Appendix A

Circuit Module Implementations

A.1 Control Modules

A.1.1 Merge

A Merge module can be implemented as an XOR gate. A standard CMOS implementation of an XOR is shown in Figure A.1 which assumes that both the positive and negative versions of the input signals are available. If the inputs are at different values, the output node is pulled up, and if they are at the same value, it the output node is pulled down. If the positive and negative version of the inputs are not available they must be generated using inverters.

A.1.2 Join

A Join module may be implemented as a Muller C-element which may, in turn, be built in a variety of ways. The C-elements used for the circuits in this thesis are all based on the simple dynamic C-element shown in Figure A.2. This is a simple inverting C-element. When both of the inputs are high, the output is pulled low. When both inputs are low, the output is driven high. When the inputs are in different states, the output is not driven and holds its last value in the dynamic storage of the output node.

For self-timed circuits, however, this dynamic behavior is not acceptable. The output node must be made static. One way to provide a restoring signal to the output is to add a *negative resistance* to the output to hold it at its current state when not driven. This negative resistance consists of a weak feedback inverter that can be overwhelmed by a stronger pullup or pulldown from the C-element. This circuit is shown in Figure A.3. In addition to providing feedback to the output node, the negative resistance stage adds an inversion so that the C-element is now a non-inverting gate. A standard way to implement a weak feedback inverter is to use long gates to weaken the current drive capabilities of the

