An automatic synchronous to asynchronous circuit convertor

Charles Brej

Abstract

The implementation methods of asynchronous circuits take time to learn, they take longer to design and verifying is very difficult. For these reasons tools are used rather than designing by hand. Unfortunately by adding this level of abstraction the designer rarely has any idea of what the gate level design will look like and how to opitomise it. To try and solve some of these problems this paper introduces a new approach where synchronous designs can be easily converted to their asynchronous counterparts and still keep all of the functionality of the original circuit. There are limits to what this system can convert but generally most synchronous state machines can be converted.

1. Introduction

It is difficult to visualise the operations of asynchronous circuits and make sure they do not deadlock. A tool that would allow a designer to design circuits and optimise them at a very low level, yet still make sure that no deadlocks or hazards occur and still create high performance circuits, would be very welcome. This is exactly what the tool introduced in this paper is trying to achieve.

The basis of the tool is “dual-rail”. Dual-rail signalling uses two wires to represent one bit of data. Raising one line represents a zero and raising the other line represents a one. When both of the lines are low then no data is being sent (this is called the “null” signal) [JDG]. Return To Zero [RTZ] protocol is used to communicate the dual rail codes around the circuit. RTZ is a four phase protocol where the a set of wires (in this case two) send data and the target raises an acknowledge wire when it receives the data. The host then returns to its null state by lowering its data line. When the target sees its input returning to null it signals that it is ready to accept new data by lowering its acknowledge line.

2. Three stage counter walk-through

2.1. Target circuit

Figure 1 shows a synchronous finite state machine circuit. This circuit cycles through three states.

Each symbol in this figure instantiates a logic circuit, the realisation of which is abstracted at this level. To convert this to an asynchronous circuit the components can be replaced by their dual-rail asynchronous counterparts. The counterparts have to have dual-rail inputs and outputs.

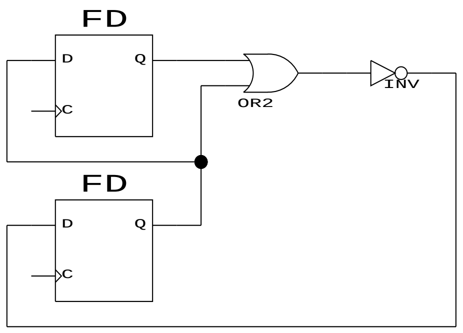

FIGURE 1. Synchronous Implementation of a modulo-3.

2.2. Component substitution

An asynchronous circuit can be generated by substituting each component in the synchronous original with its dual-rail equivalent from the appropriate library. Each net is duplicated and renamed with '_0' and '_1' postfixes to represent the two Boolean values. The complex dual-rail gates can be abstracted to their own symbols. Figure 2 shows the dual-rail implementation of the three stage counter circuit created by replacing the synchronous elements with equivalents in the standard dual-rail library.

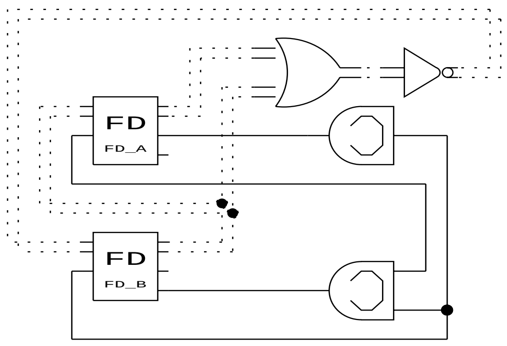

FIGURE 2. Dual-rail implementation of the three stage counter.

The upper latch outputs data to the OR gate where it is synchronised with the data from the lower latch. When both inputs are valid the OR gate will output a value. This value then goes through the inverter and into the lower latch. The upper latch simply accepts the data from the lower latch. These forward data paths are shown in dotted lines in the figure.

To control the data flow in the absence of a clock each latch emits an acknowledge signal when it has accepted new data. Other latches wait for an acknowledge signal before removing data from their outputs. If a latch outputs to two latches then both acknowledge signals have to be combined using a C element. This sequences the operation of the circuit.

2.3. Dual-rail gates

An implementation of a dual-rail OR gate can be seen in Figure 3. This is a very large component compared with the original OR gate. The set of four C-elements make sure that the output only switches when all inputs have switched to valid states. For any two input gate four C-elements are required. For any three input gate eight C-elements are required. As the number of inputs to a gate rises the number of C-elements in its asynchronous counterpart rises geometrically. Also the number of inputs to these C-elements rises with the number of inputs to the gate. Fortunately in dual rail logic an inverter has no cost as inside an NOT gate the two nets are simply swapped over to create the inversion.

