School of Electrical and Electronic Engineering, University of Newcastle

MSc Wireless Embedded Systems 2013-2014, Semester 1

**EEE8043**: Design of Asynchronous Low Power Systems

Workcraft Exercises Report

Student: Michael Walker (130623935)

# Exercise 2: Half Handshake

## Simulation using Workcraft

We were asked to explore the implementation of four different methods for representing an asynchronous Half-Handshake controller. This controller prevents the progression of data on a bus from one block of logic to another, unless the proper request and acknowledgement sequence is followed. This protocol ensures data integrity by keeping the data path opaque to the receiving logic block until the data is stable. Two blocks (perhaps implemented as registers) are used in conjunction with an asynchronous control circuit. Block A is updated by the sender logic and when ready is sent to Block B to then be used by the receiving logic.

Below is a discussion of the 4 different implementations:

## Independent Method

In this method the signal transitions representing the requests and acknowledgements that are sent to and from the sender and receiver are split into two separate and distinct cyclical graphs. It can therefore be inferred that there is no coordination between the two halves, which, most likely, would lead to many problems in practice. In fact, this is not a genuine solution to the handshaking problem at all, as it does not fulfil the requirements of the specification given, which states that the data path should be considered opaque to the receiver until the request output signal ‘**Ro**’ goes high to ‘1’; which shouldn’t happen until the request input signal ‘**Ri’** is first set ‘1’ by the sender. This coupling of the signals is simply not present in the given STG below.

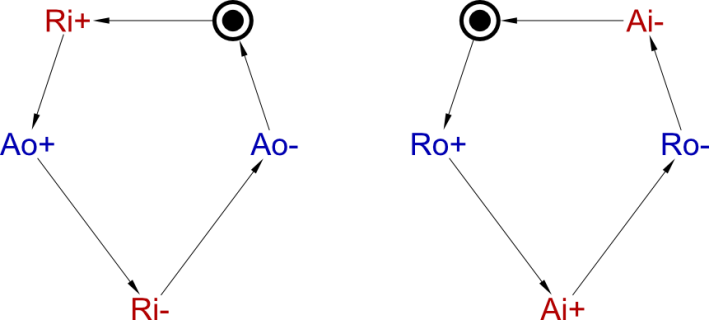


Figure : Independent STG

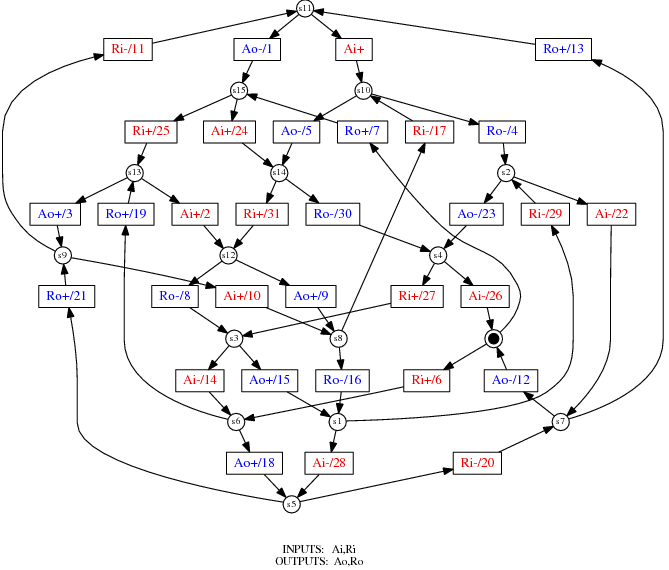


Figure : Generated Stage Graph for Independent Method

### Unbounded Method

This method joins the independent graphs above by the minimal addition of connection between the two halves of the STG as given in the specification, stated: That the ‘**Ri+**’ transition is a condition to the ‘**Ro+**’ transition, ie. “*Ri becoming 1 indicates the data in Block A is stable, causing B to receive the data and place Ro and Ai at 1*”.

However as a result of this minimal addition, the system suffers from problems of **unboundedness**. This can be observed in the behaviour of the graph when simulated in Workcraft. Tokens ‘pile up’ on the arc between the ‘**Ri+’** and **Ro+** signal transitions, this is called *infinite token generation*. The reason for this unbounded behaviour stems from **Inconsistency** caused primarily by the lack of a reciprocal linkage when signals transition back from high to low again; i.e. there needs to be another ‘reflecting’ link such that ‘**Ri-**‘ is required for ‘**Ro-**‘ (this is only one example – other links would solve this issue - it is specifically the graph that requires another arc for consistency). Boundedness can be visually checked by ensuring that all arcs are part of some ‘loop’ in the graph, this can be stated more mathematically as saying that a Petri net (general case of an STG) is unbounded if and only if there exists a reachable marking M and a sequence of transitions such that http://latex.codecogs.com/gif.latex?M%20%5Coverset%7B%5Csigma%7D%7B%3D%7D%20M%20&plus;%20L where L is some non-zero marking [1]. It seems to be, more generally that the mathematical form of a STG is a directed graph, which must have the property of *Strong Connectivity* although we have not investigated the mathematics of this sufficiently for one to state it conclusively.