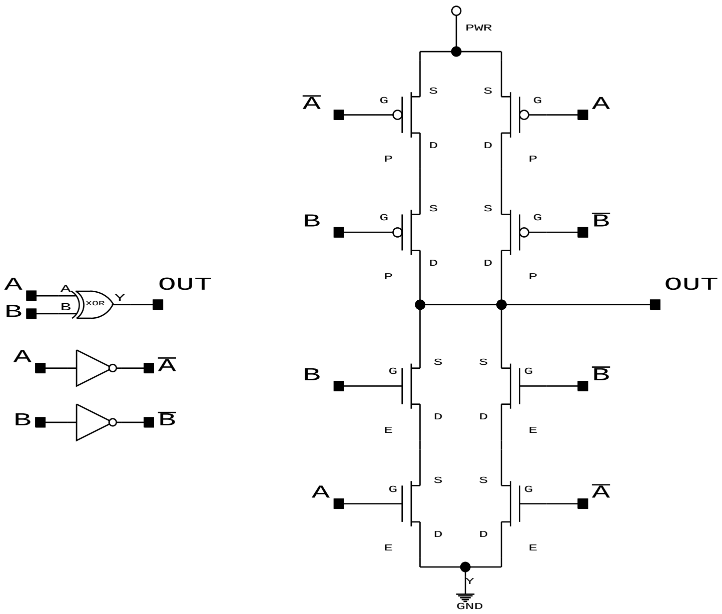

Figure A.1: A Merge Module Implemented as an XOR

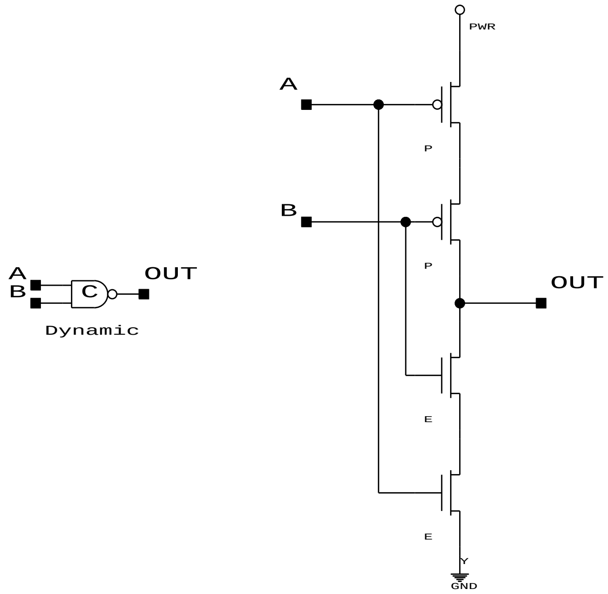

Figure A.2: A Dynamic C-element

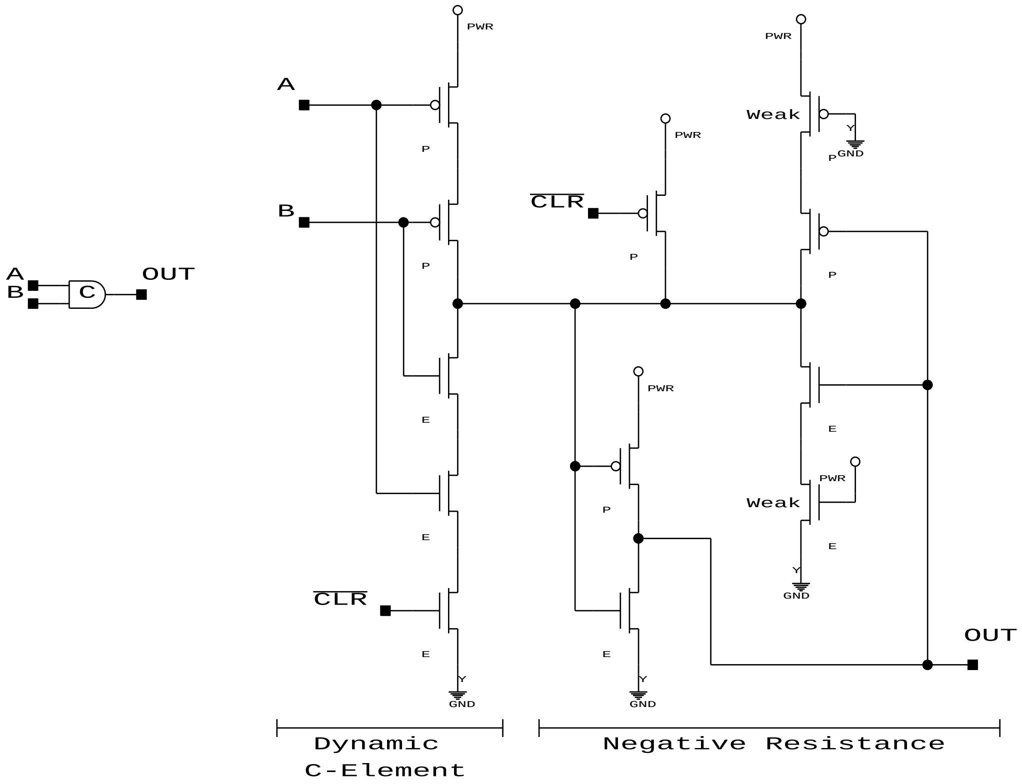

Figure A.3: A Static C-element Using Negative Resistance

inverter. This has the unfortunate side effect of increasing the capacitive load on the output node because it now has to drive long gates with more area. An alternative, as shown in Figure A.3, is to make the inverters that are driven be of minimum size. The transistors marked as *weak* on the schematic, implemented as long narrow devices in CMOS, in series with these minimum size transistors limit the drive of the entire stack. The combination is a weak feedback inverter that requires the output node to drive only a pair of minimum sized transistors.

In addition to the negative resistance used to make the C-element static, the circuit in Figure A.3 adds the appropriate transistors to clear the C-element. A low signal on the $\overline{clr}$ line will force the output of the C-element to zero.

Another method for making a static C-element is shown in Figure A.4. This circuit, similar to a majority gate, uses a passive feedback network that applies feedback to the output node only in the case that the inputs are at different values. Rather than overwhelming a negative resistance, the output node enables a feedback path to keep the output at its current value. If that value is one, for example, then the lower feedback path is enabled which