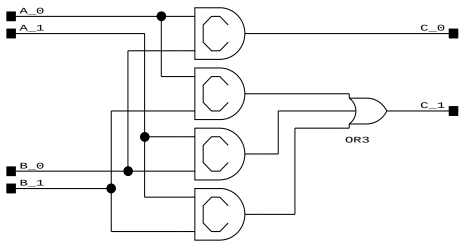

FIGURE 3. Asynchronous dial-rail counterpart to the two input OR gate.

Substituting combinatorial logic components is fairly straightforward and no considerations have to be made. This, unfortunately, is not the case when replacing data storing elements.

2.4. Dual-rail latch

Figure 4 is a schematic of a simple dual-rail latch commonly used in asynchronous designs. These latches follow the four phase RTZ protocol. When a latch is not receiving an acknowledge it will take an incoming datum and store it in the C-elements. If any datum is stored in the C-elements the latch will send the acknowledge signal out. There is a reset signal connected which will clear the latch but only if the data inputs are low.[JDG]

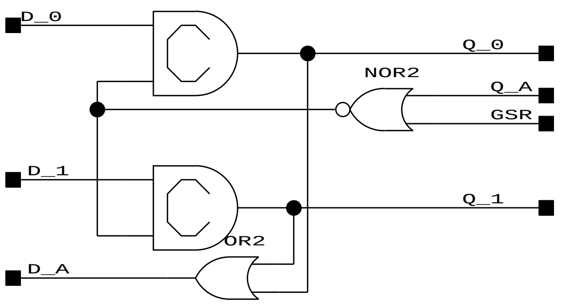

FIGURE 4. Simple dual-rail latch

When implementing pipelines there must always be at least twice as many latches as data tokens because data tokens are always separated by ‘null’ tokens. However using just two simple latches to replace the D-type flip-flops can cause the circuit to deadlock. Synchronous flip-flops act on a clock signal and can get new data and lose the old data instantaneously while latches need to go through a ‘null’ stage before a new output is valid.

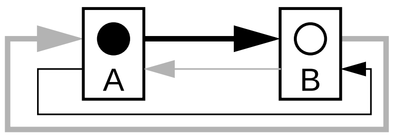

FIGURE 5. Two latch loop deadlock

The minimum number of elements in a loop is therefore three: one to hold ‘data’, one to hold ‘null’, and one ‘free’ to allow one of the other values to move forwards. The ‘flip-flops’ are therefore replaced with sets of three dual-rail latches. This is allows a latch to accept data while still transmitting. Figure 5 shows two latches in a loop where latch A is transmitting data to latch B but latch B cannot accept this data as it is still receiving an acknowledge; the circuit is deadlocked [JDG]. In situations other than the ‘tightest’ loops, having 50% more latches than required can introduce a large delay overhead. The tool will in future versions remove latches where possible and the delay cost is too large. At reset time all latches must start with a data token to represent the data in the synchronous version of the circuit at reset.

2.5. Acknowledge circuit

Each latch requires an acknowledgement when all the latches it outputs to have received the data to allow it to return to null; This is also true in the return to zero stage. In Figure 2 the top latch (FD_A) needs an acknowledge from FD_B as that is the only latch it outputs to. The bottom latch (FD_B) needs a combined acknowledge from both latches as it needs to wait until both have accepted the data before removing it from the bus. This is reflected by the C-elements in the return paths; clearly the single input gate can be optimised away!

2.6. Simulation

When simulated this design will run as fast as possible as it has no inputs or outputs to synchronise with. Figure 6 shows the simulation of this circuit. Unfortunately the design is very slow due to the complexity of the dual-rail gates.

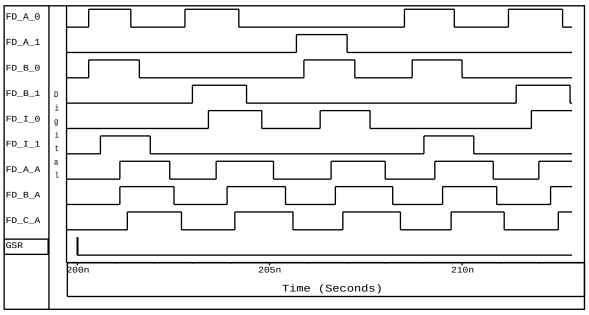

FIGURE 6. Simulation wave of dual-rail three stage counter.