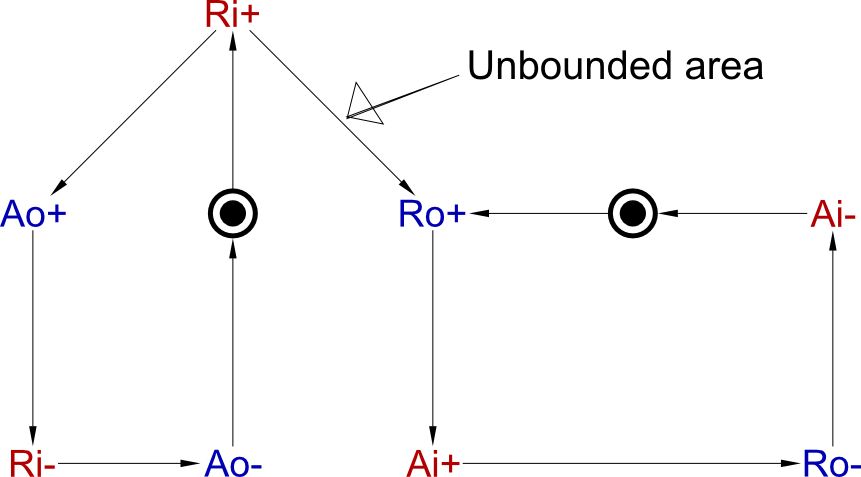


Figure : Unbounded STG

More specifically, in this case the unbounded area is the arc between **Ri+** and **Ro+** in the above STG. Also the Petrify tool cannot generate a final state graph, as the problem is inherently undecidable; hence it is not shown below in contrast to the sections describing the other three methods.

### CSC-Violation Method

This method is again an improvement on the one that came before. Now there is a better linkage between the two sides of the handshake and there are no unbounded areas. This is caused by the additional arcs requiring **Ro-** before **Ao-** and **Ro+** to **Ao+**. This graph is now bounded and fulfils the requirements of the specification. However is still has problems for logical synthesis as it suffers from **incomplete state encoding**.

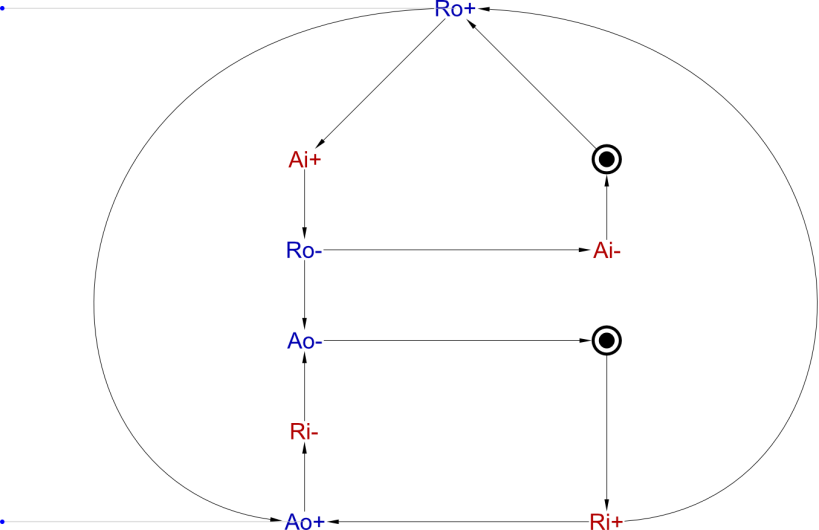


Figure : CSC-Violation Method

By generating the .g file for the preceding diagram and running the following Workcraft tool commands in the command line:

* write\_sg -bin cscviolation.g -o cscviolation.sg
* draw\_astg -bin cscviolation.sg –o cscviolation.ps

It was possible to generate the following graph to show where those state encoding violations occur:

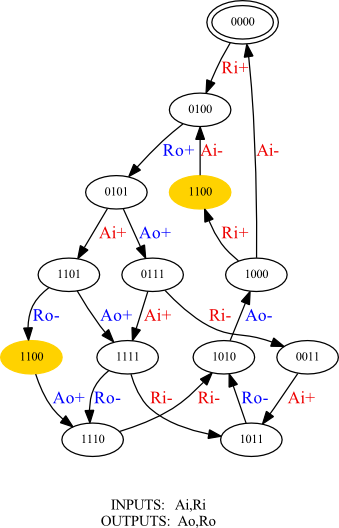


Figure : CSC-Violation Binary State Encoding

The important thing to note is that the binary state encodings for the arcs from **Ro-** to **Ao+** and **Ri+** to **Ai-** are the same (‘1100’ in the diagram). This needs to be resolved before the logic circuit can be synthesized following the asynchronous logical circuit design flow we have been taught.