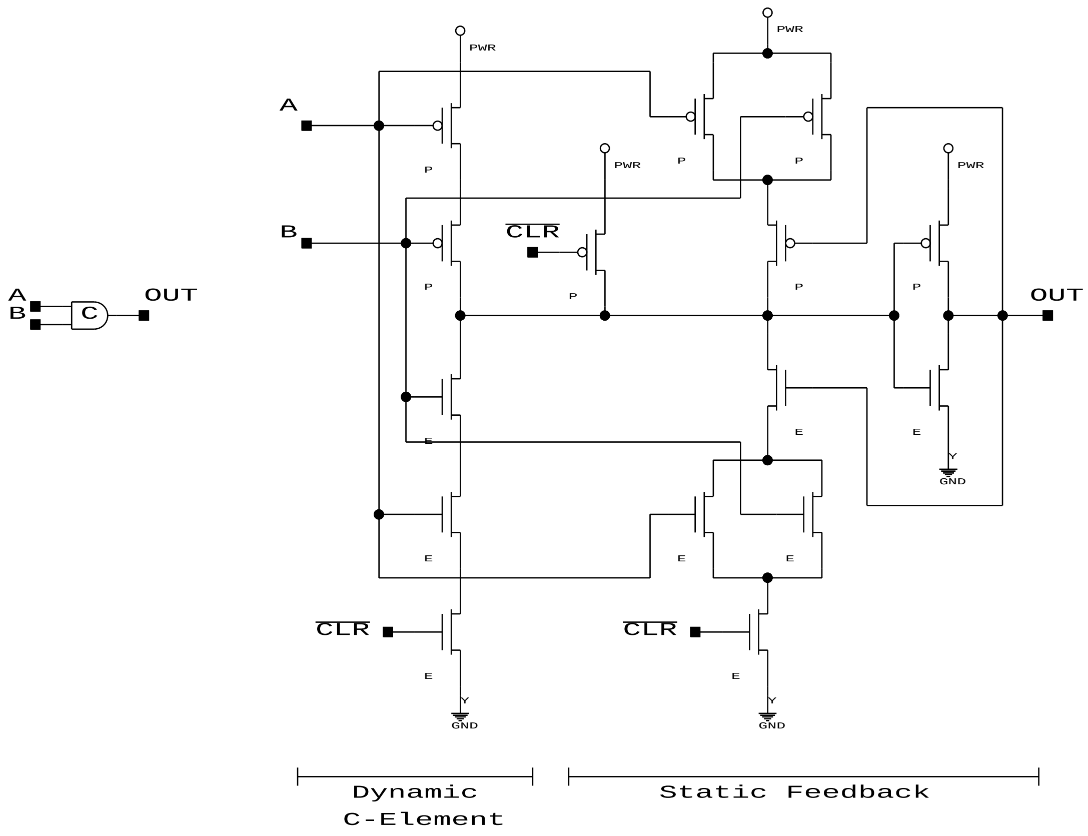

Figure A.4: A Static C-element Using Static Feedback

holds the internal node to zero as long as one of the inputs remains at one. This circuit also includes clearing transistors that force the output of the gate low on a $\overline{clr}$ signal.

A.1.3 Select

A transition select module can be implemented as shown in Figure A.5. The state-holding circuits used here are simple latches. The *G* input is the gate input to the latch. When the gate input is active, the latch is transparent, passing the *D* input through to the *Q* output. When the gate input is inactive, the latch holds its current state. A clear signal is provided to force the latch into a state where both its outputs are low.

Notice that the latch for the *outf* false output has an inverted gate input. That is, the latch is active when the input is low. This ensures that, as long as the bundling constraint on the *sel* input is met, only one of the two latches is transparent when the input transition occurs. The XOR gates cause a transition at their outputs whenever they receive a transition

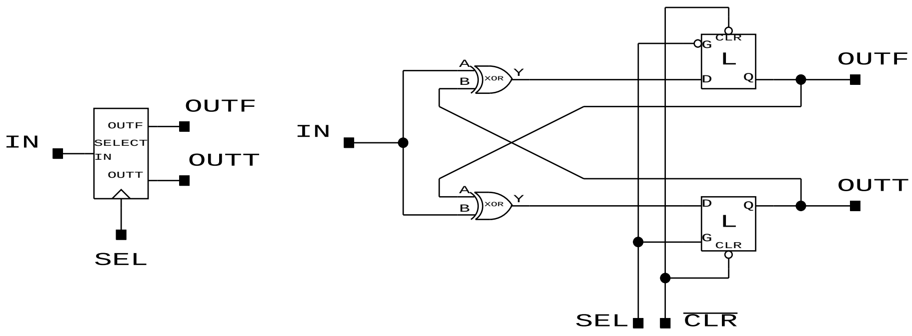

Figure A.5: A Transition Select Module

on either input. In this circuit, the XORs produce a transition at the input to both latches in response to a transition at the *in* transition input of the selector. The latch that is selected by the *sel* signal to be transparent passes this transition through to the appropriate output. The change on the output causes a second transition on the input of the other, opaque, latch. This effectively removes the effect of the previous transition on that latch so that, if the *sel* signal changes before the next input transition, there will be no change on that previously unselected output.

