ( channels
    ( sync ( at 6 3 ”seq-par.balsa” 1 ) )
    ( pull 8 ( at 7 10 ”seq-par.balsa” 1 ) ( name ”i1” ) )
    ( push 8 ( at 11 10 ”seq-par.balsa” 1 ) ( name ”o1” ) )
    ( pull 8 ( at 9 9 ”seq-par.balsa” 1 ) ( name ”i2” ) )
    ( push 8 ( at 13 9 ”seq-par.balsa” 1 ) ( name ”o2” ) )
    ( pull 8 ( at 13 15 ”seq-par.balsa” 1 ) ( name ”x2” ) )
    ( sync ( at 13 12 ”seq-par.balsa” 1 ) )
    ( pull 8 ( at 11 16 ”seq-par.balsa” 1 ) ( name ”x1” ) )
    ( sync ( at 11 13 ”seq-par.balsa” 1 ) )
    ( push 8 ( at 9 12 ”seq-par.balsa” 1 ) ( name ”x2” ) )
    ( sync ( at 9 12 ”seq-par.balsa” 1 ) )
    ( push 8 ( at 7 13 ”seq-par.balsa” 1 ) ( name ”x1” ) )
    ( sync ( at 7 13 ”seq-par.balsa” 1 ) )
    ( sync ( at 10 5 ”seq-par.balsa” 1 ) )
  )
  ( components
    ( component ”$BrzVariable” ( 8 1 ”x1[0..7]” ”” ) ( 12 ( 8 ) ) ( at 3 3 ”seq-par.balsa” 0 ) )
    ( component ”$BrzVariable” ( 8 1 ”x2[0..7]” ”” ) ( 10 ( 6 ) ) ( at 4 3 ”seq-par.balsa” 0 ) )
    ( component ”$BrzLoop” ( ) ( 1 14 ) )
    ( component ”$BrzFetch” ( 8 ”false” ) ( 13 2 12 ) )
    ( component ”$BrzFetch” ( 8 ”false” ) ( 11 4 10 ) )
    ( component ”$BrzFetch” ( 8 ”false” ) ( 9 8 3 ) )
    ( component ”$BrzFetch” ( 8 ”false” ) ( 7 6 5 ) )
    ( component ”$BrzCluster0” ( ) ( 7 9 11 13 14 ) )
  )
)
```

There is not any *SequenceOptimised* or *Concur* component but there is a *Cluster0* component with the *channel numbers* of the clustered components.

Later, *graph2stg* step describes the behavior of the cluster graph with the following signal transition graph (STG):

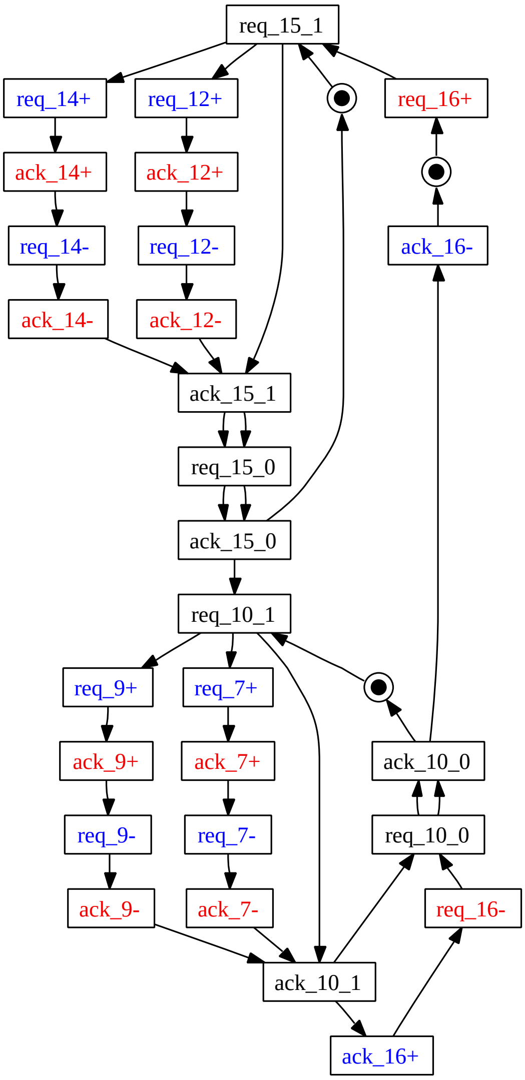


Figure 4.6: The cluster STG of the two synchronized buffers

The *SequenceOptimised* STG (channels 16, 15 and 10), the first *Concur* STG (channels 15, 14 and 12) and the second *Concur* STG (channels 10, 9 and 7) are composed automatically by sharing the 15 and 10 channel transitions. These shared transitions are marked as dummy because their channels are internal and are not in the cluster interface.

Finally, *SRR* step performs simple reduction rules on the cluster STG and *stg2g* step writes the cluster STG to a file.

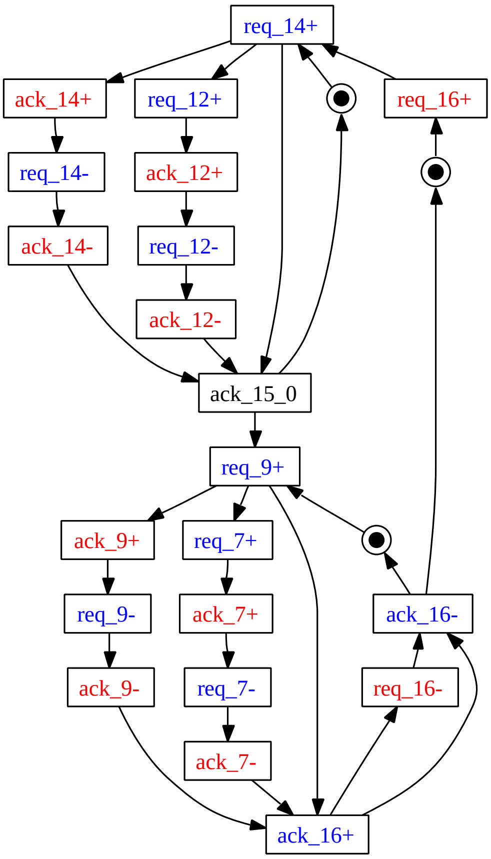


Figure 4.7: The cluster STG of the two synchronized buffers after performing SRR

Almost all dummy transitions have been hidden.

After *breeze2stg*, Moebius enforces the complete state coding (CSC) and splits the complex STG into the projections STGs of the noninput signals. Next, Petrify synthesizes this simple STGs to obtain the noninput signal equations. Later, this equations are efficiently mapped into a basic gate library to obtain the following equations:

```
[252] =!csc_2;
[253] =!csc_3;
[254] =![253]+![252];
[ack_16] =![254]*!csc_1*!ack_7*!ack_9;
[249] =!csc_1;
[202] =!ack_7*![249];
[csc_1] = [202] + ack_14;
[324] =!csc_2+![253];
[csc_2] =!req_16+![324];
[198] =!ack_9*![253];
[csc_3] = [198] + ack_12;
[191] =!csc_1*!req_14;
[req_12] =![191]*![252];
[196] = csc_2 * req_16;
[192] =!req_14*![196];
[req_14] =!csc_1*![192];
[194] =!csc_2*!csc_3;
[193] =!req_9*![194];
[req_7] =![193]*![249];
[req_9] =!ack_14*![253]*!ack_12*!req_14;
```

Then, *eqn2abs* translates this equations into the following cluster Abs specification:

```
(nodes
   (”_X252” 1 0 1)
   ...
   (”_X193” 1 0 1)
)
(gates
   ( inv (node ”_X252”) (node ”csc_2”) )
   ( inv (node ”_X253”) (node ”csc_3”) )
   ( nand (node ”_X254”) (node ”_X253”) (node ”_X252”) )
   ( nor (ack ”_16”) (node ”_X254”) (node ”csc_1”) (ack ”_7”) (ack ”_9”) )
   ( inv (node ”_X249”) (node ”csc_1”) )
   ( nor (node ”_X202”) (ack ”_7”) (node ”_X249”) )
   ( or (node ”csc_1”) (node ”_X202”) (ack ”_14”) )
   ( nand (node ”_X324”) (node ”csc_2”) (node ”_X253”) )
   ( nand (node ”csc_2”) (req ”_16”) (node ”_X324”) )
   ( nor (node ”_X198”) (ack ”_9”) (node ”_X253”) )
   ( or (node ”csc_3”) (node ”_X198”) (ack ”_12”) )
   ( nor (node ”_X191”) (node ”csc_1”) (req ”_14”) )
   ( nor (req ”_12”) (node ”_X191”) (node ”_X252”) )
   ( and (node ”_X196”) (node ”csc_2”) (req ”_16”) )
   ( nor (node ”_X192”) (req ”_14”) (node ”_X196”) )
   ( nor (req ”_14”) (node ”csc_1”) (node ”_X192”) )
   ( nor (node ”_X194”) (node ”csc_2”) (node ”csc_3”) )
   ( nor (node ”_X193”) (req ”_9”) (node ”_X194”) )
   ( nor (req ”_7”) (node ”_X193”) (node ”_X249”) )
   ( nor (req ”_9”) (ack ”_14”) (node ”_X253”) (ack ”_12”) (req ”_14”) ) )
```

Finally, *balsa-netlist* implements the clustered Breeze specification using the cluster Abs specification.

Chapter 5

Experimental Results and Evaluation

This chapter presents the results obtained performing the new design flow on the following Balsa examples: arbiter tree, population counter, shifter, SSEM processor and stack.

Table 5.1 shows the following information: column *Cluster* reports the cluster identifiers, then the number of HCs, number of places and transitions in its signal transition graphs and the area of Balsa and the new design flow cluster implementations are reported. For example, PopCount has been splitted into 13 clusters. The cluster 0, 3, 4 and 9 consist of 3 HCs and the underlying STGs contains 34 places and 24 transitions. The area of the Balsa implementation for each one of these clusters is 408, whereas the implementation of the new design flow is 104.

Figure 5.1 compares, for each cluster in the Balsa examples, the area of Balsa and the new design flow implementations. All new design flow implementations, excepts SSEM Cluster 0, are better in area than Balsa implementations.

The HC net, clustered HC net, clusters HC nets and clusters STGs of the Balsa examples can be founded in appendix B.

Example	Cluster	#HC	|P|	|T|	Balsa	New Design Flow
ArbTree	0	13	156	118	3464	**2744**
PopCount	0,3,4,9	5	54	36	680	**312**
	1,2,5,6,8,11,12	3	34	24	408	**104**
	7	4	44	30	544	**208**
Shifter	0	7	87	59	1920	**960**
SSEM	0	7	74	62	**2112**	2400
	1	6	73	57	2208	**1600**
	2	7	89	69	2240	**2160**
Stack	0	18	214	137	4160	**2344**
	1	18	209	135	3984	**2024**
	2	23	256	187	9720	**7208**

Table 5.1: The Balsa examples results

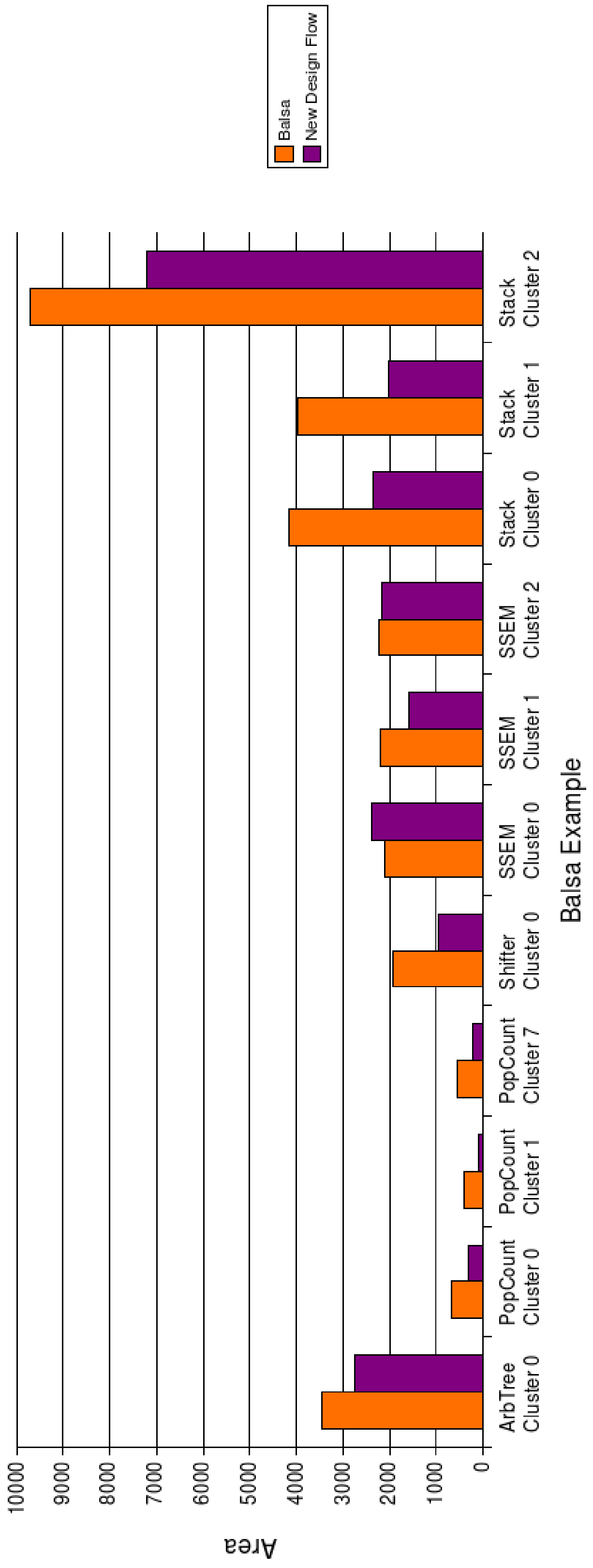


Figure 5.1: The Balsa examples area results

Chapter 6

Discussion and Conclusions

A new design flow of asynchronous circuits has been presented. It combines the advantages of the existent tools to easily specify and efficiently implement asynchronous circuits. HDL implementations are improved with logic synthesis and the state explosion problem of behavioral tools is minimized with structural tools.

Two necessary applications to perform the new design flow have been implemented. One application clusters HC nets resulting from the SDT of HDL specifications and specifies the resulting clusters with STGs. The other application translates cluster signals equations resulting from logic synthesis into HDL cluster specifications.

The new flow design have been performed on several HDL specifications. Structural methods have minimized the state explosion problem and have allowed performing logic synthesis. The experimental results obtained shows the improvement of HDL implementations.

As future work, all control HCs could be specified with STGs. New clustering techniques also could be implemented to avoid the indication of the splitting components and channels in complex designs. Algorithms to eliminate redundant places also could be implemented.

Appendix A

Description of HCs behavior using STGs

This appendix describes the behavior of the selected HCs using STGs.

In [1], the behavior of HCs is described with a notation in terms of the sequencing, concurrency and enclosure of their handshakes. To describe this behavior with a STG, term expasion must be applied to terms of the notation. In a 4 phase handshake protocol (see section 2.4), each term expands into the up phase (indicated by $\triangle$) and the down phase (indicated by $\triangledown$). A complete expansion of a description can be obtained by sequentially composing up and down phases.

In a complete expansion, c_r (c_a) denotes the request (acknowledge) signal of port c, $\uparrow$ ($\downarrow$) denotes the rising (falling) transition of a signal, ";" ("||") denotes sequentially (concurrency), "|" denotes choice and "*" denotes repetition.