The other problem with this approach is that, even if one of the signals is received by the OR-gate and the output can be resolved, the gate waits for the second input before driving the output. This wait is necessary so the acknowledge signal is asserted only when all inputs are valid otherwise a signal will could after the other pieces of data came and were acknowledged and thus the data incoming is one stage behind. The logical and timing circuits are combined into one large and slow circuit. By separating out the timing and the logical parts we can achieve a smaller and faster implementation.

3. ‘Unguarded’ elements

3.1. Unguarded OR gate

The original, dual-rail library used components with both timing and logical elements. By separating out the timing and the logical parts it becomes possible to achieve a smaller and faster implementation. This library of components is called ‘unguarded’ as it has no timing controls. The circuit in Figure 7 shows an unguarded implementation of the dual-rail OR gate. This implementation will raise the output O_1 high when *either* of the inputs signals a one as at this point the result is known. Unfortunately the separation of the logical and timing parts could cause a hazard by ceasing to observe one (or more) input. This is prevented by adding a third, ‘validity’ signal which indicates that the gate’s input set is complete. The O_VALID line is only raised when both inputs are valid. This gate is now about two times smaller and faster than the guarded version.

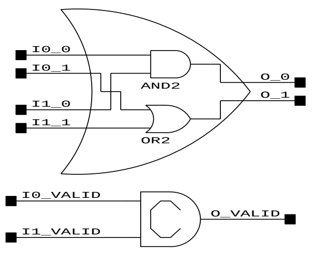

FIGURE 7. Unguarded dual-rail OR gate

3.2. Unguarded implementation

In order to create circuits using these gates a test must be made to make sure all the inputs are valid before acknowledging them. In the previous design style the result would only become valid when all inputs were valid and the logical operation has a result. In the unguarded implementation the acknowledge from the latch means only that the logical part of the operation has completed. Some of the inputs might not yet be valid so the circuit must wait until all inputs are valid before acknowledging. To allow this latches have a VALID pin added which states if the latch is outputting a value. Figure 8 shows an intermediate circuit created by the tool where the validity nets are shown with dashed lines. Directly under the OR and NOT gates are C-elements combining the validity signals of the inputs to these gates. These validity nets are then combined with the unguarded acknowledge of the latch the data flows into. These trees of C-elements can be flattened down to one large C-element.

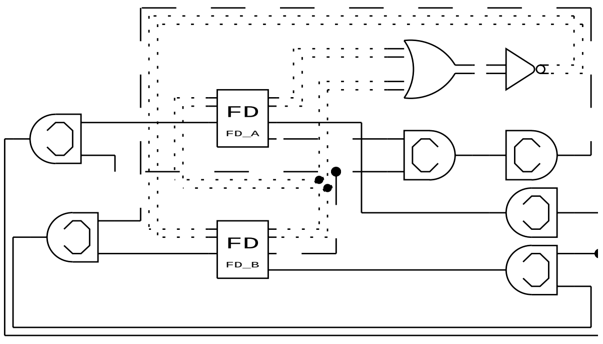

FIGURE 8. Unguarded dual-rail implementation of the modulo-3 counter

Figure 9 shows the tool’s final implementation of the modulo-3 counter using the unguarded element library. In this design two new C-elements (CA_A and CA_B) are used to signal when the latches acknowledge and all the inputs are valid. These replace the validation and combining with acknowledge C-elements from Figure 8. The outputs of these C-elements replace the acknowledge outputs from the latches in the guarded design. C-element CA_A takes the acknowledge signal of FD_A and the validity signals from all latches that feed into FD_A (in this case FD_B VALID pin). Now CA_A outputs the guarded acknowledge of FD_A. CA_B similarly takes the acknowledge from FD_B and the validity signals from all latches that feed into FD_B and outputs the guarded acknowledge of FD_B. CB_A and CB_B as before combine the guarded acknowledge signals from the latches.

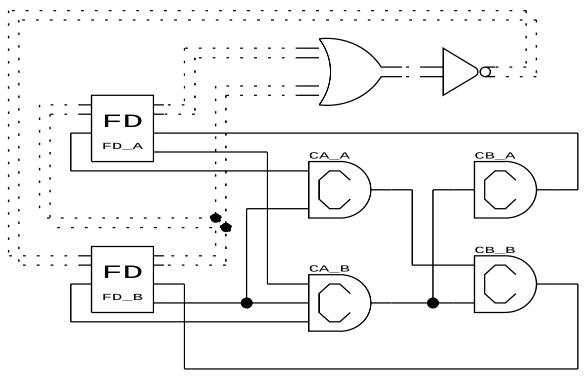

FIGURE 9. Unguarded dual-rail implementation of the modulo-3 counter with combined validation C-element trees.