Note that although the bundling constraint only specifies that the *sel* signal be stable before the transition arrives, that there is another timing constraint caused by this implementation of a transition selector. This second constraint is that the *sel* signal must not change and the input transition must not occur after an output from the selector until that output change has had a chance to pass through the other XOR gate and take the transition away from the other latch's input. In practice this is rarely a problem because the *sel* signal is not allowed to change until after an output from the selector, and the circuitry that senses this event and decides to make a change to the *sel* signal rarely takes less time than a single XOR gate.

A.1.4 Q-elements

In situations where the bundling constraint on the *sel* signal cannot be met, a different form of selector is required. This module, called a Q-select, looks to the designer just like a transition selector but with no bundling constraint on the *sel* signal. That signal is now free to change at any time.

Qflop

The basic circuit that allows a Q-select to be built is called a Q-flop [79]. A Q-flop is a flip-flop that signals when its outputs are stable. That is, it samples its *D* input after some request signal, and produces an acknowledge when the output has settled in one or the other state sufficiently that further changes to the *D* input will not affect the state of the output. This requires that the circuit have some way to detect when the circuit has flipped or flopped far enough that it will not change its mind. The chief problem is that of metastable states [16]. If the input is changing at the same time that the circuit is trying to sample that input, the flip-flop could get stuck in a metastable state halfway between the voltage representing a logic "1" and that representing a logic "0." This is analogous to flipping a coin. The stable states of the coin after being flipped are heads and tails, but occasionally the coin will land on its third intermediate state: on edge. The flip-flop may linger in this metastable state, or oscillate around this metastabile state, for an unbounded, but finite, time. The goal of the *resolver* portion of a Q-flop is to determine that the flip-flop is far enough away from this metastable state that it will settle solidly onto "heads" or "tails" and stay there.

One circuit design for a Q-flop resolver is shown in Figure A.6. This circuit was simulated, fabricated, and tested as a part of the "Parts-R-Us" test chip [10] and consists of an input section on the left and a comparator on the right to sense when the input value has been sampled correctly. The input stage is controlled by two signals *C1* and *C2* which are generated from a request. If there is no active request then both *C1* and *C2* are low which will force the inverter between *A* and *B* to "eat its own tail." This will settle the voltage at both *A* and *B* to approximately half of the voltage swing of the inverter. This value also appears at the input of the enabled inverter enabled on *C2*. The input section will eventually form a flip flop from these two inverters which is now poised right at the metastable state because of the intermediate voltage at the inputs to the inverters. When a request to sample occurs, the *C1* signal goes high, which routes the current *D* signal through the *AB* inverter to the input of the *C2* inverter. Then *C2* goes high to make the two inverters into a flip flop which latches the current *D* value, and also decouples the input from the flip flop. Actually *C1* and *C2* can be the same signal as long as *C1* changes before, or at least not after, *C2*. In this case the value that is being sampled is actually the value stored on the dynamic node on the output of the enabled inverter at the input.

The comparator section of this circuit needs to ensure that the flip-flop in the input section has settled into a stable state. This is implemented using a mutual exclusion element like the circuit used in an arbiter. This comparator uses cross-coupled pass transistors to sense the outputs of the input flip-flop. Neither of these two pass transistors will turn on and let a signal pass until the voltage levels at their gates differ by more than a transistor threshold. This is a good indication that the flip-flop has exited metastability. Recall that when the Q-flop is not sampling, the voltage at *A* and *B* will be the same. Until these voltages differ,