These are the term expansions of the notation:

Active synchronization communication c

$\triangle(\text{ c }) = c_r \uparrow ; c_a \uparrow$

$\triangledown(\text{ c }) = c_r \downarrow ; c_a \downarrow$

Sequencing A ; B

$\triangle(\text{ A ; B }) = \triangle(\text{A}) ; \triangledown(\text{A}) ; \triangle(\text{B})$

$\triangledown(\text{ A ; B }) = \triangledown(\text{B})$

Concurrency A || B

$\triangle(\text{ A } || \text{ B }) = (\ \triangle(\text{A}) ; \triangledown(\text{A})\ ) \mid\mid (\ \triangle(\text{B}) ; \triangledown(\text{B})\ )$

$\triangledown(\text{ A } || \text{ B }) = \varepsilon$

Concurrency with synchronized phases A , B

$\triangle(\text{ A , B }) = \triangle(\text{A}) \mid\mid \triangle(\text{B})$

$\triangledown(\text{ A , B }) = \triangledown(\text{A}) \mid\mid \triangledown(\text{B})$

Enclosure c : C

$\triangle(\text{ c : C }) = c_r \uparrow ; \triangle(\text{C}) ; c_a \uparrow$

$\triangledown(\text{ c : C }) = c_r \downarrow ; \triangledown(\text{C}) ; c_a \downarrow$

Precedence overriding/grouping [C]

$\triangle(\ [\ \mathrm{C}\ ]\ ) = \triangle(\mathrm{C})$

$\triangledown(\ [\ \mathrm{C}\ ]\ ) = \triangledown(\mathrm{C})$

Indefinite repetition #[C]

$\triangle(\ \#[\ \mathrm{C}\ ]\ ) = (\ \triangle(\mathrm{C})\ ;\ \triangledown(\mathrm{C})\ )\ ^*$

$\triangledown(\ \#[\ \mathrm{C}\ ]\ ) = \varepsilon$

Communication choice [a : A | b : B]

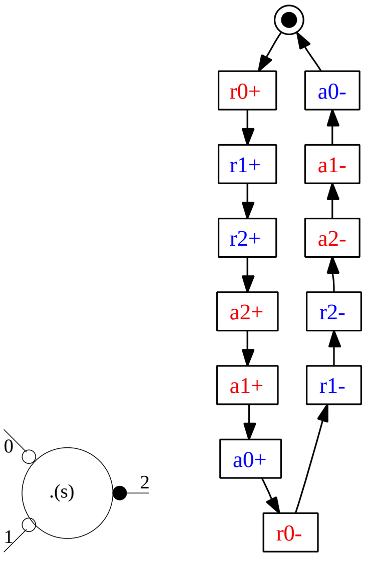
$\triangle(\ [\ \mathrm{a} : \mathrm{A} \mid \mathrm{b} : \mathrm{B}\ ]\ ) = a_r \uparrow\ ;\ \triangle(\mathrm{A})\ ;\ a_a \uparrow \mid b_r \uparrow\ ;\ \triangle(\mathrm{B})\ ;\ b_a \uparrow$

$\triangledown(\ [\ \mathrm{a} : \mathrm{A} \mid \mathrm{b} : \mathrm{B}\ ]\ ) = a_r \downarrow\ ;\ \triangledown(\mathrm{A})\ ;\ a_a \downarrow \mid b_r \downarrow\ ;\ \triangledown(\mathrm{B})\ ;\ b_a \downarrow$

Next sections present the descriptions, the complete expansion and the STGs of the selected components with 2 inputs/outputs (they can be easily extended to components with n input/outputs):

A.1 Synch

#[0 : 1 : 2]



(a) (b)

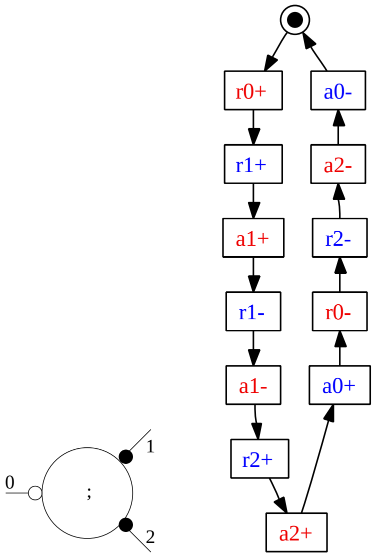
Figure A.1: (a) The *Synch* component and (b) its STG

$\triangle($ #[0 : 1 : 2] $) = (0_r \uparrow ; 1_r \uparrow ; 2_r \uparrow ; 2_a \uparrow ; 1_a \uparrow ; 0_a \uparrow ; 0_r \downarrow ; 1_r \downarrow ; 2_r \downarrow ; 2_a \downarrow ; 1_a \downarrow ; 0_a \downarrow) *$

$\bigtriangledown($ #[0 : 1 : 2] $) = \varepsilon$

A.2 SequenceOptimised

#[0 : [1 ; 2]]



(a) (b)

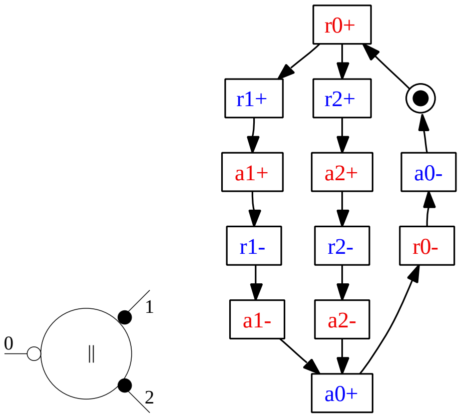
Figure A.2: (a) The *SequenceOptimised* component and (b) its STG

$\triangle$(#[0 : [1 ; 2]]) = ($0_r\uparrow$; $1_r\uparrow$; $1_a\uparrow$; $1_r\downarrow$; $1_a\downarrow$; $2_r\uparrow$; $2_a\uparrow$; $0_a\uparrow$; $0_r\downarrow$; $2_r\downarrow$; $2_a\downarrow$; $0_a\downarrow$) *

$\triangledown$(#[0 : [1 ; 2]]) = ε

A.3 Concur

#[0 : [1 || 2]]



(a) (b)

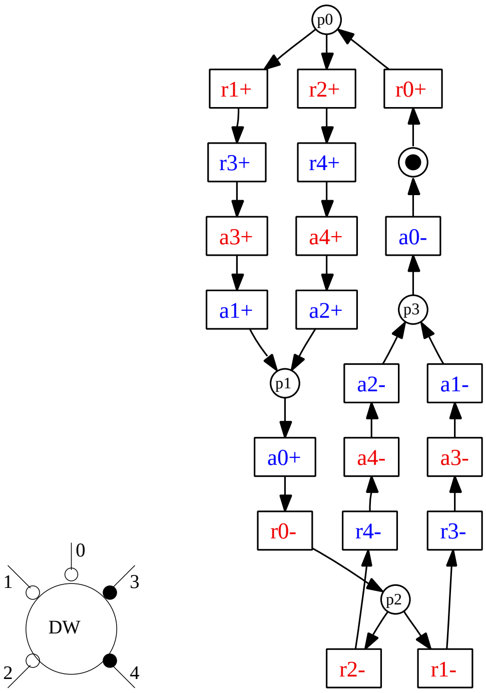
Figure A.3: (a) The *Concur* component and (b) its STG

$\triangle(\#[\ 0 : [\ 1 \mid\mid 2\ ]\ ]\ ) = (\ 0_r \uparrow ; (\ (\ 1_r \uparrow ; 1_a \uparrow ; 1_r \downarrow ; 1_a \downarrow\ ) \mid\mid (\ 2_r \uparrow ; 2_a \uparrow ; 2_r \downarrow ; 2_a \downarrow\ )\ ) ; 0_r \uparrow ; 0_r \downarrow ; 0_a \downarrow\ ) *$

$\bigtriangledown(\#[\ 0 : [\ 1 \mid\mid 2\ ]\ ]\ ) = \varepsilon$

A.4 DecisionWait

#[0 : [1 : 3 | 2 : 4]]



(a) (b)

Figure A.4: (a) The *DecisionWait* component and (b) its STG

$\triangle$(#[0 : [1 : 3 | 2 : 4]]) = ($0_r\uparrow$; (($1_r\uparrow$; $3_r\uparrow$; $3_a\uparrow$; $1_a\uparrow$) | ($2_r\uparrow$; $4_r\uparrow$; $4_a\uparrow$; $2_a\uparrow$)) ; $0_a\uparrow$; $0_r\downarrow$; (($1_r\downarrow$; $3_r\downarrow$; $3_a\downarrow$; $1_a\downarrow$) | ($2_r\downarrow$; $4_r\downarrow$; $4_a\downarrow$; $2_a\downarrow$)) ; $0_a\downarrow$) *

$\bigtriangledown$(#[0 : [1 : 3 | 2 : 4]]) = ε

The original *DecisionWait* STG can be modified because it can be non-consistent. [1] The transitions of this channels alwFor example, if channels 2 and 4 are external (their transitions are not shared), after the rising transitions of channels 1 and 3 there can be the falling transitions of channels 2 and 4. Therefore, the rising and falling transitions of channels 1, 2, 3 and 4 don't alternate in this trace. In contrast, if channels 2 and 4 are internal (their transitions are shared), after the rising transitions of channels 1 and 3 there can not be the falling transitions of channels 2 and 4 because the marking does not enable these transitions. The inconsistency can be solved by adding a place between the rising and falling transitions of the external channels 2 and 4, as figure A.5 shows. This place only enables the falling transitions of channels 2 and 4 if there have been their rising transitions.

---

[1] Inconsistency is due to the existence of transitions which are not shared by two components because their channels are in the cluster interface and only have one neighbour component. Therefore, these nonshared transitions are enabled and can fire in markings where it is not desired.

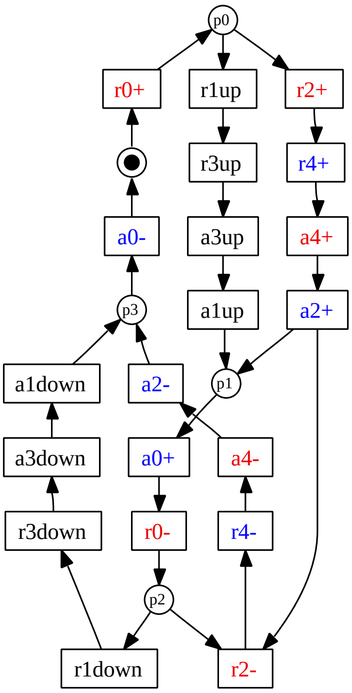
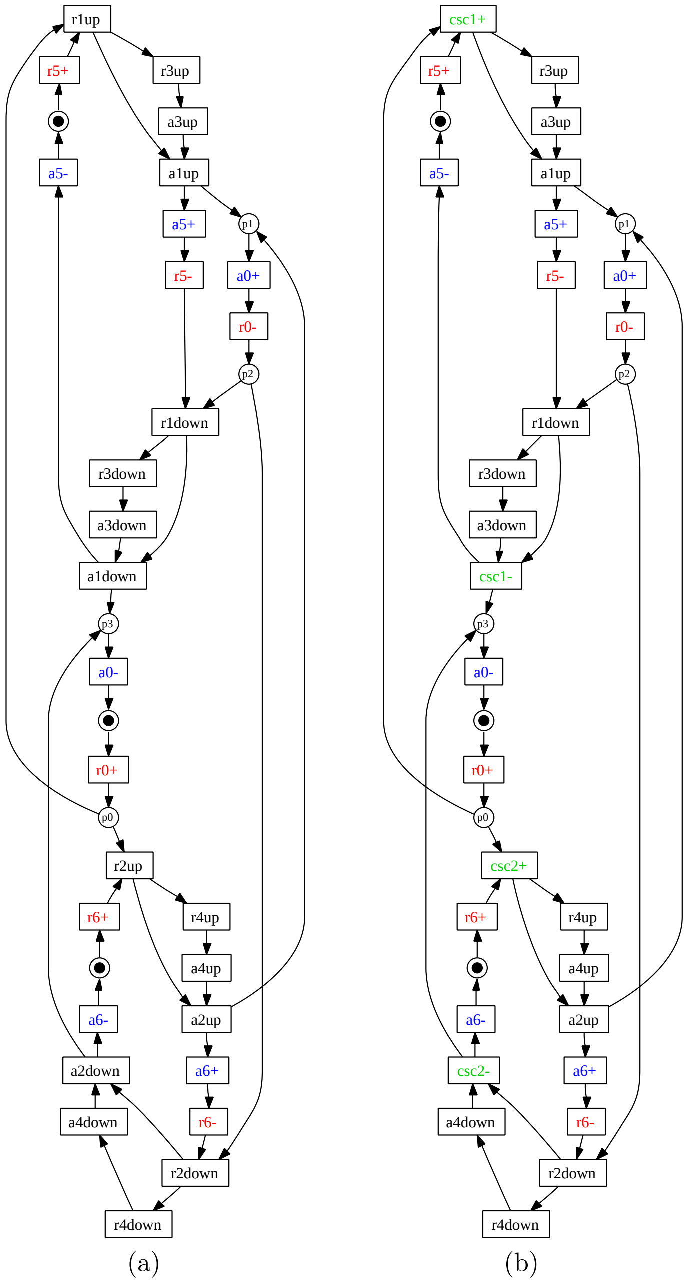


Figure A.5: The consistent *DecisionWait* STG

The original *DecisionWait* STG also can be modified because it can have irreducible CSC conflicts. [2] Figure A.6(a) shows the STG of this component with internal channels 1, 2, 3 and 4 and other external channels 5 and 6 of other components enclosing channels 1 and 2. The firing sequences *r5+ r0+ r6+* and *r6+ r0+ r5+* have the same encoding but can have different markings with different behaviors. For example, after the firing sequence *r5+ r0+*, the dummy transitions *r1up r3up a3up a1up* can fire and, after firing the transition *r6+*, only the transitions *a0+* and *a5+* are enabled. In contrast, after the firing sequence *r6+ r0+*, the dummy transitions *r2up r4up a4up a2up* can fire and, after firing the transition *r5+*, only the transitions *a0+* and *a6+* are enabled. Synthesis tools can not resolve this CSC conflicts by adding internal signals. Figure A.6(b) shows the previous STG with internal signals *csc1* and *csc2* replacing the dummy transitions *r1up*, *a1down*, *r2up* and *a2down*. The firing sequences *r5+ r0+ r6+* and *r6+ r0+ r5+*, after the firing sequences *csc1+ r3up a3up a1up* or *csc2+ r4up a4up a2up* have different encoding. Therefore, the CSC conflicts are solved.

---

[2] An irreducible conflict can not be solved without changing the behavior of the STG.

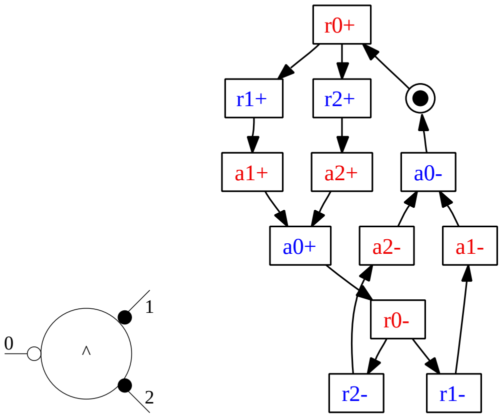


(a) (b)

Figure A.6: (a) The STG with irreducible CSC conflicts and (b) the STG without CSC conflicts of the *DecisionWait* STG

A.5 Fork

#[0 : [1 , 2]]



(a) (b)

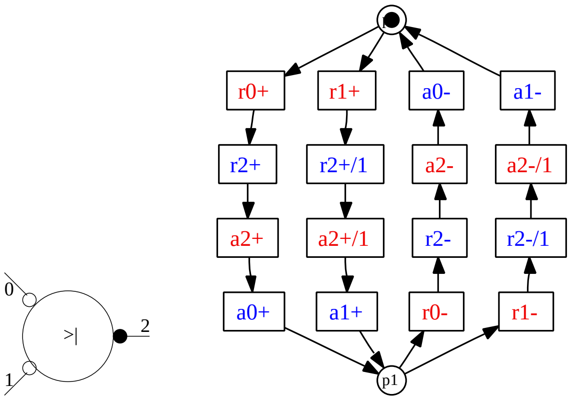
Figure A.7: (a) The *Fork* component and (b) its STG

$\triangle(\ \#[\ 0 : [\ 1\ ,\ 2\ ]\ ]\ ) = (\ 0_r \uparrow ; (\ (\ 1_r \uparrow ; 1_a \uparrow\ ) \mid\mid (\ 2_r \uparrow ; 2_a \uparrow\ )\ ) ; 0_a \uparrow ; 0_r \downarrow ; (\ (\ 1_r \downarrow ; 1_a \downarrow\ ) \mid\mid (\ 2_r \downarrow ; 2_a \downarrow\ )\ ) ; 0_a \downarrow\ )\ *$

$\triangledown(\ \#[\ 0 : [\ 1\ ,\ 2\ ]\ ]\ ) = \varepsilon$

A.6 Call

#[[0 : 2 | 1 : 2]]



(a) (b)

Figure A.8: (a) The *Call* component and (b) its STG

$\triangle$(#[[0 : 2 | 1 : 2]]) = ((($0_r \uparrow$; $2_r \uparrow$; $2_a \uparrow$; $0_a \uparrow$) | ($1_r \uparrow$; $2_r \uparrow$; $2_a \uparrow$; $1_a \uparrow$)) ; (($0_r \downarrow$; $2_r \downarrow$; $2_a \downarrow$; $0_a \downarrow$) | ($1_r \downarrow$; $2_r \downarrow$; $2_a \downarrow$; $1_a \downarrow$))) *

$\triangledown$(#[[0 : 2 | 1 : 2]]) = ε

The original *Call* STG can be modified because it can be non-consistent. For example, if channel 1 is external (its transitions are not shared), after the rising transitions of channel 0 there can be the falling transitions of channel 1. The inconsistency can be solved by adding a place between the rising and falling transitions of the external channel 1, as figure A.9 shows. This place only enables the falling transitions of 1 if there have been its rising transitions.

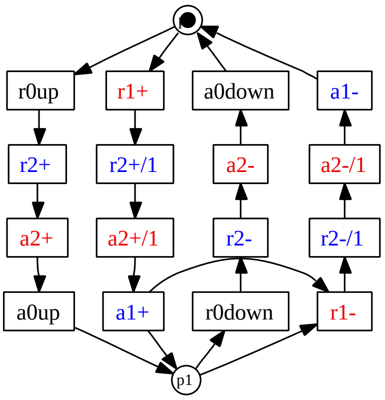
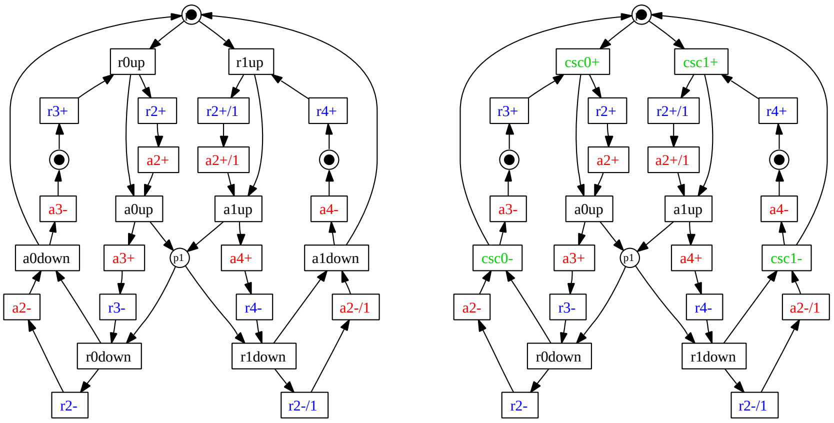


Figure A.9: The consistent *Call* STG

The original *Call* STG also can be modified because it can have irreducible CSC conflicts. Figure A.10(a) shows the STG of this component with internal channels 0 and 1 and other external channels 3 and 4 of other components enclosing channels 0 and 1. The firing sequences *r3+ r2+ a2+ r4+* and *r4+*

r2+ a2+ r3+ have the same encoding but can have different markings with different behaviors. For example, after the firing of the transition *r3+*, the transitions *r0up r2+ a2+ a0up* can fire and, after firing the transition *r4+*, only the transition *a3+* is enabled. In contrast, after the firing of the transition *r4+*, the transitions *r1up r2+ a2+ a1up* can fire and, after firing the transition *r3+*, only the transition *a4+* is enabled. Synthesis tools can not resolve this CSC conflicts by adding internal signals. Figure A.10(b) shows the previous STG with internal signals *csc0* and *csc1* replacing the dummy transitions *r0up*, *a0down*, *r1up* and *a1down*. The firing sequences *r3+ r2+ a2+ r4+* and *r4+ r2+ a2+ r3+*, after the firing sequences *csc0+ r2+ a2+ a0up* or *csc1+ r2+ a2+ a1up* have different encoding. Therefore, the CSC conflicts are solved.



(a) (b)

Figure A.10: (a) The STG with irreducible CSC conflicts and (b) the STG without CSC conflicts of the *DecisionWait* STG

The original *Call* STG also can be modified to compose STGs by sharing transitions. For example, if channel 2 is internal, its transitions must be shared. But they can not be shared because they are not unique and are indexed. It can be solved by replacing indexed transitions with *pre* and *post* dummy transitions, unique shared transitions and places between them, as figure A.11 shows.

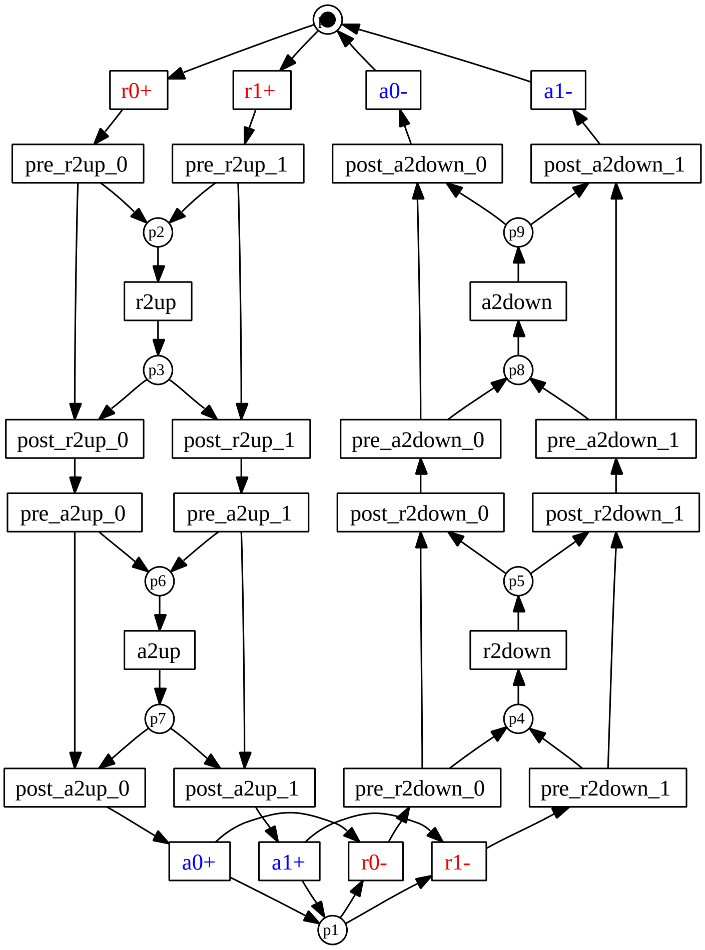


Figure A.11: The *Call* STG with unique shared transitions

This solution adds a lot of dummy transitions and places. Other solution is that rising and falling transitions of channels 0 and 1 share unique rising and falling transitions of channel 2, as figure A.12 shows.

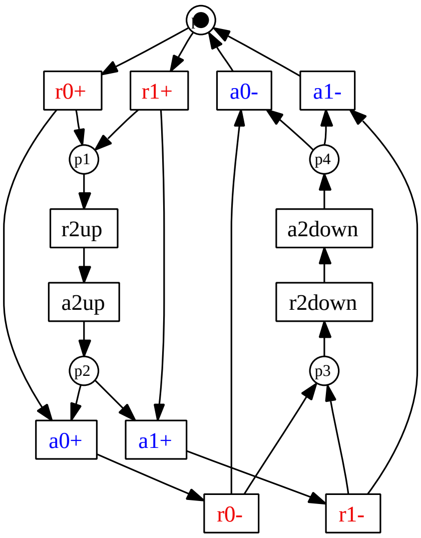


Figure A.12: The efficient *Call* STG with unique shared transitions

Appendix B

HC Nets and STGs of Balsa Examples

This appendix shows for each Balsa example: the HC nets, clustered HC nets, cluster HC nets and cluster STGs resulting from the new design flow.