4. Does it work?

It is difficult for to prove that this system works but the following are some of the most common questions.

4.1. How do inter-pipeline stage communications work?

A common misconception is that these systems cannot communicate across pipeline stages. Figure 10 shows an example of a gate whose inputs come from two different stages. The gate expects the stages to be filled with data five clocks apart. The figure shows that one of the stages is not filled and the gate is looking at the wrong data. This situation will not arise as both the source latches are synchronised by the latch C and this doesn't allow latch B to move onto the next set of data until latch A has entered its inputs. [AB]

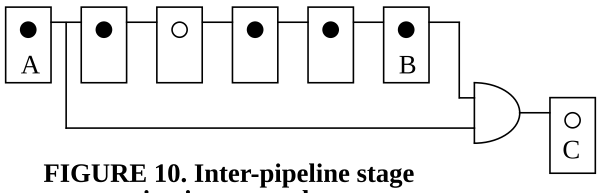

FIGURE 10. Inter-pipeline stage communication example

4.2. How does this approach cope with large logic blocks?

When using this approach to implement blocks of logic with a large number of inputs and outputs the quantity of inputs into the C-elements grows to amounts where it becomes far too slow and energy consuming to use this approach. When creating elements such as a 32 bit adder then the top output bit tries to synchronise all the inputs as it is dependent on them. This would require a 65 input C-element which is far too costly for such an operation. To solve this problem we introduce a new element. It simply is an empty latch and is described in the next section.

4.3. Is there anything it cannot do?

There are two circuit types that the tool cannot convert. The first is the tristate buffer nets. This functionality will be added to future versions of the tool. The second is already asynchronous circuits. These types of circuits cause deadlocks as one wire can change state more than once per clock cycle and a circuit not involving a latch can have hysteresis. Figure 11 shows an example of an asynchronous circuit deadlocking after conversion. Both gates are co-dependent on each other's results to produce and output. In a guarded implementation this circuit will deadlock every time. In and unguarded implementation this will only deadlock when R is high and S is low. This is because the circuit is designed to keep its state but as the circuit is always reset between data stages there is no state to keep. The tool will not detect these circuits and convert them assuming the user has made sure deadlocking situations will never arise.

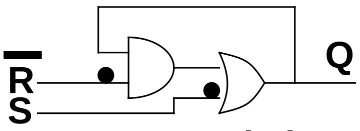

FIGURE 11. S-R latch deadlock example

5. Optimization

5.1. Empty latch

All latches in the system start at reset with valid data similar to the synchronous version. The empty latch is a latch that starts with no data inside and so is functionally invisible. It is an effective way of dividing a pipeline stage into two or more sections. Currently in the synchronous design schematic the tool uses buffers to represent empty latches as they are functionally invisible. If in the example of the adder these empty buffers were placed on the carry chains the functional operation of the adder would not change but these buffers would split the element into 32 pipeline stages. Each stage would have only three inputs and two outputs. More interestingly the carry output often can be calculated before the all inputs are valid. If this adder was placed inside a processor running code that adding two numbers, many bits can be output before the whole computation is completed.

5.2. Pre-emptive acknowledge

The above example showed how the not all inputs are required to form some outputs. In other cases an input is not required at all as the other inputs have already resolved all outputs. In this case the stage stalls and waits for a valid input before acknowledging. For example a multiplexer can get the select and the desired data input but will have to wait for the second data input. The desired situation is if this stage would be able to move to the next set of data from the inputs and tell the stalling input to go and acknowledge itself. This method is not yet implemented into the tool but will be a future feature. When used correctly the tool produces circuits of asynchronous elements with acknowledge lines on each dual-rail bus. Basically 1 bit wide pipelines. By creating these circuits it becomes possible to forward some bits of the result to the next operation while others are still being worked on.

6. Conclusions

This paper presents a tool for converting synchronous designs and a very elegant level of abstraction for designers. The tool is just an example of the advantages of this approach to designing asynchronous logic and is probably not the best implementation.

6.1. Acknowledgements

I would like to thank James Garside and Andrew Bardsley and others who have found many holes in my theories. Also all the members of the Amulet group and the UK Async. forum which is an excellent place to bounce ideas around.

6.2. References

[RTZ] Martin Rem,“The Nature of Delay Insensitive Computing”, Higher Order Workshop, Banff 1990.

[JDG] Conversations with Jim Garside.

[AB] Conversations with Andrew Bardsley.

[WT] Conversations with William Toms.