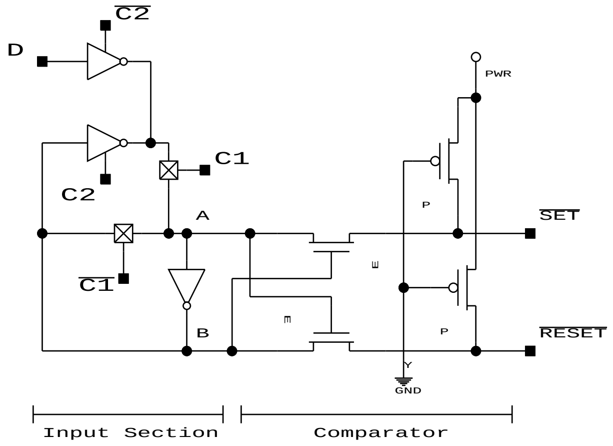

Figure A.6: A Q-Flop Resolver Circuit

both the $\overline{set}$ and $\overline{reset}$ outputs will be pulled high by the weak p-type pull-up transistors. When the input flip-flop has settled and one of the pass transistors turns on, one of these signals will be pulled to ground and thus send a signal to the output latch of the Q-flop.

Another circuit for a Q-flop resolver that has been fabricated and tested is shown in Figure A.7. This version is used as part of the Q-select used in the routing chip and memory control chip examples from Chapter 6. The description of its operation is taken from [88].

The circuit consists of five parts that may be identified in the figure. First, there is a flip-flop composed of transistors *M1, M2, M3*, and *M4* at the center of the figure. Second there is a power up network built from transistors *M7, M8, M9, M11*, and *M12* at the top of the figure. Third there are two threshold detectors. One of them, whose output is $\overline{R}$, is made with transistors *M13* and *M15*; the other, whose output is $\overline{S}$ if made with transistors *M14* and *M16*. Fourth there are several discharge transistors, *M17, M18, M19*, and *M20*, whose gates are connected to $\overline{C}$; these serve to return the nodes to which they are connected to ground between uses of the resolver. Finally, there are the master clear transistors, *M5, M6, M21, M22, M23*, and *M24*. If the "clock", $\overline{C}$, should be active during master clear, as can happen in a two-phase version of this device, these master clear transistors serve to force the flip-flop into the reset state, driving $\overline{R}$ low.

Operation of the device is easy to understand. When the signal called $\overline{C}$ is high, the flip-flop is unpowered, and most nodes are forced to ground by the discharge transistors. The output threshold detectors will have both their inputs low, and thus will permit the signals $\overline{S}$ and $\overline{R}$ to go high, i.e. inactive. When the signal called $\overline{C}$ goes low, the flip-flop is powered up through the network at the top of the figure. The data values received from the inputs D and $\overline{D}$ imbalance this network, providing for more rapid delivery of charge to either node 1 or to node 2 depending on the value of the data input. Eventually the flip-flop