B.1 Arbiter Tree

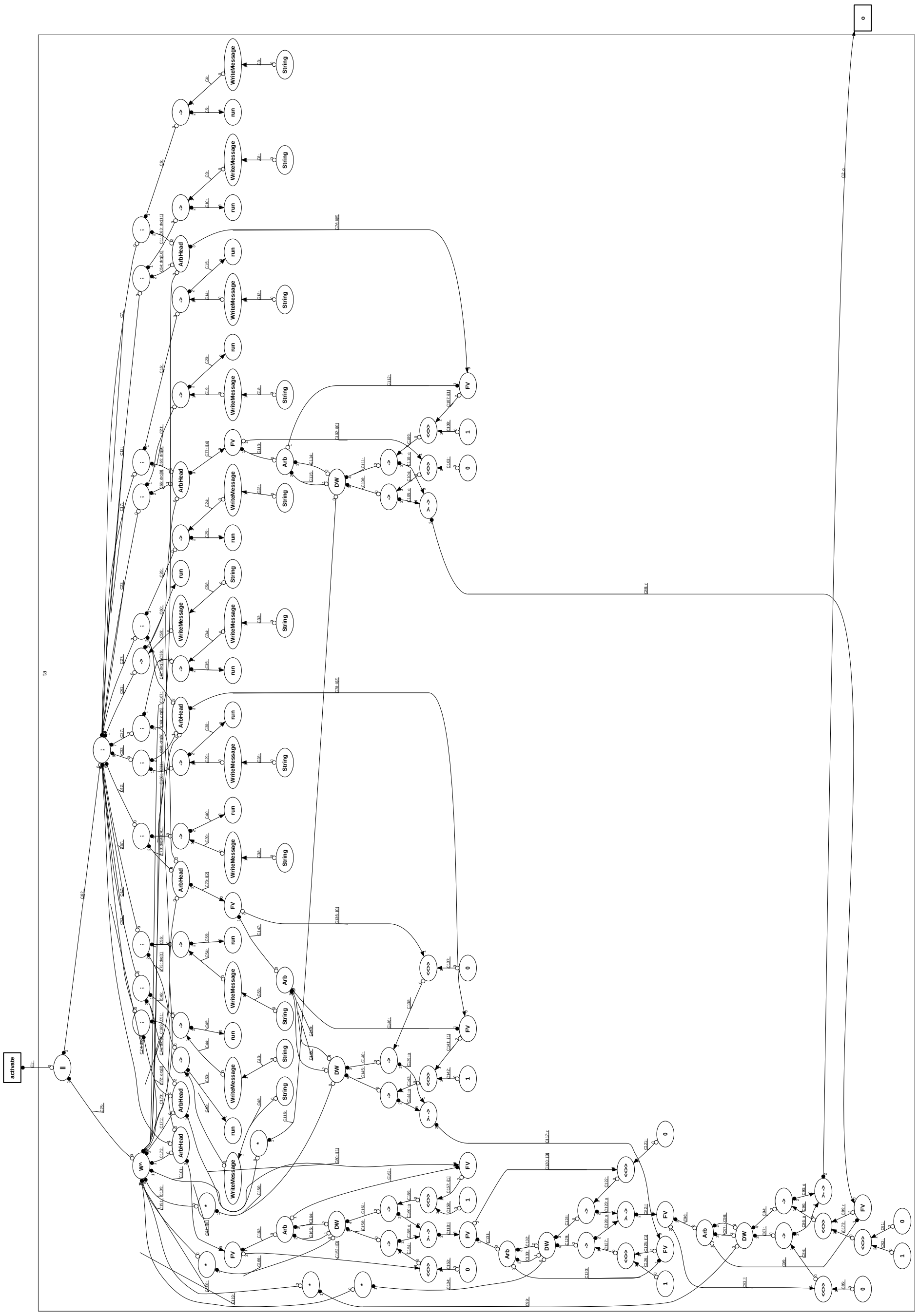


Figure B.1: The HC netlist of the arbiter tree

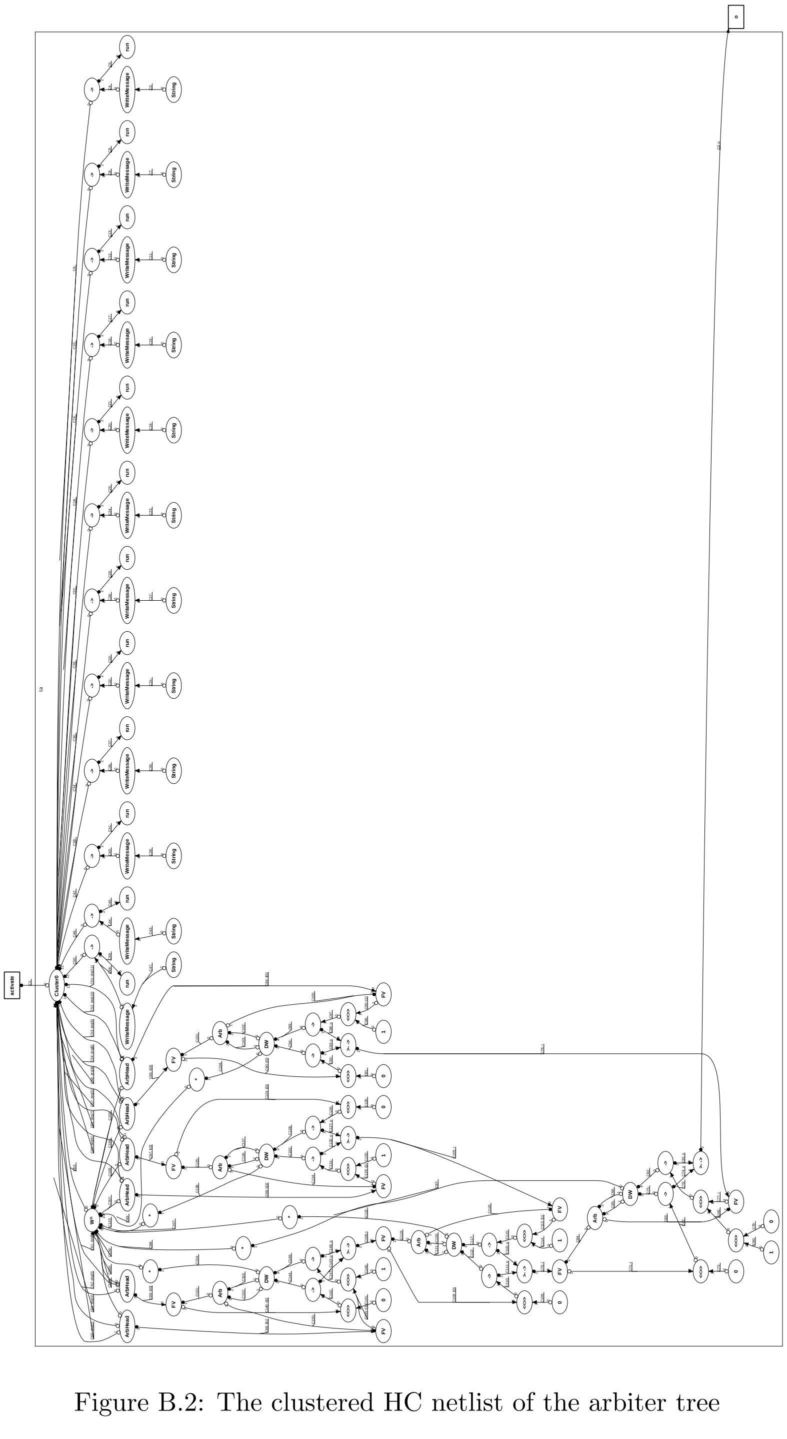


Figure B.2: The clustered HC netlist of the arbiter tree

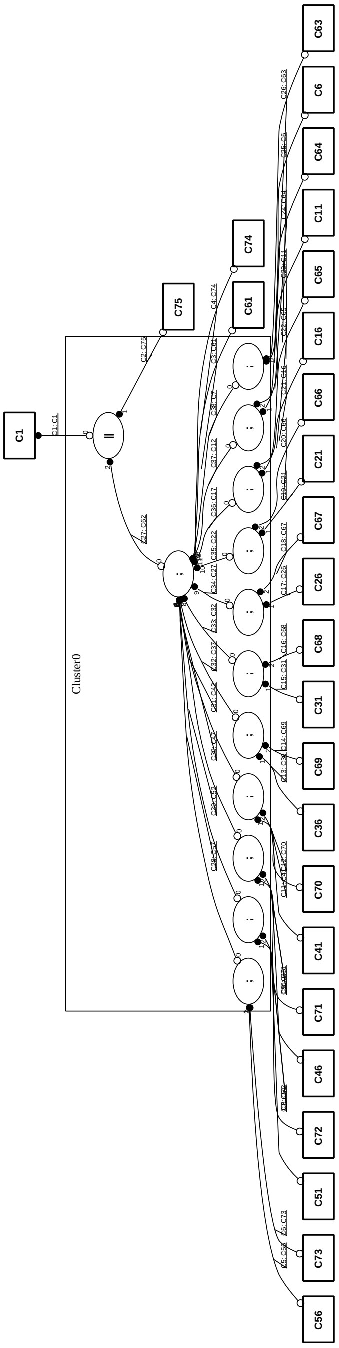


Figure B.3: The cluster 0 HC netlist of the arbiter tree

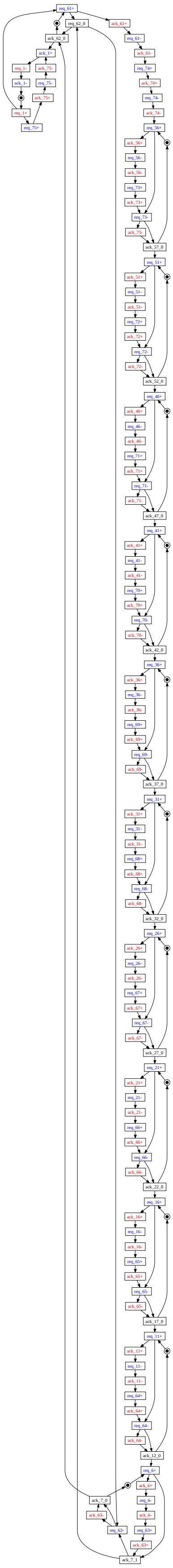


Figure B.4: The cluster 0 STG of the arbiter tree

B.2 Population Counter

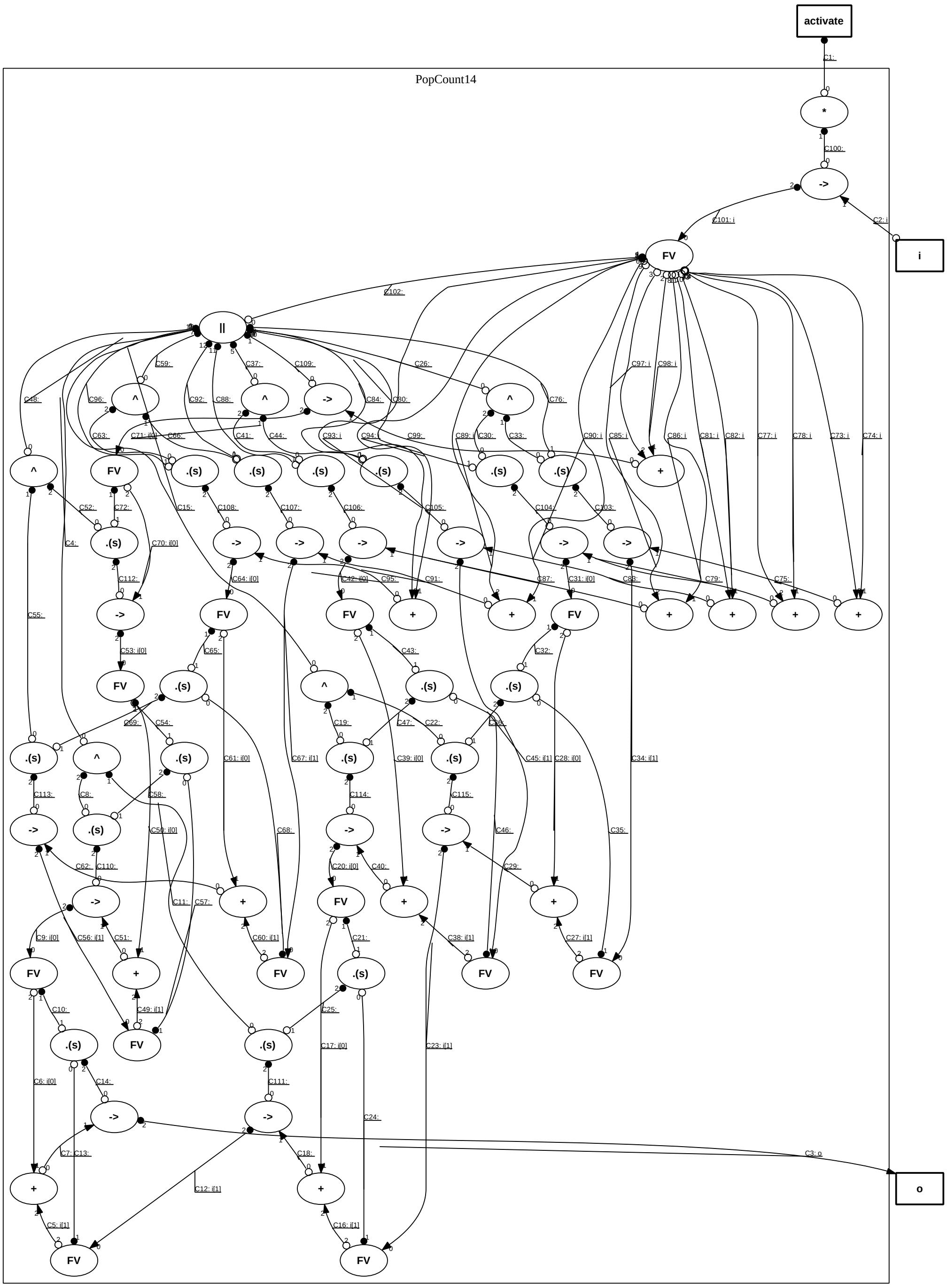


Figure B.5: The HC netlist of the population counter first part definition

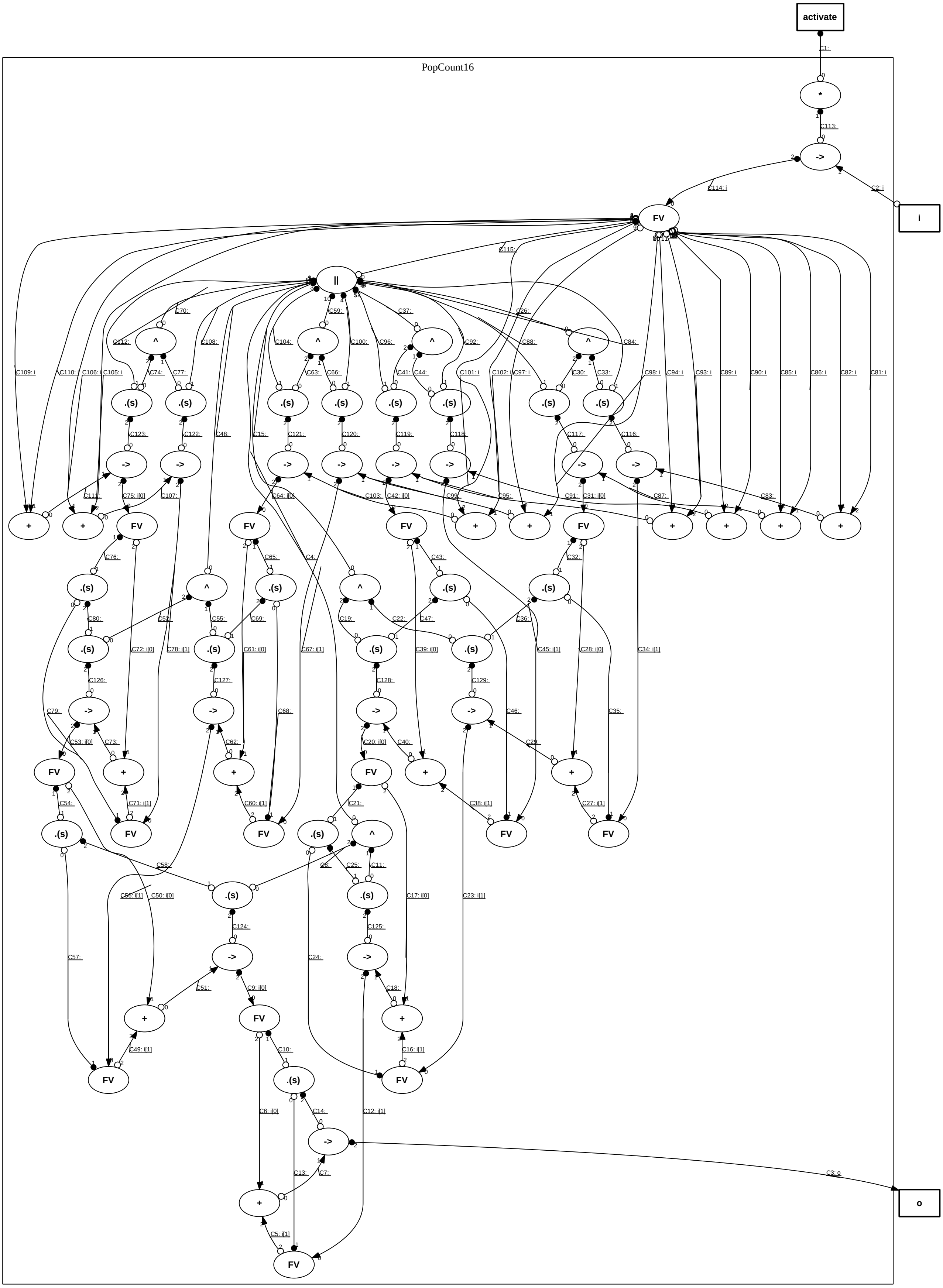


Figure B.6: The HC netlist of the population counter second part definition

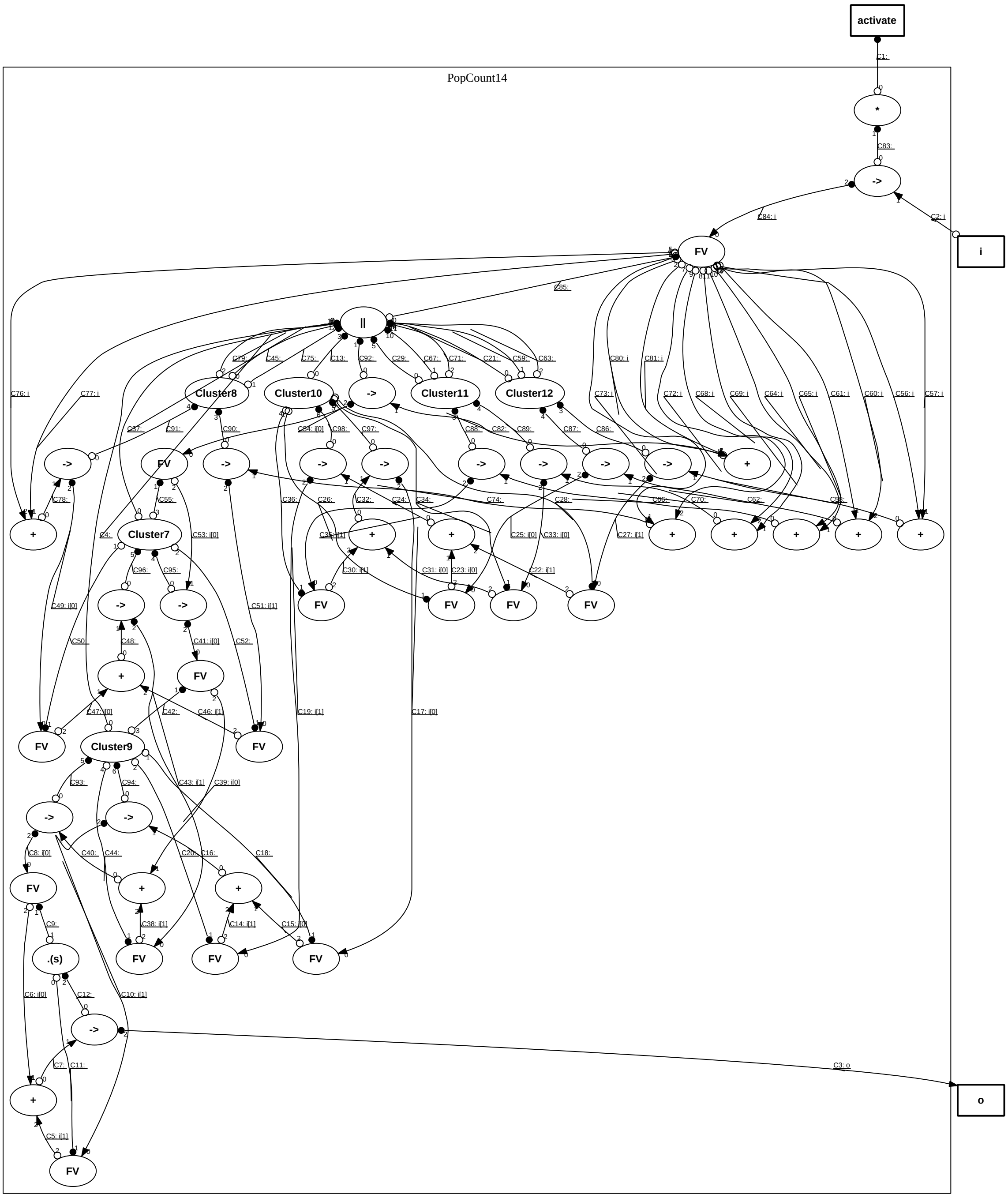


Figure B.7: The clustered HC netlist of the population counter first part definition

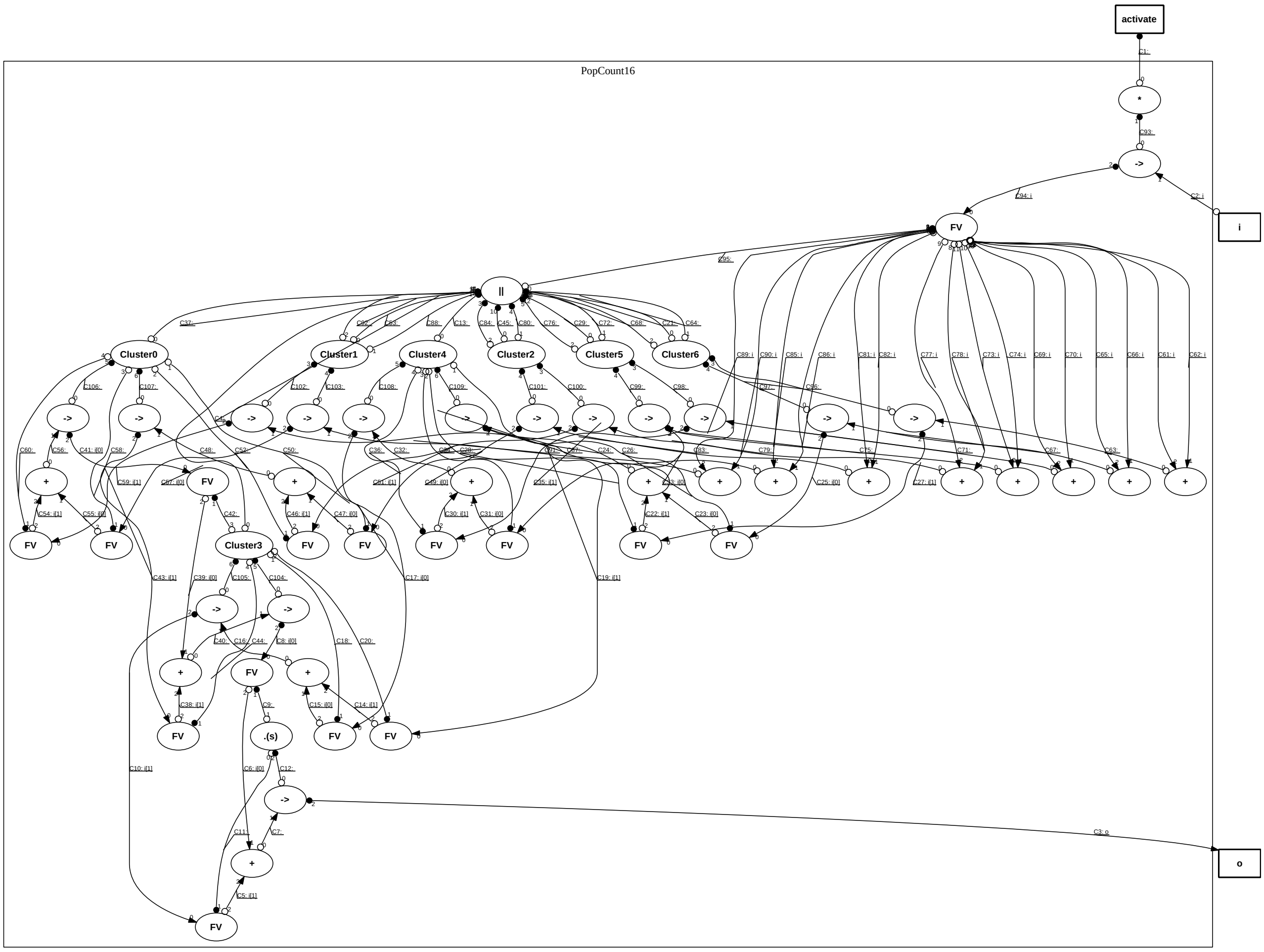


Figure B.8: The clustered HC netlist of the population counter second part definition

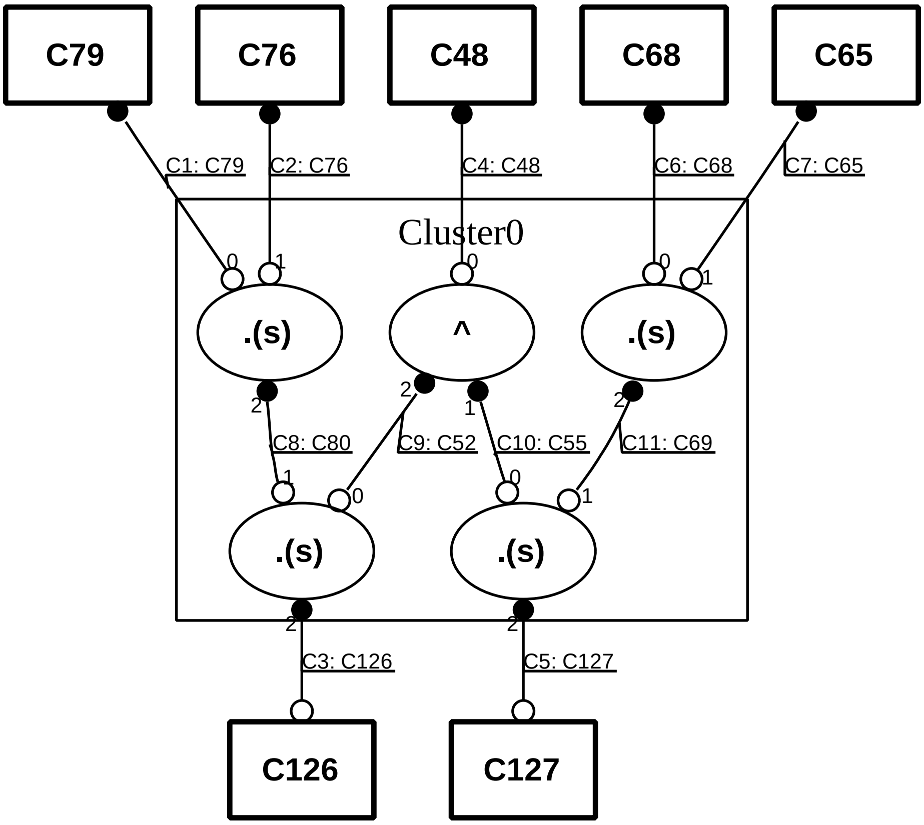


Figure B.9: The cluster 0 HC netlist of the population counter

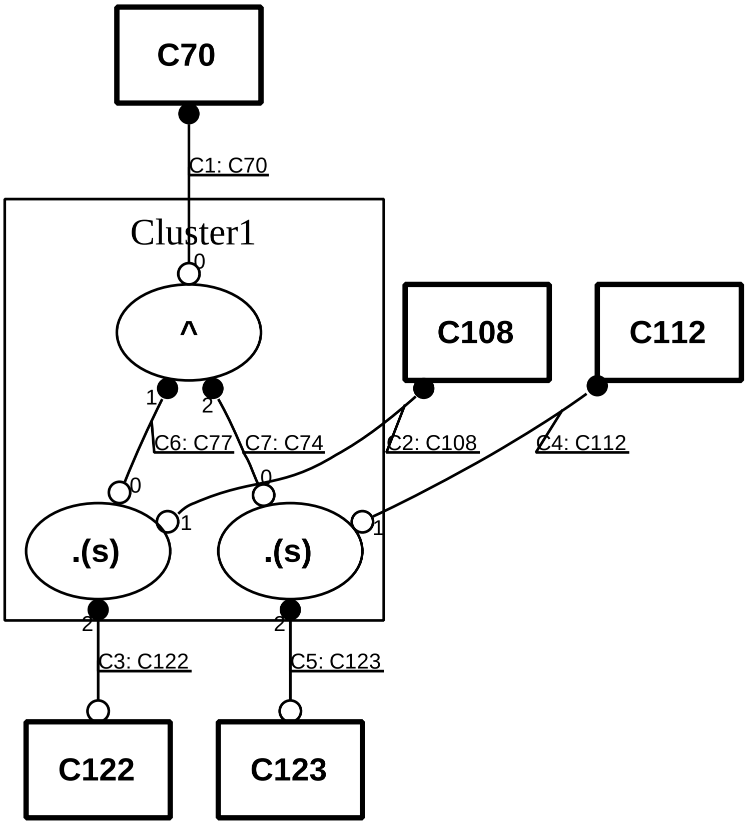


Figure B.10: The cluster 1 HC netlist of the population counter

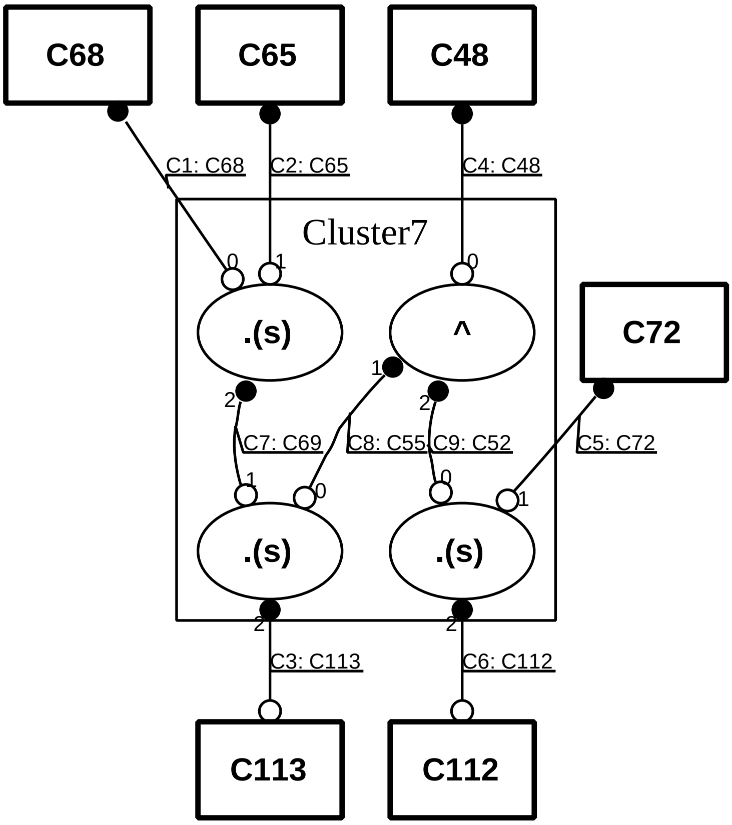


Figure B.11: The cluster 7 HC netlist of the population counter

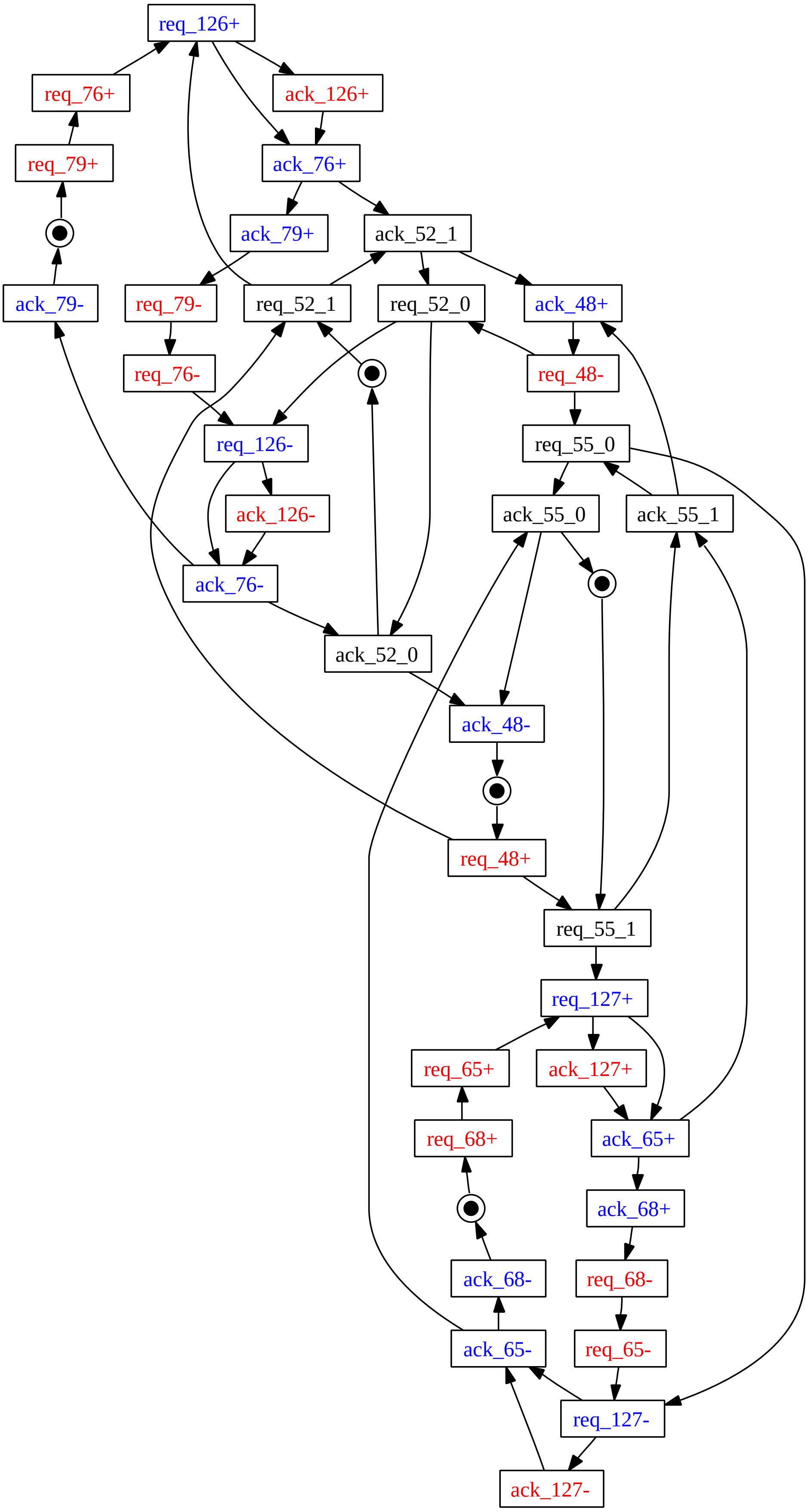


Figure B.12: The cluster 0 STG of the population counter

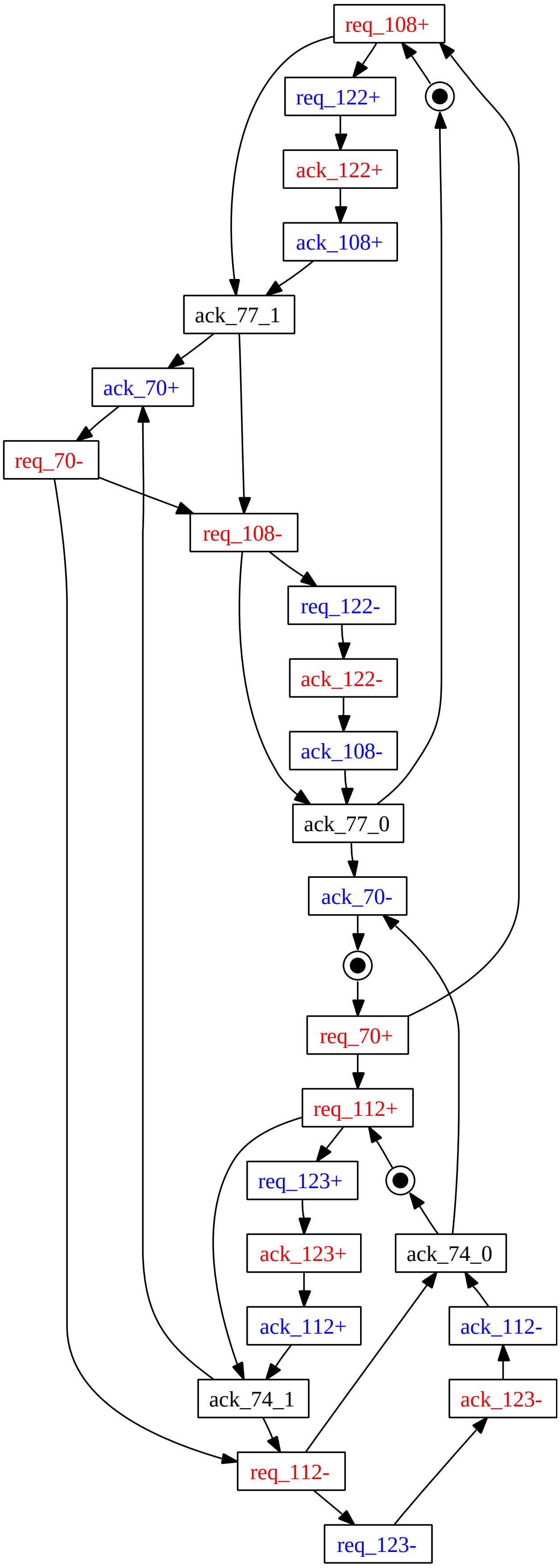


Figure B.13: The cluster 1 STG of the population counter

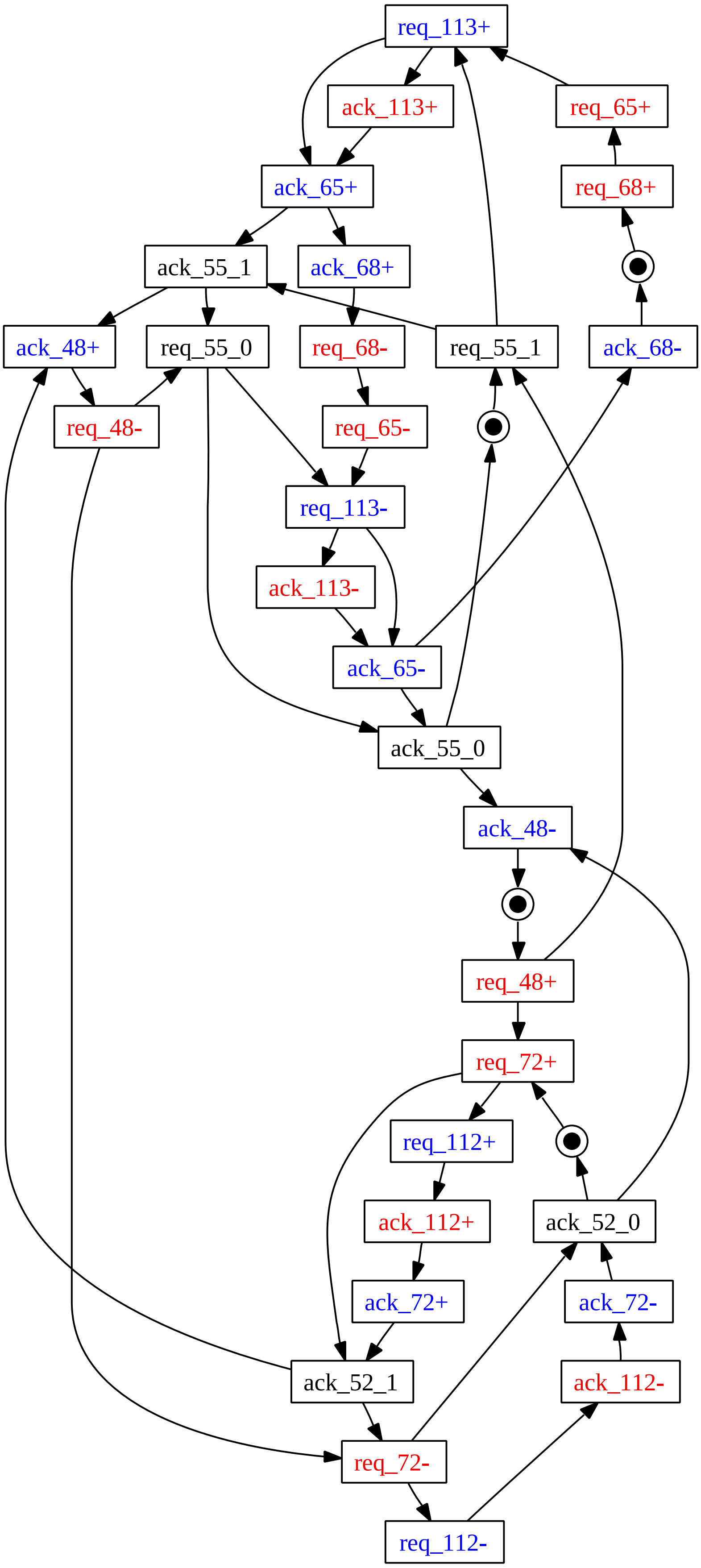


Figure B.14: The cluster 7 STG of the population counter

B.3 Shifter

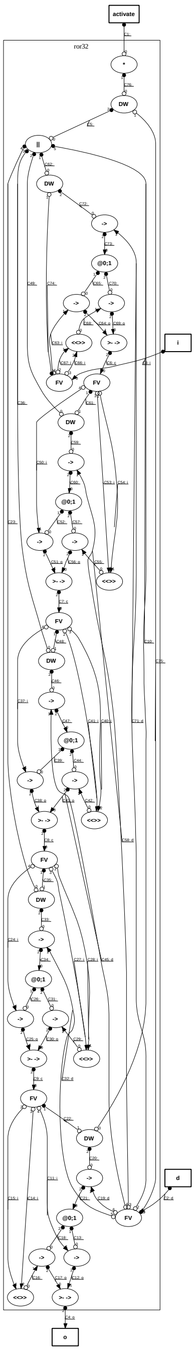


Figure B.15: The HC netlist of the shifter

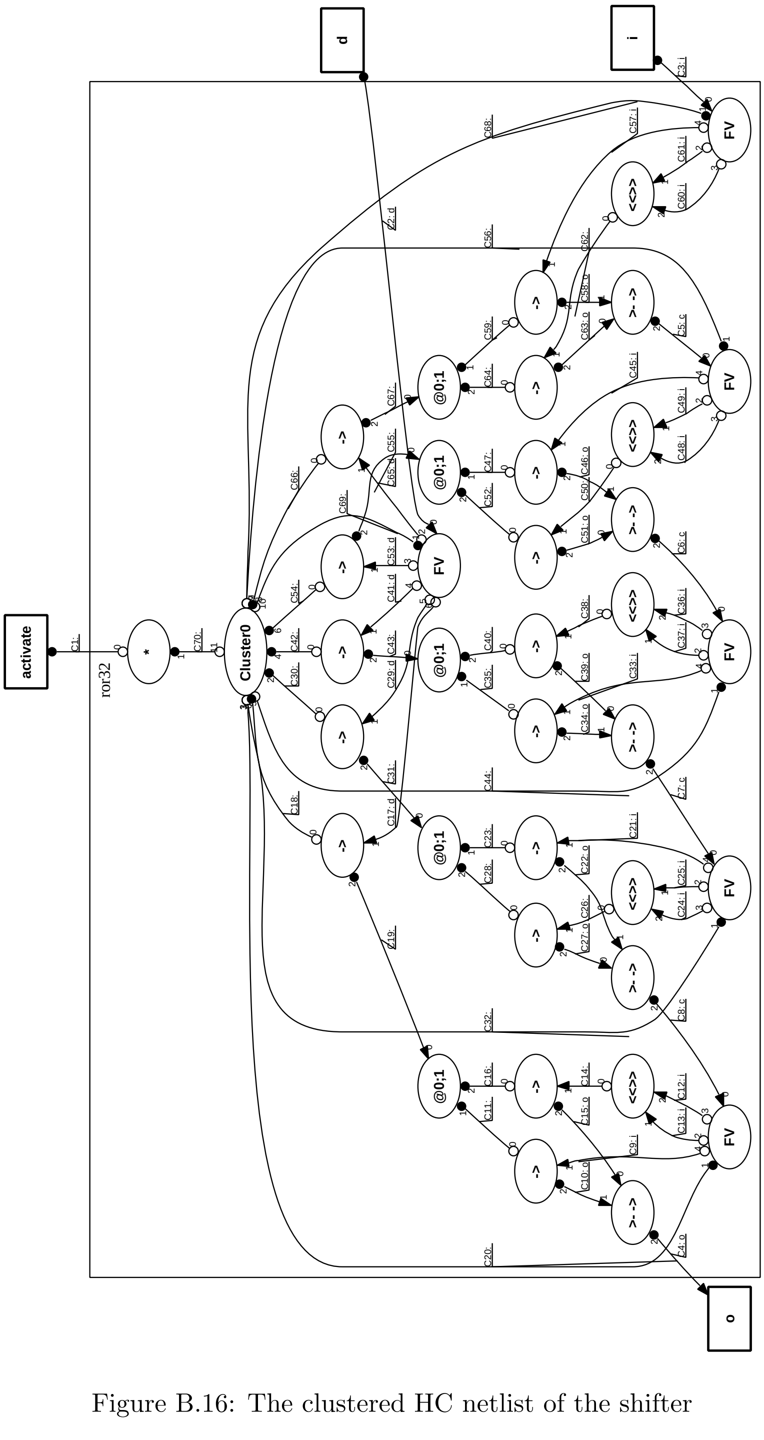


Figure B.16: The clustered HC netlist of the shifter