Workcraft can resolve the problem automatically. This results in the insertion of an additional internal binary state and two extra signal transitions to the STG, thus creating unique state encodings for all transitions. The result of this automatic STG generation is shown in the figure below:

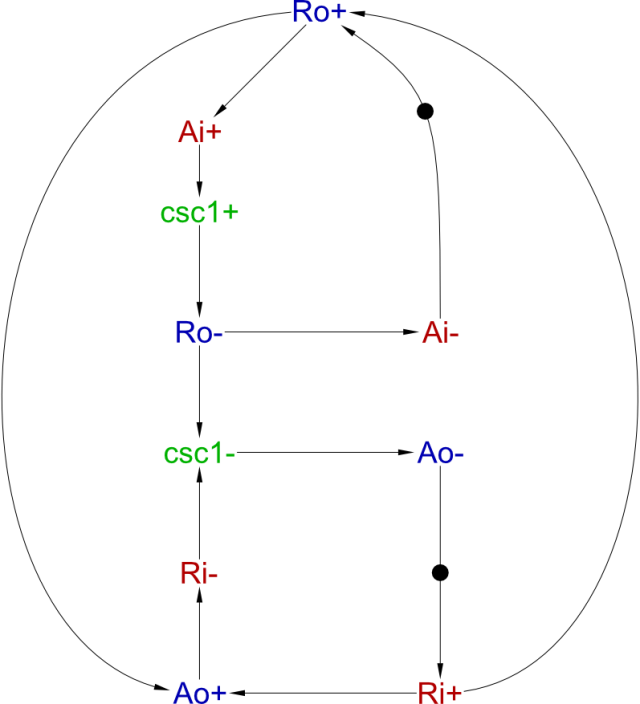


Figure : CSC-Violation-Resolved Method

The new internal signal ‘csc1’ can be seen added to the diagram on the previous page. The Stage Graph however has grown in complexity, as can be seen by comparing the State Graphs for the CSC-Violation and CSC-Violation-Resolved STGs:

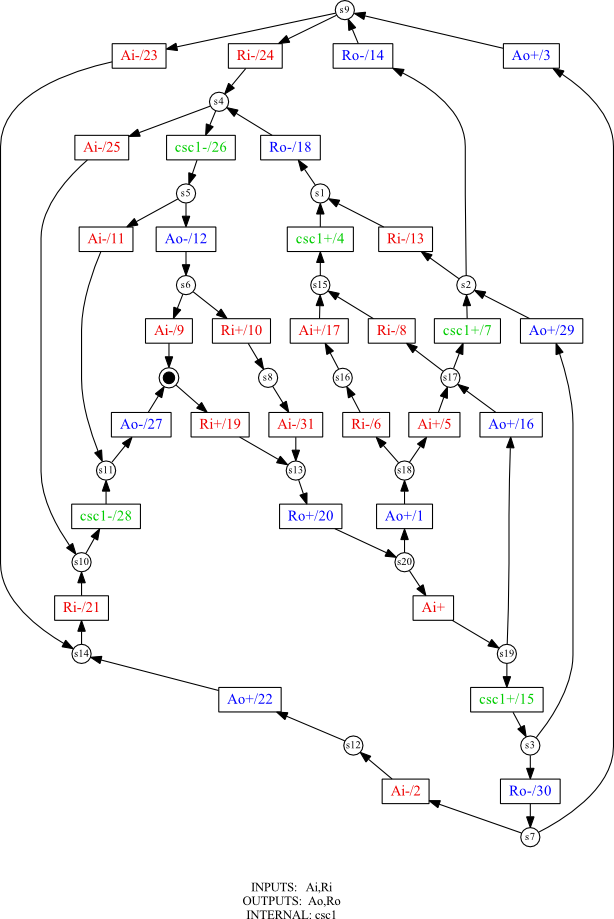
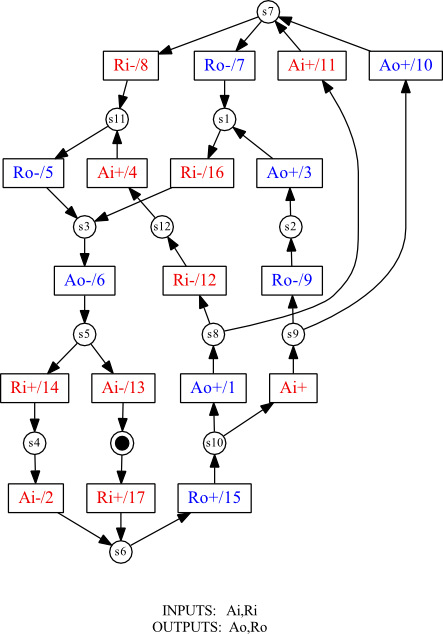


Figure : CSC-Violation Resolved Stage Graph

Figure : CSC-Violation Stage Graph

The Original CSC Violating method has **18 transitions** and **13 states**; whereas the resolved version has **32 transitions** and **21 states.** The extra complexity is caused by increased redundancy due mostly to the extra overhead incurred by the addition of internal state. However the resulting resolved STG is now implementable through our synthesis method as it is **Consistent**, **Persistent** and has **Complete State Coding**.