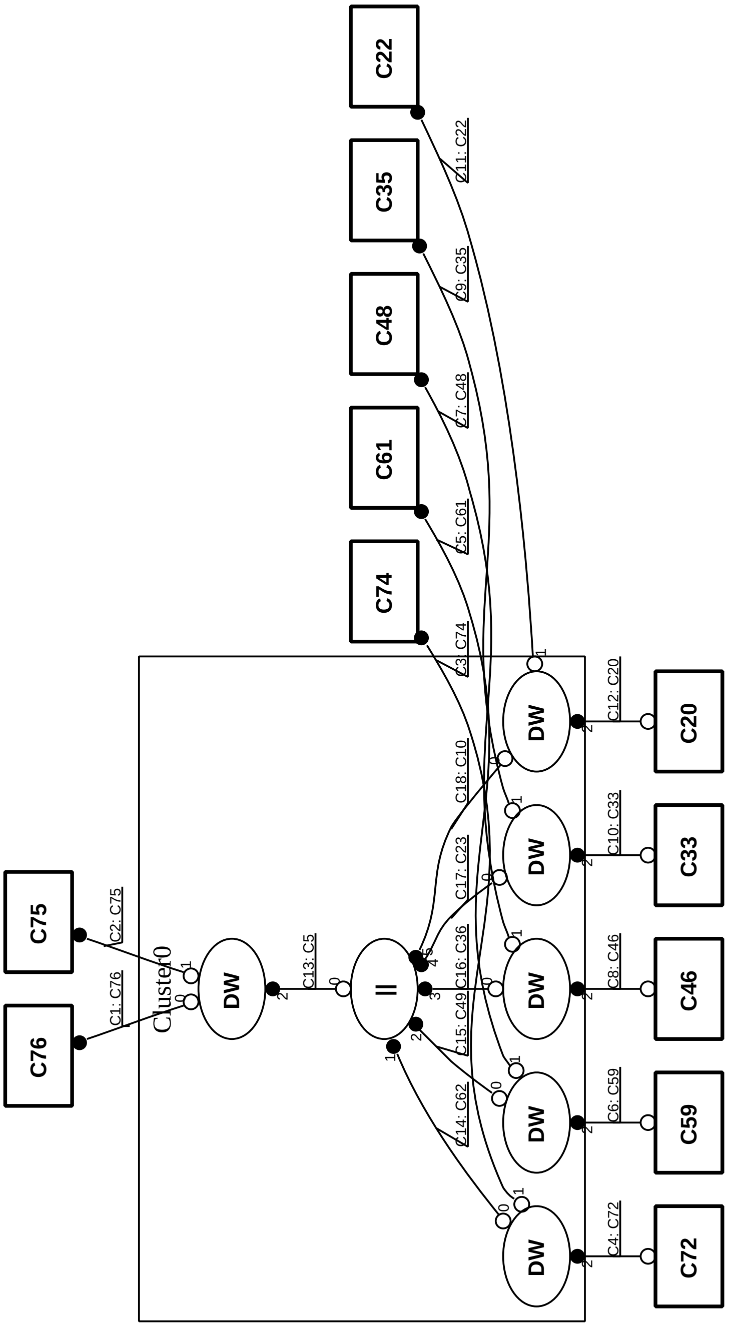


Figure B.17: The cluster 0 HC netlist of the shifter

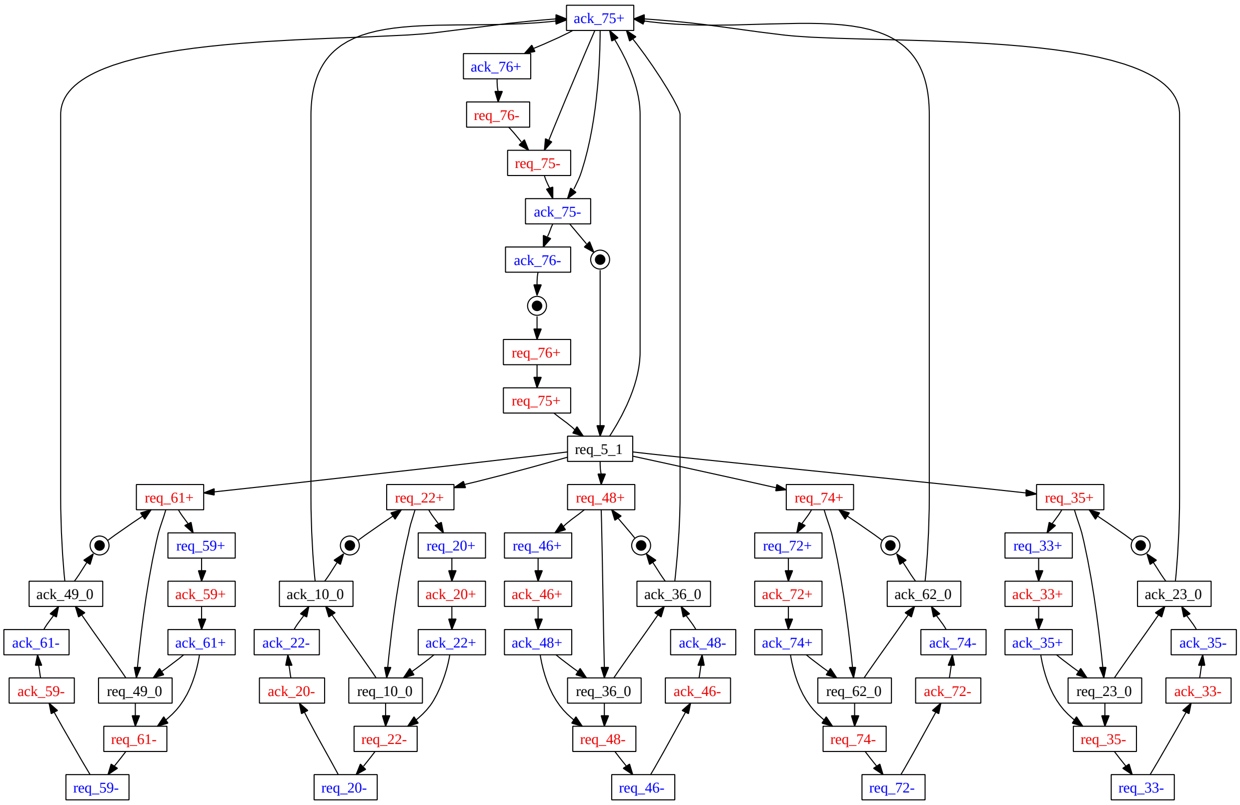


Figure B.18: The cluster 0 STG of the shifter

B.4 SSEM

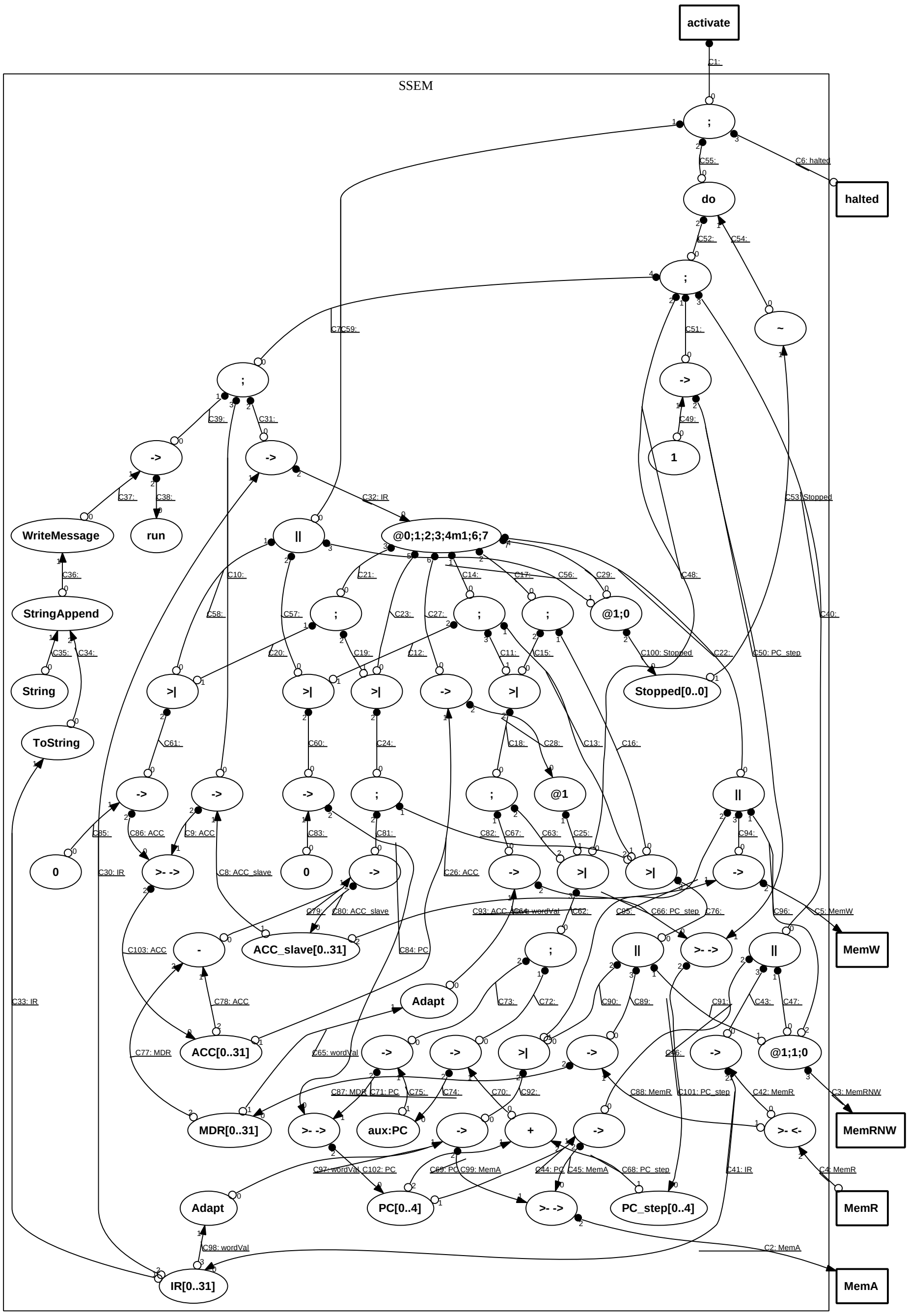


Figure B.19: The HC netlist of the SSEM

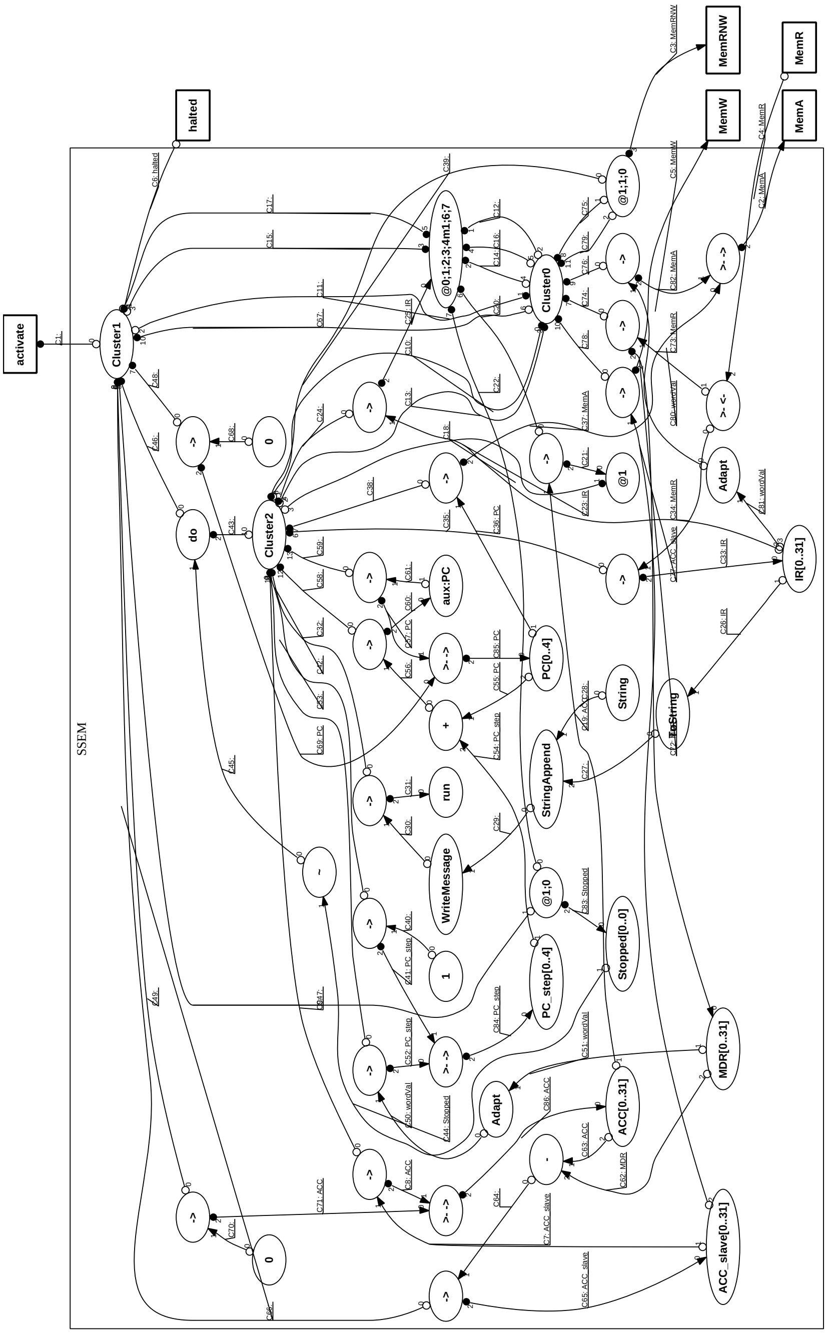


Figure B.20: The clustered HC netlist of the SSEM

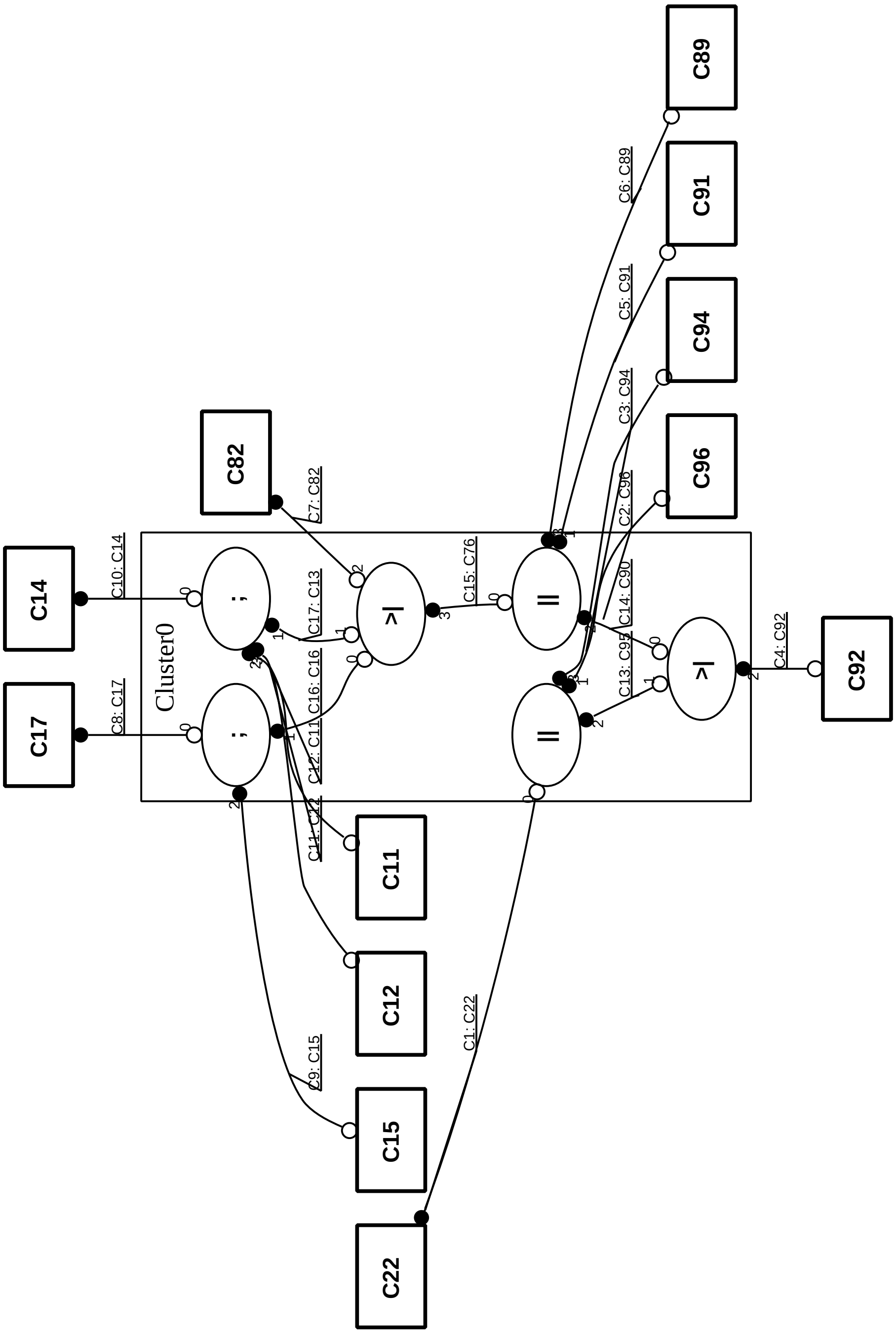


Figure B.21: The cluster 0 HC netlist of the SSEM

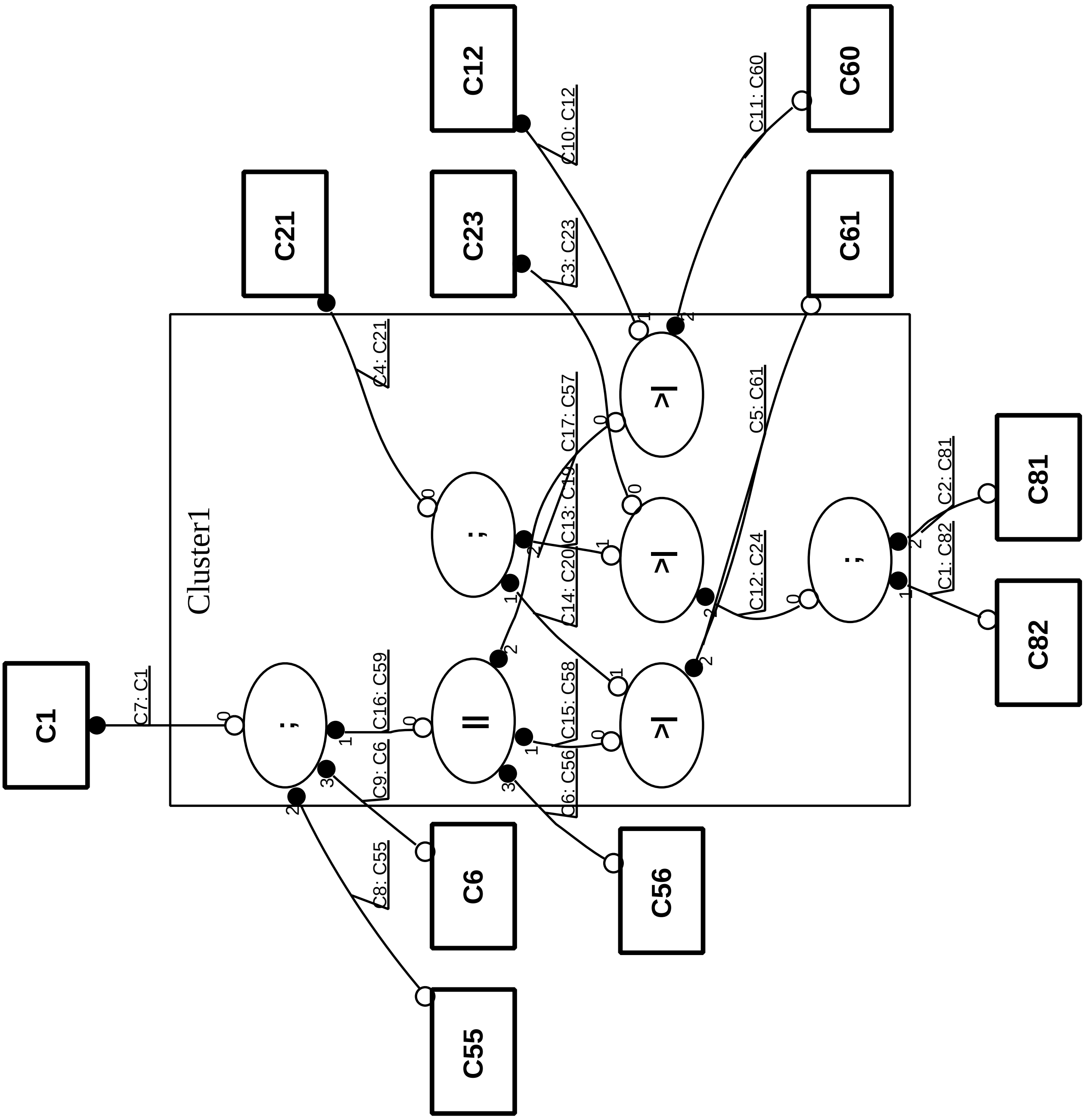


Figure B.22: The cluster 1 HC netlist of the SSEM

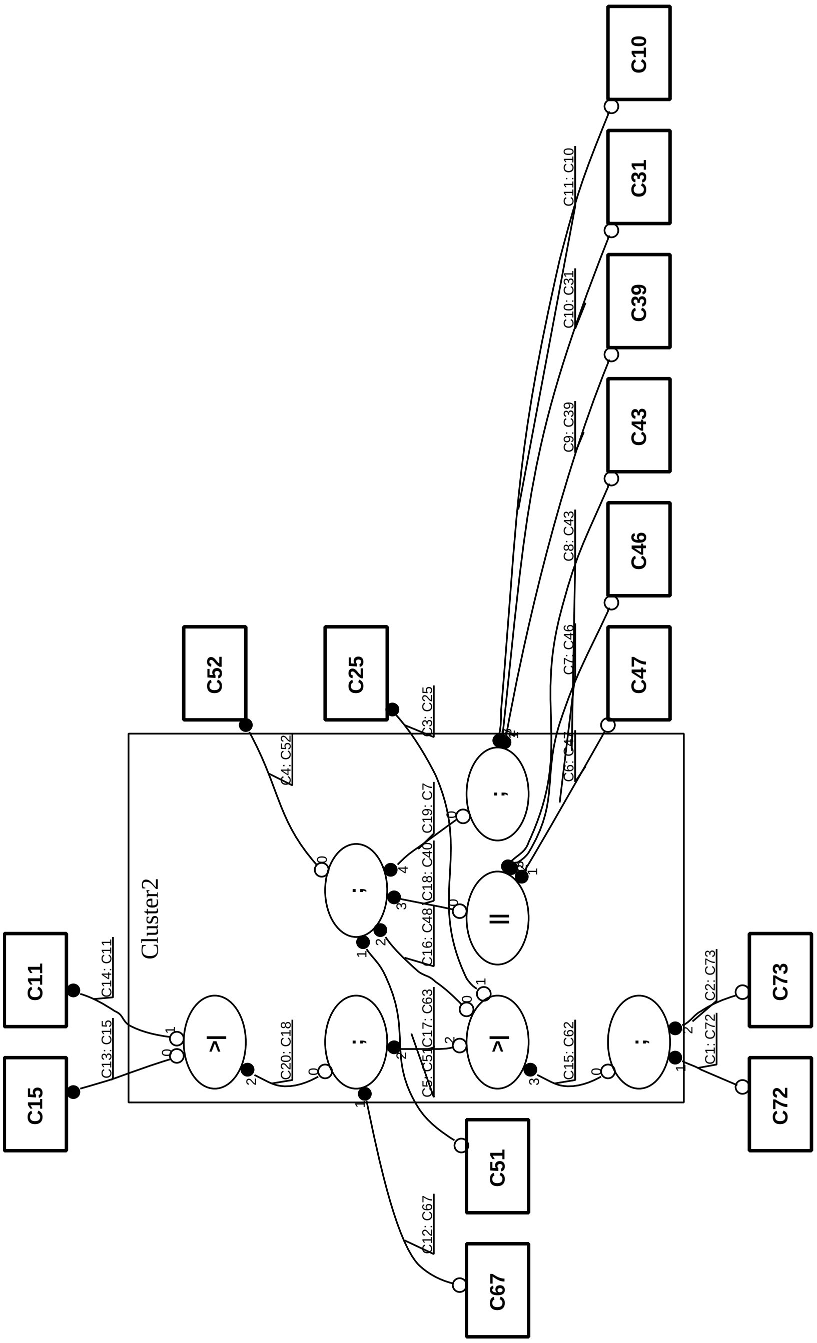


Figure B.23: The cluster 2 HC netlist of the SSEM

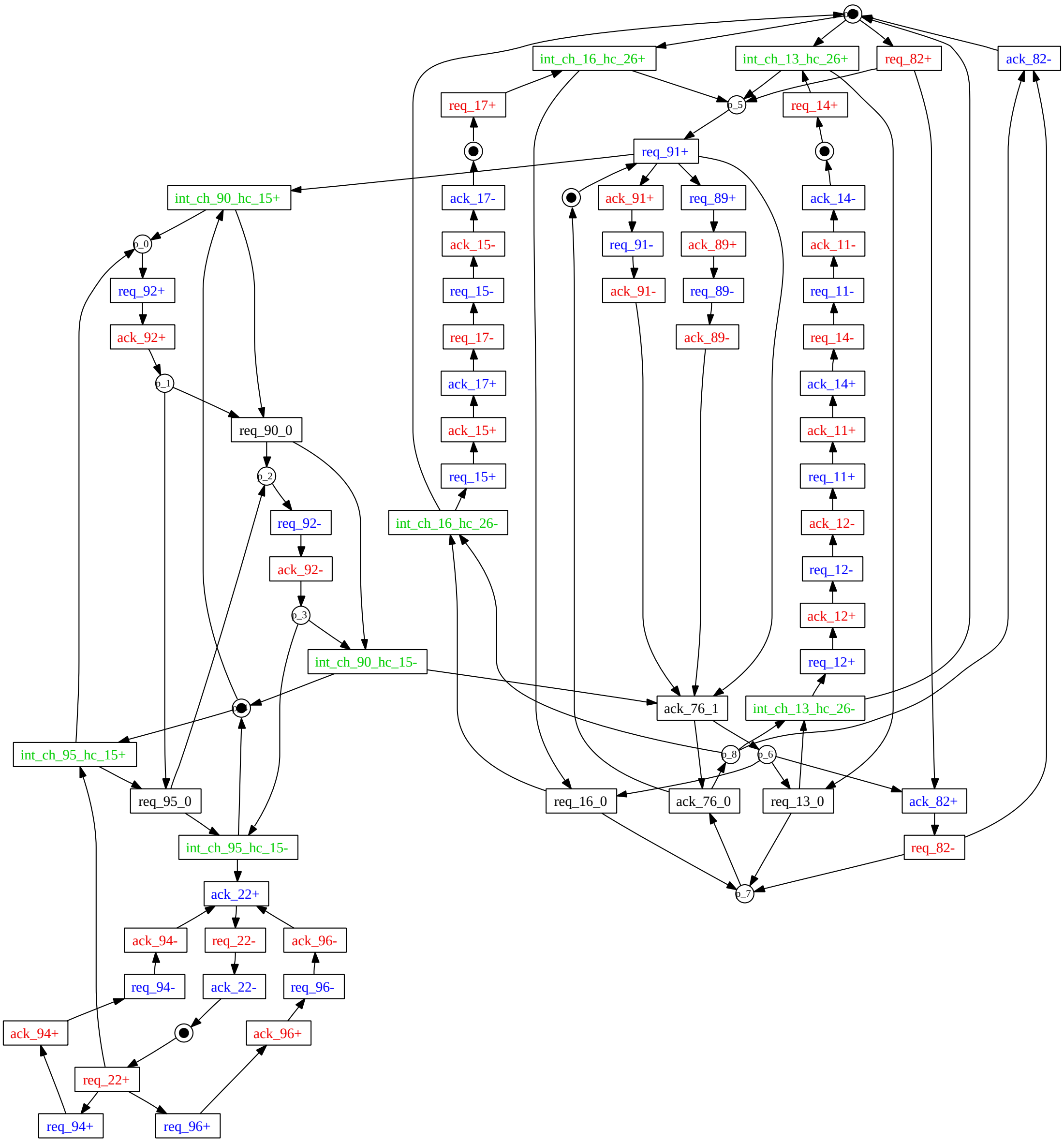


Figure B.24: The cluster 0 STG of the SSEM

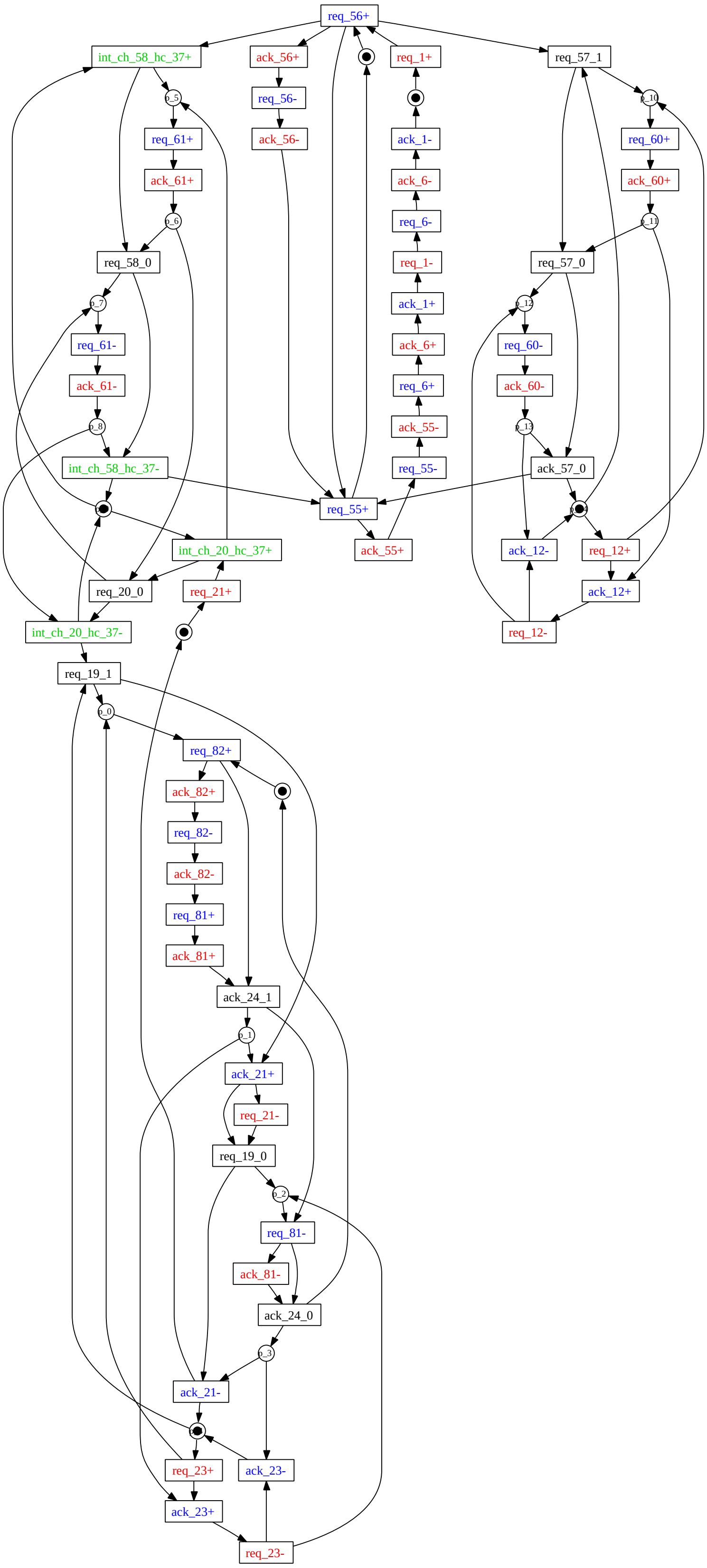


Figure B.25: The cluster 1 STG of the SSEM

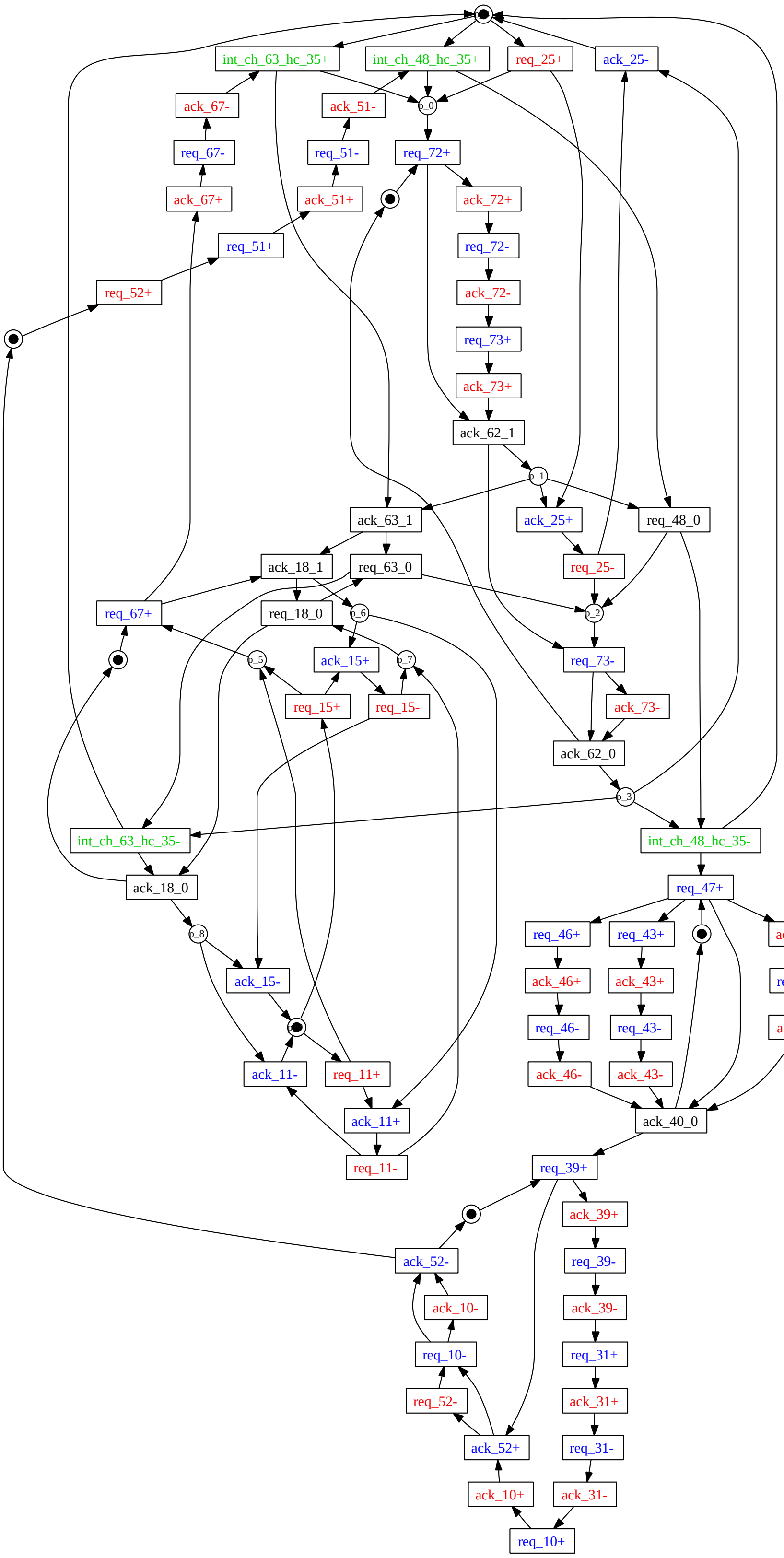


Figure B.26: The cluster 2 STG of the SSEM

B.5 Stack

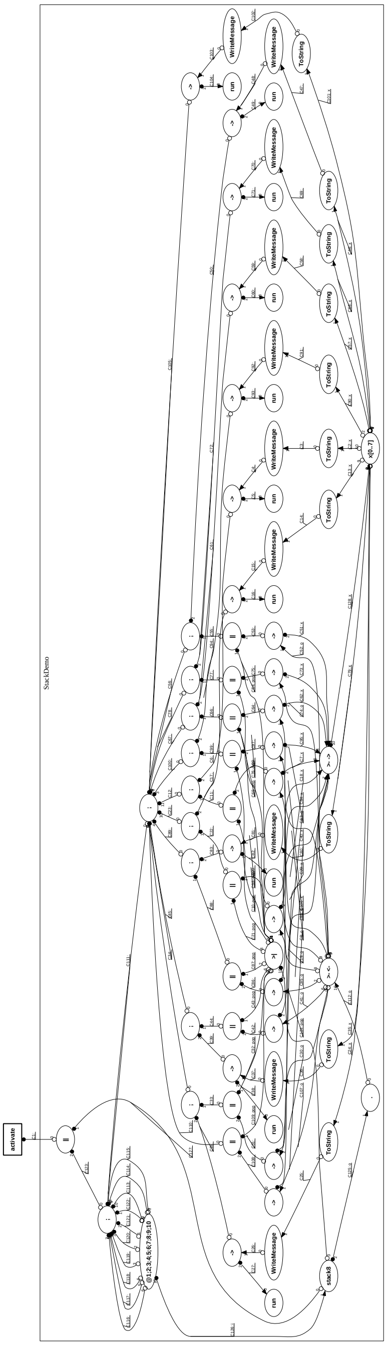


Figure B.27: The HC netlist of the stack first part definition

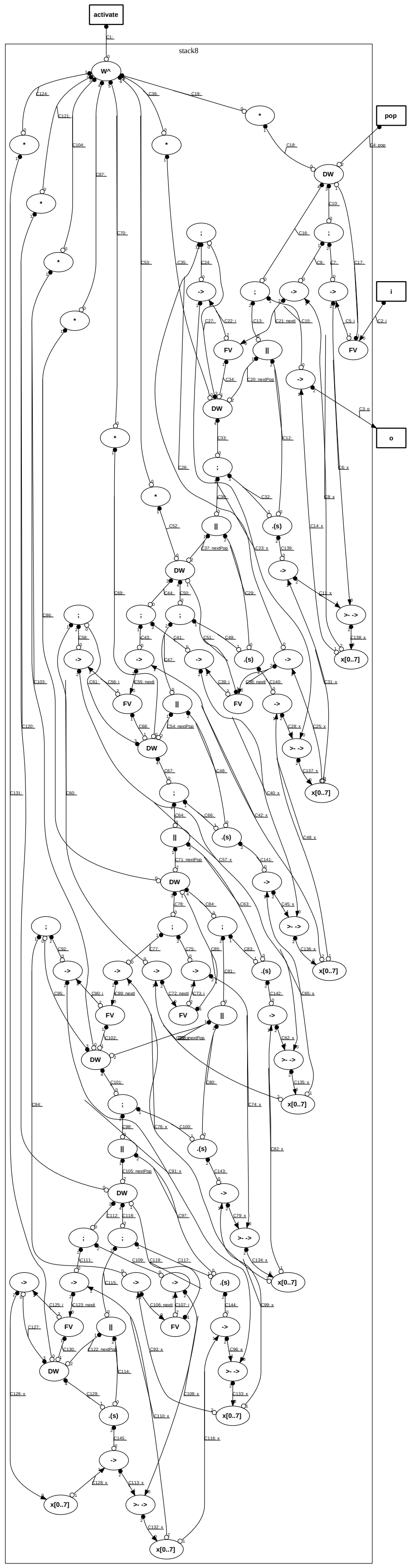


Figure B.28: The HC netlist of the stack second part definition

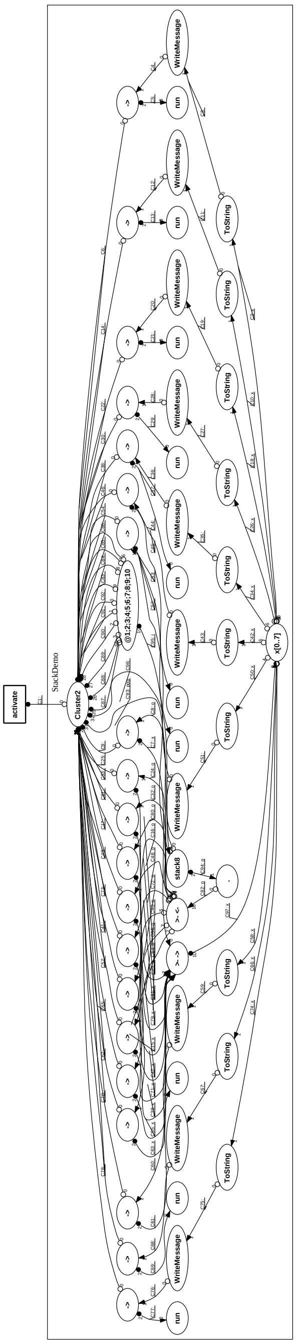


Figure B.29: The clustered HC netlist of the stack first part definition

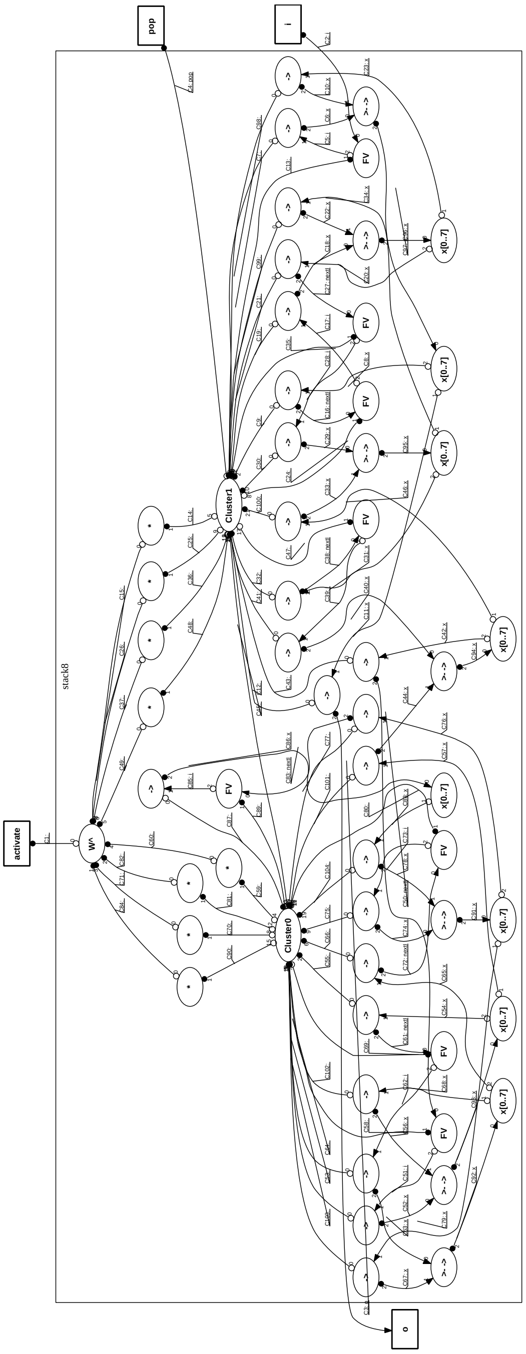


Figure B.30: The clustered HC netlist of the stack second part definition

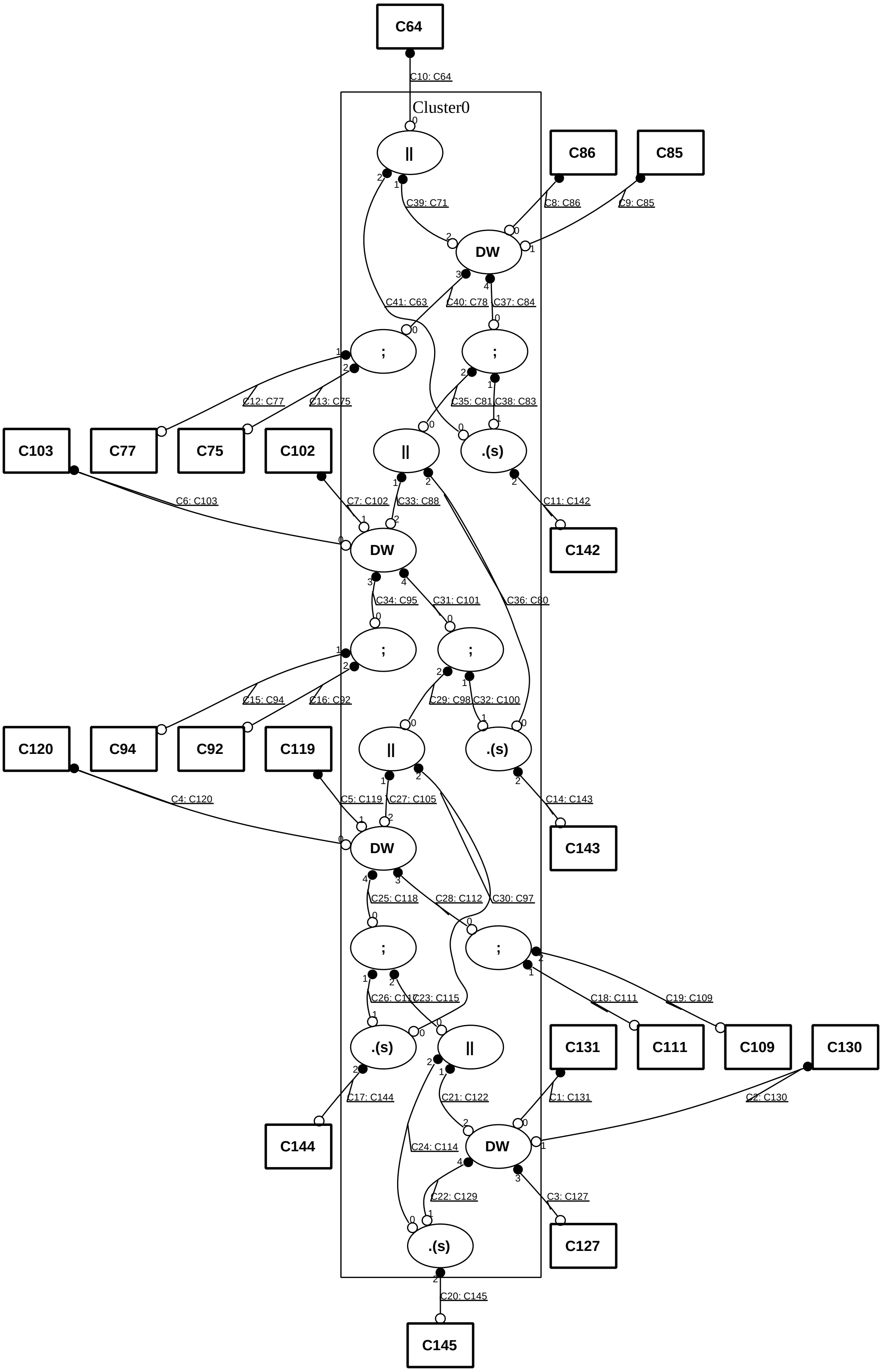


Figure B.31: The cluster 0 HC netlist of the stack

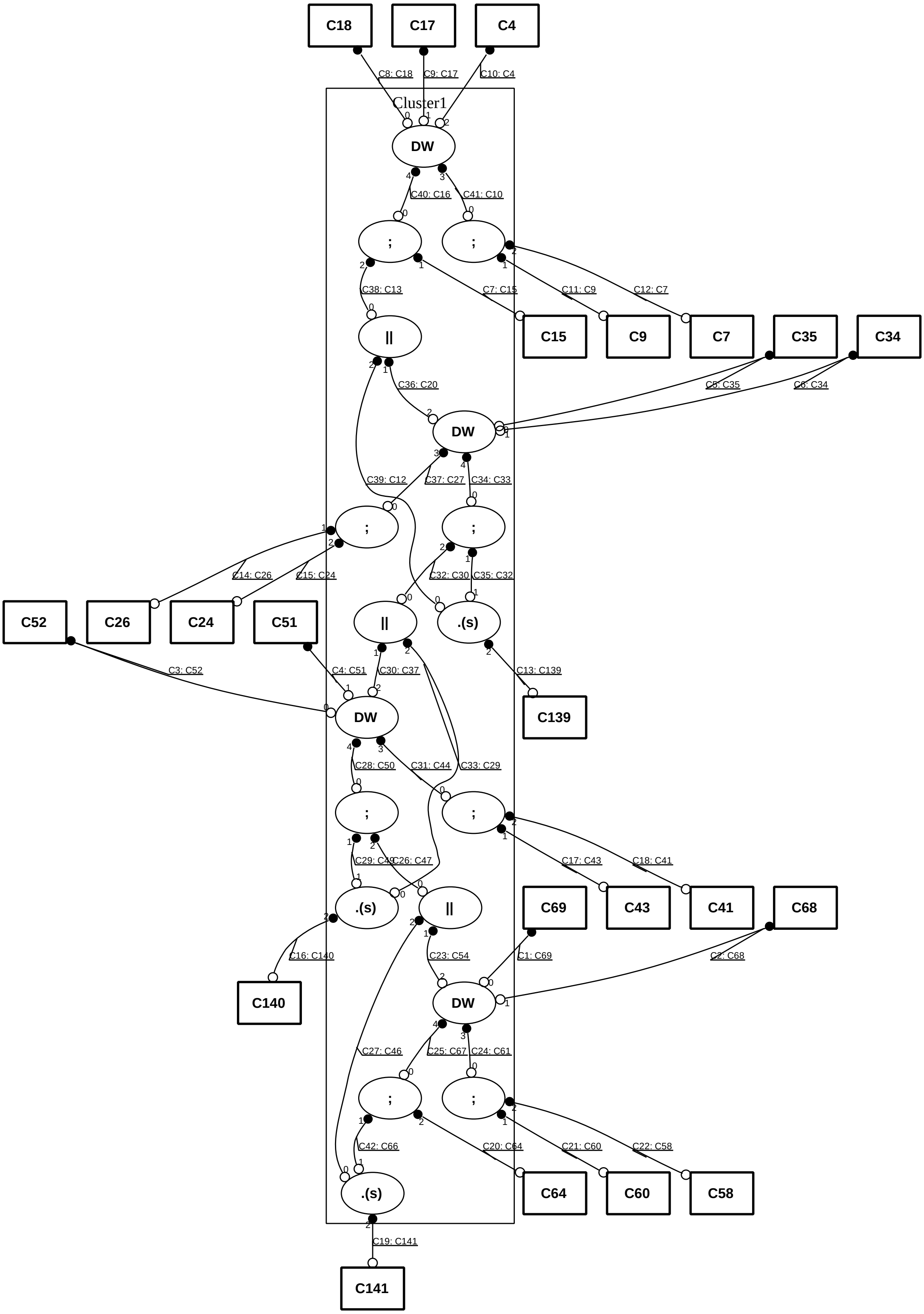


Figure B.32: The cluster 1 HC netlist of the stack

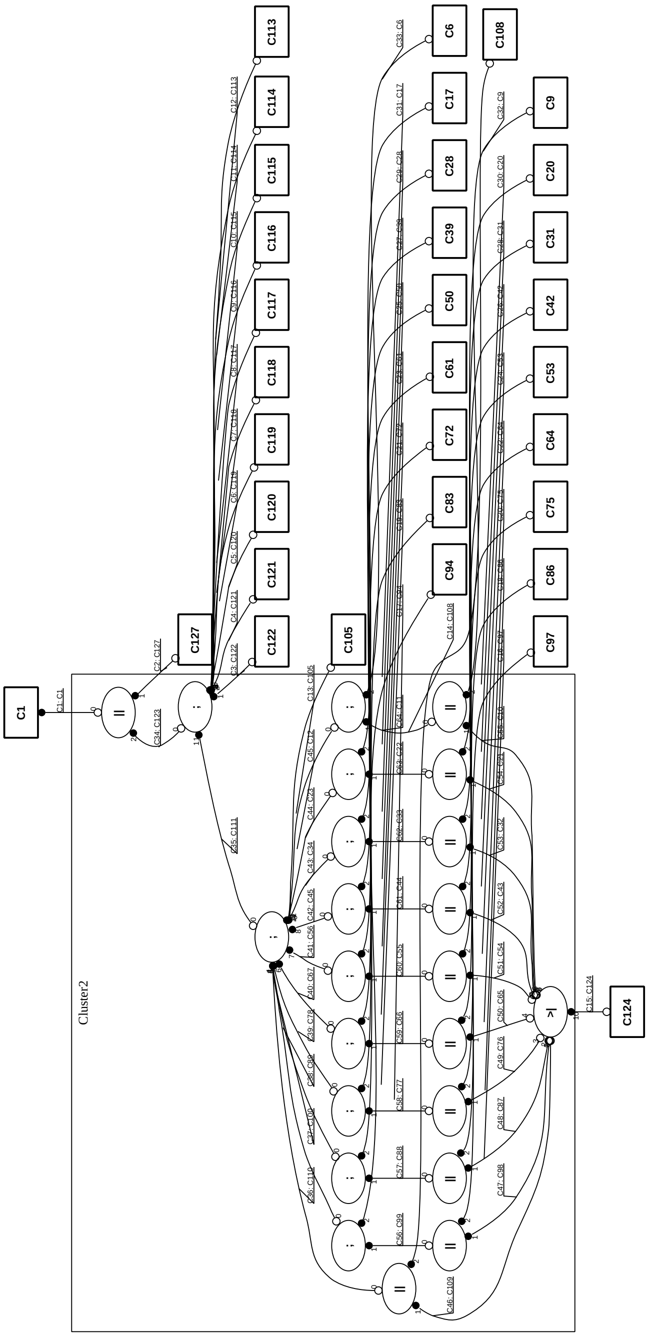


Figure B.33: The cluster 2 HC netlist of the stack

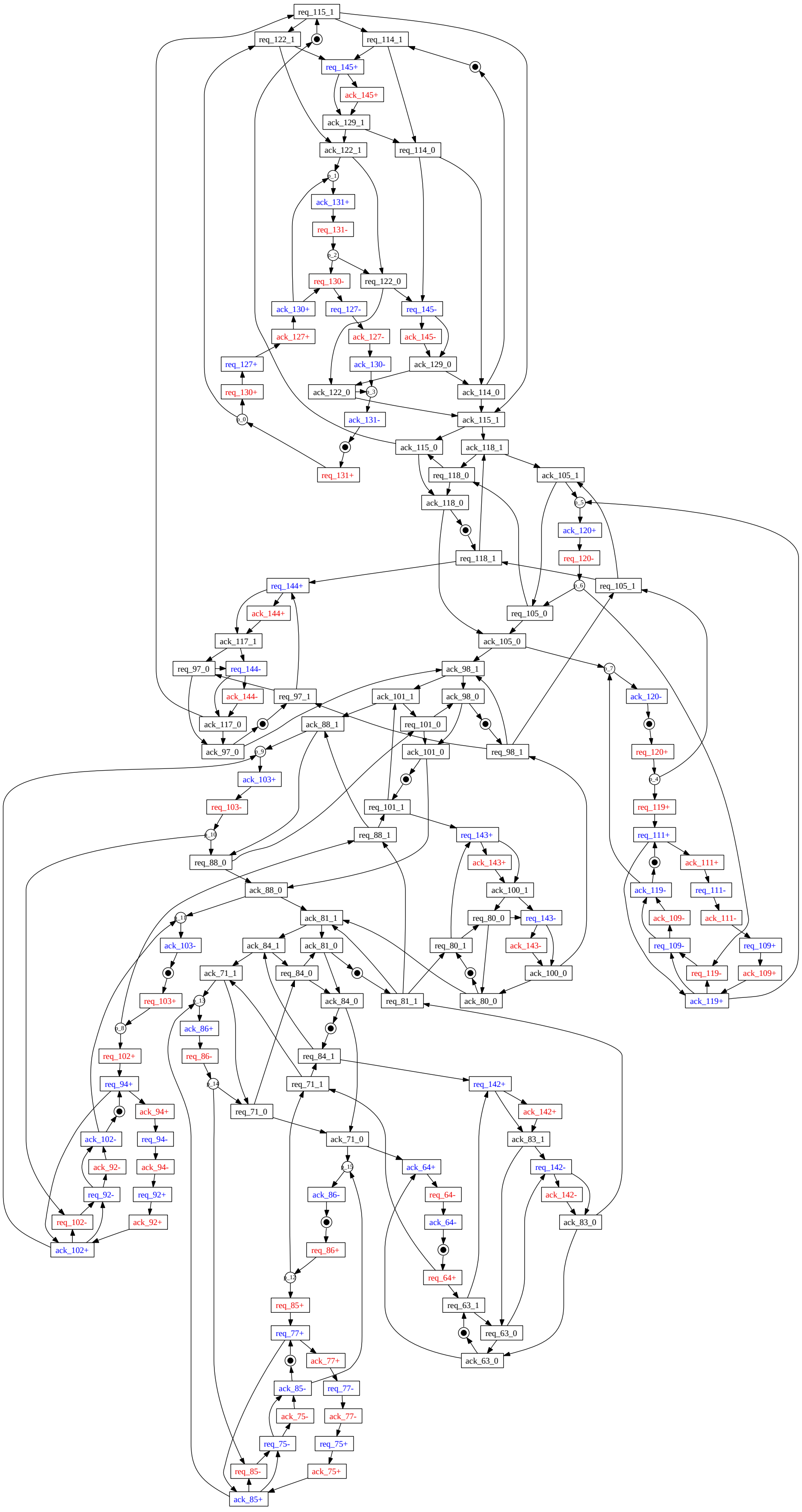


Figure B.34: The cluster 0 STG of the stack

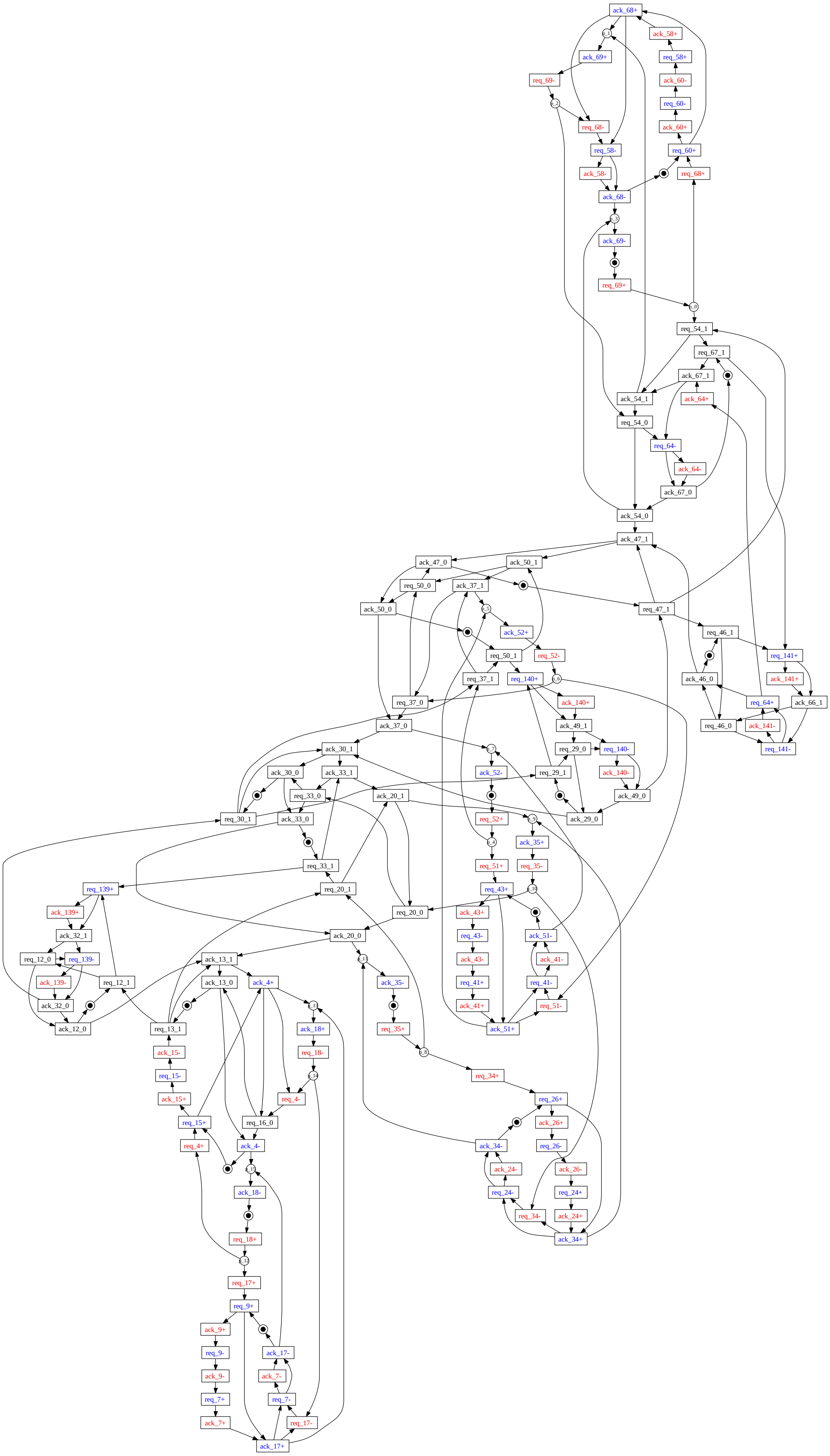


Figure B.35: The cluster 1 STG of the stack

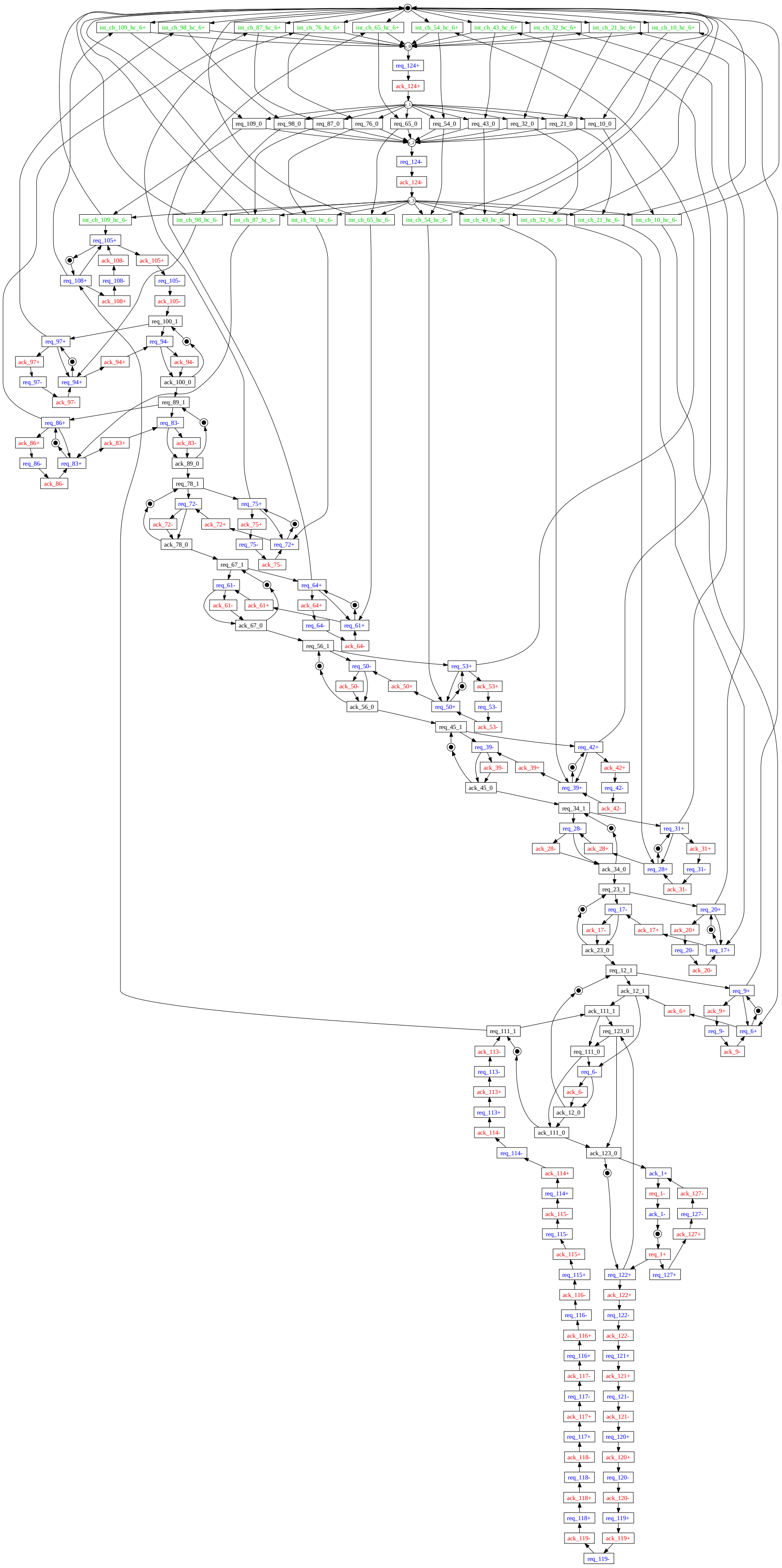


Figure B.36: The cluster 2 STG of the stack

Bibliography

[1] A. Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, 2000.

[2] J. Carmona and J. Cortadella. State encoding of large asynchronous controllers. In *Proc. ACM/IEEE Design Automation Conference*, pages 939–944, July 2006.

[3] Josep Carmona, José M. Colom, Jordi Cortadella, and Fernando García-Vallés. Synthesis of asynchronous controllers using integer linear programming. *IEEE Transactions on Computer-Aided Design*, 25(9):1637–1651, September 2006.

[4] T. Chelcea, S. Nowick, A. Bardsley, and D. Edwards. A burst-mode oriented back-end for the balsa synthesis system. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 330, Washington, DC, USA, 2002. IEEE Computer Society.

[5] J. Cortadella, M. Kishinevsky, A.Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[6] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Advanced Microelectronics. Springer-Verlag, 2002.

[7] Doug Edwards and Andrew Bardsley. Balsa: An Asynchronous Hardware Synthesis Language. *The Computer Journal*, 45(1):12–18, 2002.

[8] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, 1999.

[9] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state coding conflicts in stgs using integer programming. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 338, Washington, DC, USA, 2002. IEEE Computer Society.

[10] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Detecting state coding conflicts in stg unfoldings using sat. In *ACSD '03: Proceedings of the Third International Conference on Application of Concurrency to System Design*, page 51, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Tilman Kolks, Steven Vercauteren, and Bill Lin. Control resynthesis for control-dominated asynchronous designs. In *ASYNC '96: Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 233, Washington, DC, USA, 1996. IEEE Computer Society.