## No Redundancy Method

Finally the last method modifies the connections between state transitions with the result of removing the internal state required the previous resolved STG:

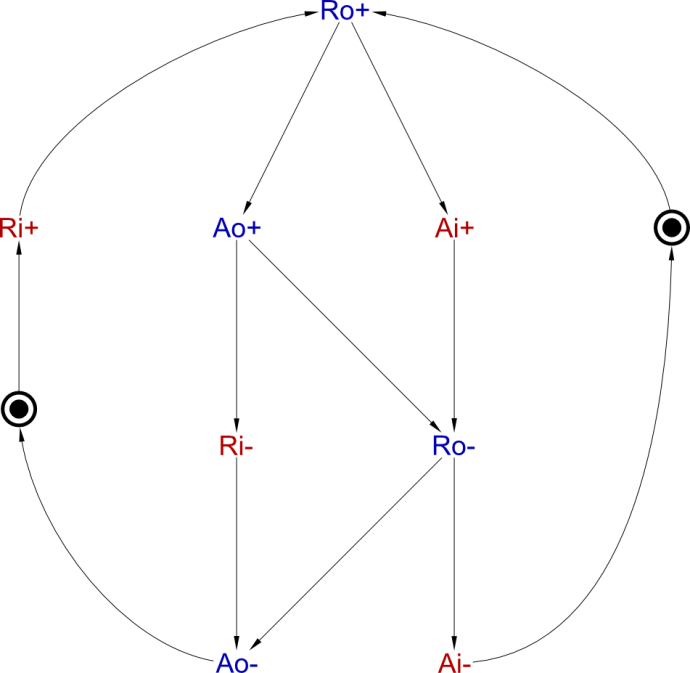


Figure : No Redundancy Method

While the diagram looks fundamentally different from the CSC-Violation Method, there are only two alterations. Firstly, the **Ri+** to **Ao+** arc is removed from the graph and in its place an **Ao+** to **Ro-** is added. This small change removes two different redundant paths from the system. This is because in the simple case the **Ri+** to **Ao+** is redundant as, at the same time, **Ri+** causes **Ro+** which in turn causes the same **Ao+** that **Ri+** to **Ao+** causes to happen anyway. By removing the arc, the design is simplified.

The second redundant path through the CSC-Violating STG is a little harder to spot; but it is that **Ai-** causes **Ro-**. Going back one step we can see that **Ro-** causes **Ai-** which in turn causes **Ro+**. However, we can note that **Ro-** also causes **Ao-** which causes **Ri+** which finally causes **Ro+** as well.

Simply removing the **Ri+** to **Ao+** arc from the CSC-Violation method still results in an STG with an incomplete state encoding. Adding the **Ao+** to **Ro-** arc has the unfortunate side effect of reducing concurrency, which could introduce delays into the final implemented circuit. However, it does resolve the state encoding conflict without requiring the insertion of an extra internal signal.

By removing the internal signal required by the CSC-Violation Resolved STG, we gain a simplified Stage Graph, with **20 transitions** and **14 States**. This is only a slight increase on the CSC-violating method, yet still produces a valid (**Consistent**, **Persistent** and has **Complete State Coding**) stage graph, as shown below:

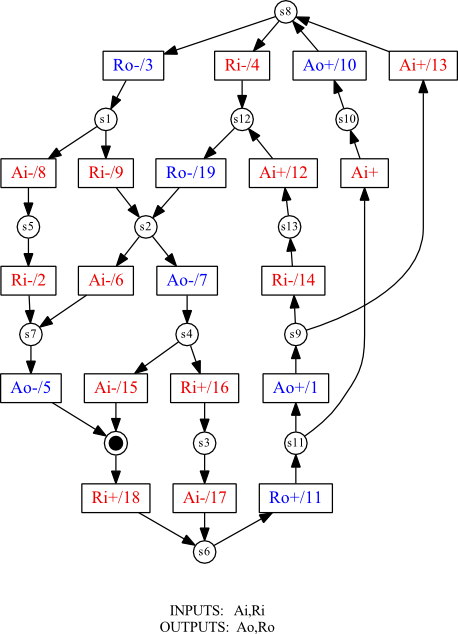


Figure : No Redundancy Stage Graph

The final stage of the synthesis design flow is to use the Petrify tool to synthesize an asynchronous circuit that implements the given STG as ‘*given a signal transition graph it produces an optimized net-list of an asynchronous controller in the target library while preserving the specified input-output behaviour’*[3]. The net-list generated by petrify for this method is (in effect) just two lines:

[Ao] = Ri Ao + Ro;

[Ro] = Ai' (Ri Ao' + Ro) + Ao' Ro;

This compares to the three line results for the CSC methods (the tool automatically seems to resolve the CSC-violation but does so in a slightly different way to the ‘Fix CSC Violation’ Tool.

**CSC Violation Net-list**

[Ao] = csc0';

[Ro] = csc0 (Ri Ai' + Ro) + Ai' Ro;

[csc0] = Ro' (csc0 + Ri');

**CSC Resolved Net-list**

[Ao] = csc1 + Ro;

[Ro] = csc1' (Ai' Ri + Ro);

[csc1] = csc1 Ri + Ai Ro;

The net-list for the No Redundancy Method may look reasonably simple, consisting of two lines of text. However, when attempting to draw the diagram of the gates required, it can be seen in the circuit diagram below that the circuit involves quite a lot of complex feedback. The split of the two lines can also be seen: The 3-input complex-gate at the top of the circuit diagram corresponds to the first line of the Net-list, and the lower 6-input complex-gate corresponds to the second. This is by no means the optimal way to draw the circuit, but simply as an illustration that a representation of the circuit diagram could and was drawn by hand (Or rather using the computer vector graphics drawing tool ‘*Inkscape*’). Better tools are available for the creation of clearer schematics, but gaining proficiency in these was considered outside the scope of this project.

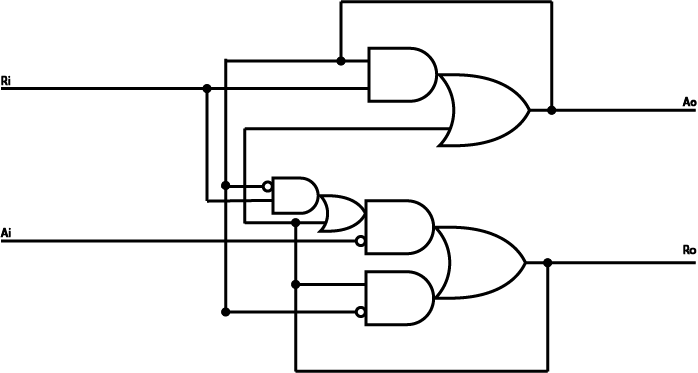


Figure : Drawn Circuit that Implements Net-list

# Exercise 3: VME Bus Controller

VMEbus is a computer architecture. The term 'VME' stands for ‘*VERSAmodule Eurocard’* and was first coined in 1980 by the group of manufacturers who defined it. This group was composed of people from *Motorola*, *Mostek* and *Signetics* corporations who were cooperating to define the standard. The term 'bus' is a generic term describing a computer data path, hence the name VMEbus [2].

We were asked to implement an asynchronous controller for the VMEbus using Workcraft. Following the design flow it is first necessary to implement the specification as a State Transition Graph. This resulted in the following graph:

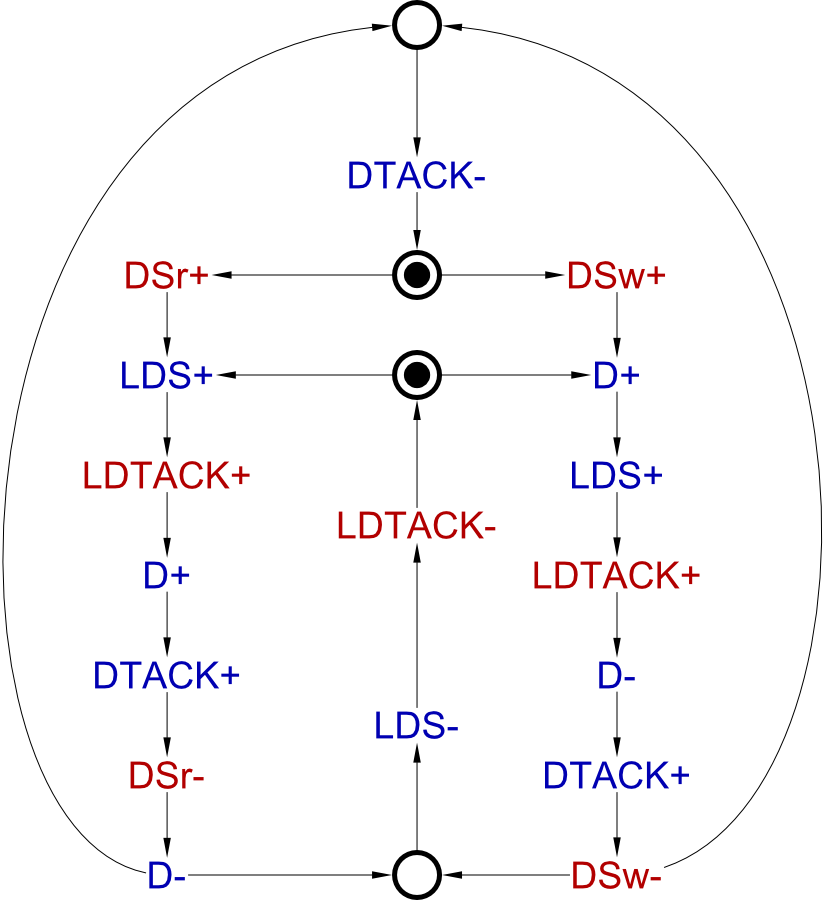


Figure : VMEbus STG

Please note that this is the Full STG of the VMEbus Controller. We were originally asked to construct the STG for the read cycle only. However, it was suggested during the later laboratory sessions that exploring the full controller behaviour would be of further interest. Therefore the full controller synthesis is discussed in this report instead of the read cycle.