[12] Bill Lin, Chantal Ykman-Couvreur, and Peter Vanbekbergen. A general state graph transformation framework for asynchronous synthesis. In *EURO-DAC '94: Proceedings of the conference on European design automation*, pages 448–453, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[13] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–574, April 1989.

[14] M. A. Peña and J. Cortadella. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[15] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The vlsi-programming language tangram and its translation into handshake circuits. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 384–389, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

List of Figures

2.1 A marking (a) before and (b) after a transition firing 10
2.2 An asynchronous controller (a) interface and (b) STG 11
2.3 The previous asynchronous controller SG 13
2.4 The previous asynchronous controller (a) STG and (b) SG without CSC conflicts 14
2.5 Another asynchronous controller (a) STG and (b) SG 15
2.6 Karnaugh map and logic equations of the previous asynchronous controller 16
2.7 A handshake circuit 16
2.8 The handshake protocol: (a) 2 phases and (b) 4 phases 17
2.9 A handshake component 17

3.1 The *Sequencer* asynchronous protocol (a) interface and (b) STG 19
3.2 The *Sequencer* asynchronous protocol (a) SG with a CSC conflict, (b) STG with signal *csc0* and (c) SG without CSC conflicts . . . 20
3.3 The design flow of Petrify 21
3.4 The *ssem* net of handshake components 22
3.5 The design flow of Balsa framework 23
3.6 The *Parallelizer* asynchronous protocol (a) interface, (b) STG and (c) SG with CSC conflicts 24
3.7 The *Parallelizer* asynchronous protocol (a) STG and (b) SG without CSC conflicts 24
3.8 The design flow of Moebius 25
3.9 The signal *a_req* projection (a) STG and (b) SG of the *Concur* asynchronous protocol. Dummy transitions are denoted by the prefix "_eps" 26

4.1 A new design flow for asynchronous circuits 28
4.2 The *breeze2stg* flow 29
4.3 The SRR: (a) FSP, (b) FST, (c) FPP, (d) FPT, (e) ESP and (f) EST 33
4.4 The constructed graph of the two synchronized buffers 36
4.5 The two synchronized buffers (a) clustered graph and (b) cluster graph 37
4.6 The cluster STG of the two synchronized buffers 39
4.7 The cluster STG of the two synchronized buffers after performing SRR 40

5.1 The Balsa examples area results 44

A.1 (a) The *Synch* component and (b) its STG 49
A.2 (a) The *SequenceOptimised* component and (b) its STG 50
A.3 (a) The *Concur* component and (b) its STG 51
A.4 (a) The *DecisionWait* component and (b) its STG 52
A.5 The consistent *DecisionWait* STG 53
A.6 (a) The STG with irreducible CSC conflicts and (b) the STG without CSC conflicts of the *DecisionWait* STG 54
A.7 (a) The *Fork* component and (b) its STG 55
A.8 (a) The *Call* component and (b) its STG 56
A.9 The consistent *Call* STG 56
A.10 (a) The STG with irreducible CSC conflicts and (b) the STG without CSC conflicts of the *DecisionWait* STG 57
A.11 The *Call* STG with unique shared transitions 58
A.12 The efficient *Call* STG with unique shared transitions 59

B.1 The HC netlist of the arbiter tree 62
B.2 The clustered HC netlist of the arbiter tree 63
B.3 The cluster 0 HC netlist of the arbiter tree 64
B.4 The cluster 0 STG of the arbiter tree 65