After constructing the diagram and running some simulations in the Workcraft environment to ensure it was deadlock free and bounded, the next task was to generate a Stage Graph:

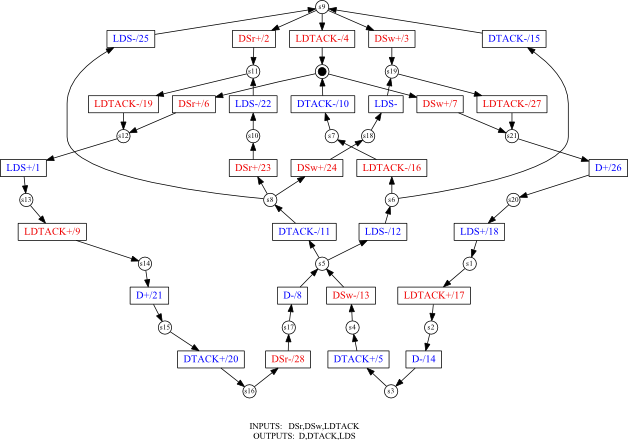


Figure : VMEbus Stage Graph

This Stage graph has **29 transitions** and **22 states**, and the symmetry of the read and write cycles is apparent.

However it has 2 CSC violations as shown in the following binary signal transition graph generated in a similar way to that in exercise 2:

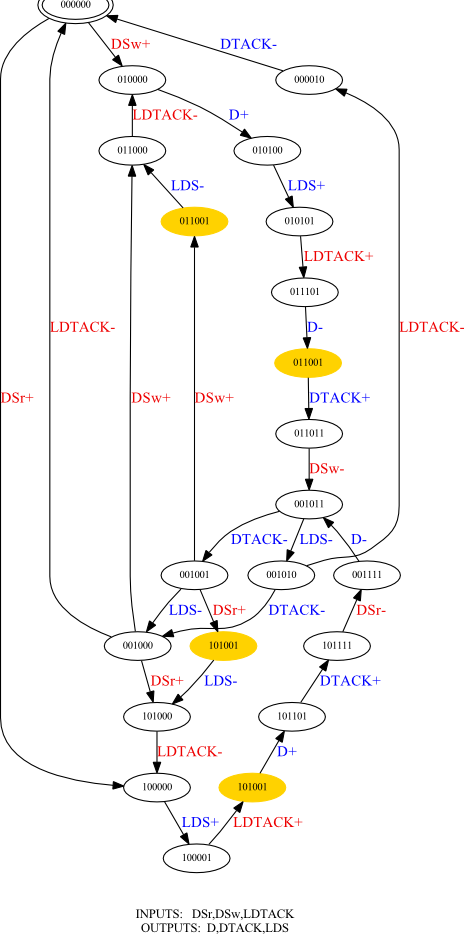


Figure : VMEbus Binary Encoding STG

Thus we need to perform a CSC fix using the Workcraft tool and recreate the STG and the Stage Graph for the non-violating case. This has already been discussed in the previous section on the Half-Handshake Controller and the procedure carried out is identical. Therefore, merely the results of this are shown on the next page:

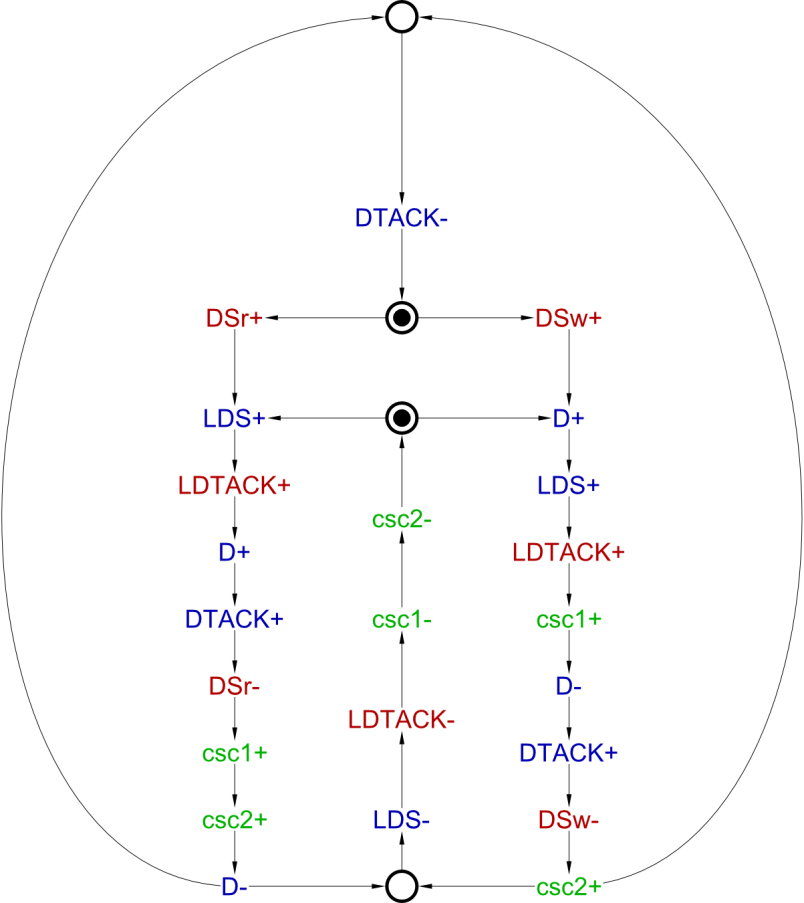


Figure : VMEbus CSC Violations Resolved

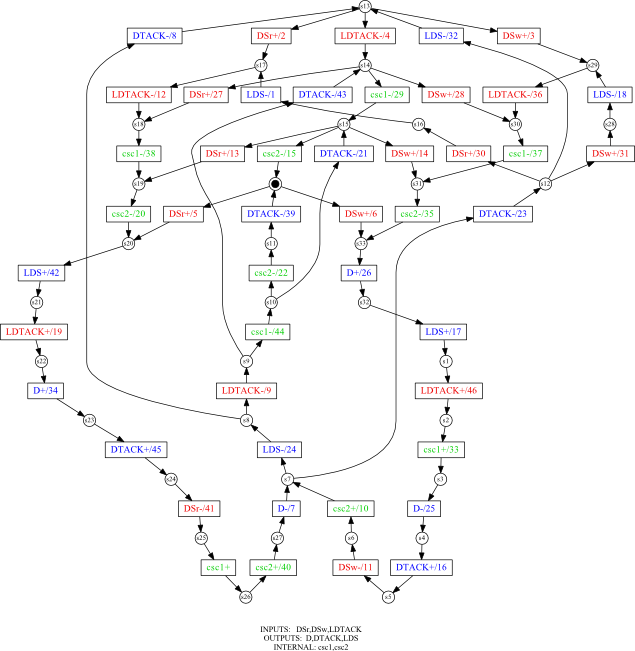


Figure : VMEbus CSC Resolved Stage Graph

Having fixed the CSC Violations, the STG is now **Consistent**, **Persistent** and has **Complete State Coding**. Thus we are ready to synthesize the STG as a circuit. The Petrify tool can be used to synthesize the circuit, and can be passed options to target different implementation templates, for example: -cg for complex gates; -gc for a generalized C-element circuit; -gcm for a C-element solution that satisfies the monotonic cover requirement and –tm to perform a technology mapping to a given gate library[4].

Using the petrify tool to do so, the resulting complex gate net-list is produced as follows:

# EQN file for model h:\Projects\EEE8043\vme

# Generated by petrify 4.2 (compiled <unknown compile date>)

# Outputs between brackets "[out]" indicate a feedback to input "out"

# Estimated area = 22.00

INORDER = DSr DSw LDTACK D DTACK LDS csc1 csc2;

OUTORDER = [D] [DTACK] [LDS] [csc1] [csc2];

[D] = DSw' D csc2' + csc1' (csc2' DSw + LDTACK);

[DTACK] = D' csc2' csc1 + D DSw';

[LDS] = csc2' (DSr + csc1) + D;

[csc1] = LDTACK (DSr' + csc1);

[csc2] = csc1 (DSw' + csc2);

# Set/reset pins: reset(D)

Using the petrify tool with technology mapping option enabled to decompose the circuit into 3-input gates results in a much larger net-list, as shown below:

# EQN file for model h:\Projects\EEE8043\vme

# Generated by petrify 4.2 (compiled <unknown compile date>)

# Outputs between brackets "[out]" indicate a feedback to input "out"

# Mapping with library petrify.lib in 1.14 sec --on host a Windows host

# Total area = 672.00 using depth level 2, and area/delay ratio of 0.00

INORDER = DSr DSw LDTACK D DTACK LDS csc1 csc2 map4;

OUTORDER = [D] [DTACK] [LDS] [csc1] [csc2] [map4];