B.5 The HC netlist of the population counter first part definition . . 66
B.6 The HC netlist of the population counter second part definition . 67
B.7 The clustered HC netlist of the population counter first part definition 68
B.8 The clustered HC netlist of the population counter second part definition 69
B.9 The cluster 0 HC netlist of the population counter 70
B.10 The cluster 1 HC netlist of the population counter 71
B.11 The cluster 7 HC netlist of the population counter 72
B.12 The cluster 0 STG of the population counter 73
B.13 The cluster 1 STG of the population counter 74
B.14 The cluster 7 STG of the population counter 75
B.15 The HC netlist of the shifter 76
B.16 The clustered HC netlist of the shifter 77
B.17 The cluster 0 HC netlist of the shifter 78
B.18 The cluster 0 STG of the shifter 79
B.19 The HC netlist of the SSEM 80
B.20 The clustered HC netlist of the SSEM 81
B.21 The cluster 0 HC netlist of the SSEM 82
B.22 The cluster 1 HC netlist of the SSEM 83
B.23 The cluster 2 HC netlist of the SSEM 84
B.24 The cluster 0 STG of the SSEM 85
B.25 The cluster 1 STG of the SSEM 86
B.26 The cluster 2 STG of the SSEM 87
B.27 The HC netlist of the stack first part definition 88
B.28 The HC netlist of the stack second part definition 89
B.29 The clustered HC netlist of the stack first part definition 90
B.30 The clustered HC netlist of the stack second part definition . . . 91
B.31 The cluster 0 HC netlist of the stack 92

B.32 The cluster 1 HC netlist of the stack 93
B.33 The cluster 2 HC netlist of the stack 94
B.34 The cluster 0 STG of the stack 95
B.35 The cluster 1 STG of the stack 96
B.36 The cluster 2 STG of the stack 97

List of Tables

4.1 The components frequency in Balsa examples 32

5.1 The Balsa examples results 43