[0] = csc2' csc1'; # gate nor2:combinational

[1] = LDTACK' DSw' + [0]'; # gate oai12:combinational

[D] = map4' + [1]'; # gate nand2:combinational

#PRAGMA: zero delay

[3] = csc2'; # gate inv:combinational

[4] = D' ([3]' + csc1'); # gate aoi12:combinational

[DTACK] = [4]' (DSw' + D'); # gate aoi12:combinational

#PRAGMA: zero delay

[6] = csc2'; # gate inv:combinational

[7] = D' ([6]' + DSr'); # gate aoi12:combinational

#PRAGMA: zero delay

[8] = csc1'; # gate inv:combinational

[LDS] = [8]' csc2' + [7]'; # gate oai12:combinational

#PRAGMA: zero delay

[10] = D'; # gate inv:combinational

#PRAGMA: zero delay

[11] = csc2'; # gate inv:combinational

[12] = [11] D DTACK; # gate and3:combinational

[13] = DSr' map4' + [12]'; # gate oai12:combinational

[14] = [13]'; # gate inv:combinational

#PRAGMA: zero delay

[15] = map4'; # gate inv:combinational

[16] = [14]' ([15]' + [12]'); # gate aoi12:combinational

[17] = [16] DSw'; # gate and2\_1:combinational

[18] = [10]' [17]' [14]'; # gate nor3:combinational

[csc1] = [18] (LDTACK + csc1) + LDTACK csc1; # gate c\_element0:asynch

#PRAGMA: zero delay

[20] = map4'; # gate inv:combinational

[21] = [17]' [20]' + csc1'; # gate oai12:combinational

[csc2] = csc2 csc1 + [21]'; # gate sr\_nand:asynch

#PRAGMA: zero delay

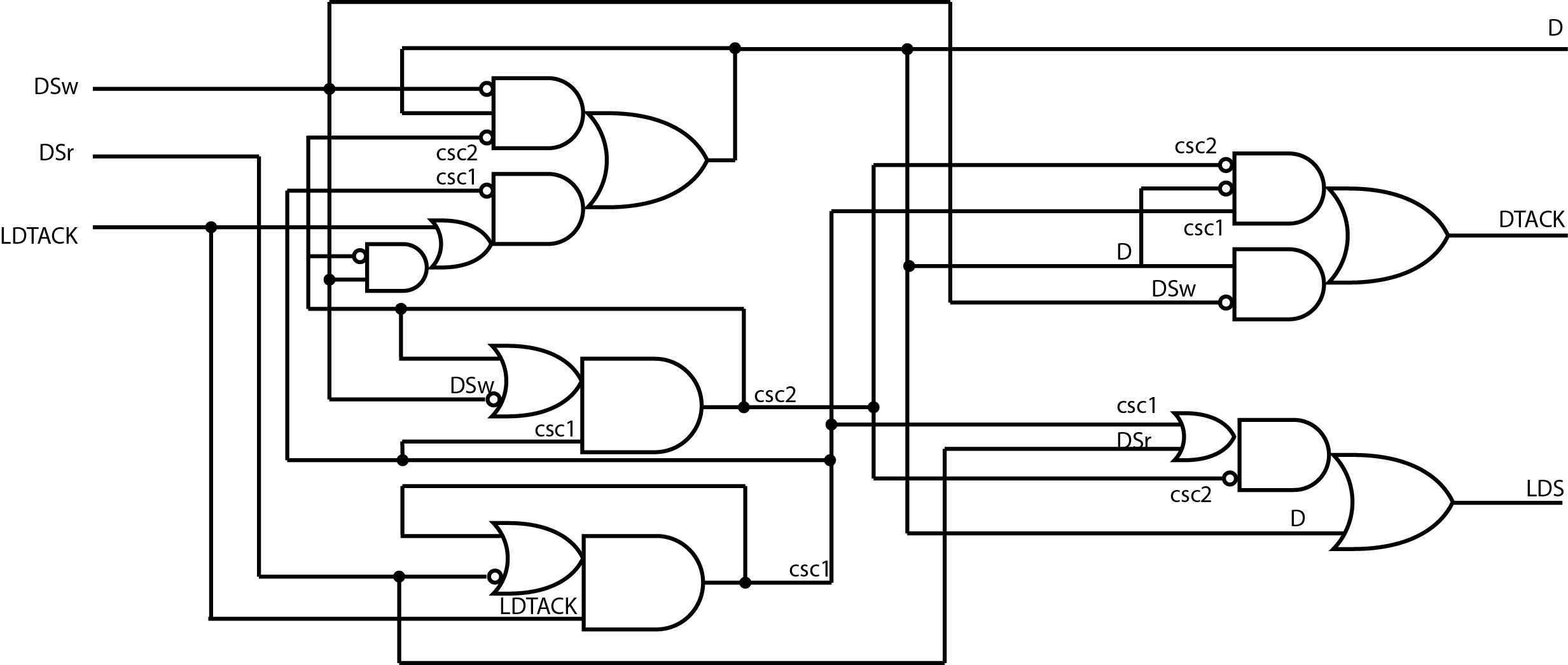
[23] = D'; # gate inv:combinational

[map4] = DSw + [17] + [23]; # gate or3:combinational

# Set/reset pins: reset(D)

This larger net-list would seem likely to be the result of the decomposition of the complex gates into a restricted library of 3-input gates. This **Technology Mapping** is important as it allows the implementation of the circuit using a practical, real-world component library. In effect, the technology mapping shows the internal signals that are implicit in the complex-gate net-list. These are the numbered outputs [1]..[23] used within the net-list code.

Drawing the Technology Mapped 3-input gate synthesis result was not attempted. However, the following circuit was drawn based upon the complex-gate Net-list:



Although the circuit layout is sub-optimal and could be improved greatly, this was not attempted as it was considered unnecessarily time-consuming, and the purpose was only to complete the process and produce a concrete result. It is also noted that the use of C-elements would help greatly in making the circuit more instructive and clear in its purpose, again this was not carried out due to time constraints.

The use of both Petrify and the Workcraft Editor with its associated tools has been a very informative introduction to the fundamental principals and techniques of asynchronous synthesis, and has helped in gaining a clearer appreciation of the topic of asynchronous circuit design in general.

## References

[1] J. Esparza & M. Nielsen *Decidability Issues for Petri Nets*, p5 BRICS Report Series RS-94-8 May 1994: <http://home.ifi.uio.no/andersmo/petrinet/papers/complexity/brics_98_8.pdf>

[2] VITA (VMEbus International Trade Association) Website *FAQ on VME history and basic technology*: <http://www.vita.com/home/Learn/vmefaq/vmefaq.html>

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno & A. Yakolev *Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers* In IEICE Transactions on Information and Systems, Vol. E80-D, No. 3, March 1997, p315-325.

[4] J. Sparsø & S. Furber *Principals of Asynchronous Circuit Design, A Systems Perspective*. Chapter 6.7, p103