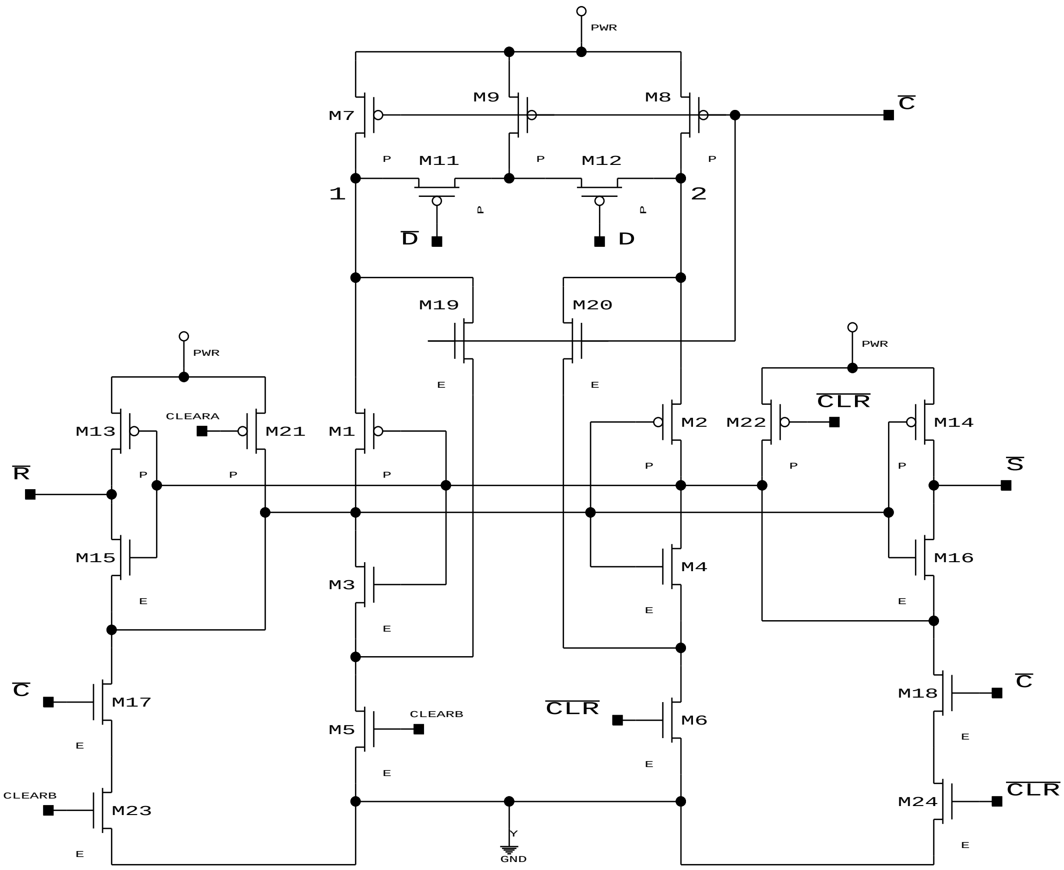

Figure A.7: Another Q-Flop Resolver Circuit

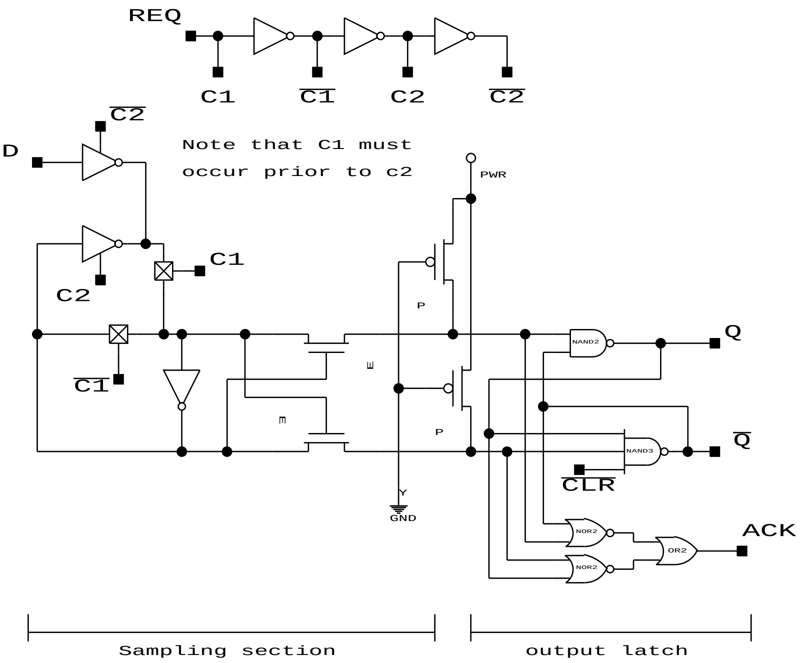

Figure A.8: A Four Phase Q-Flop

takes on one or the other of its two possible states, depending on this data input value. When the flip-flop has sufficiently flipped or flopped, one of the two threshold detectors will be activated, causing either the signal $\overline{R}$ or $\overline{S}$, but never both, to go low.

Still another form of Q-flop resolver circuit, this one designed by the inventors of the Q-flop, can be found in [79]. Any one of these three Q-flop resolver circuits may be used to build a complete four-phase Q-flop by adding an output latch and a circuit to generate the acknowledge signal. For example, a Q-flop built using the Q-flop resolver from Figure A.6 is shown in Figure A.8. In this circuit, the output stage is a simple set-reset flip-flop with a *clear* input. When the resolver stage has sampled the input it will pull one of $\overline{set}$ or $\overline{reset}$ low, causing the output stage to set or reset. When one of the set or reset signals has gone low, and the output flip-flop agrees with the value, then the Q-flop acknowledge is asserted. In this case, the *C1* and *C2* signals required by the input section are generated from the *Req* signal to the Q-flop.

Two of these four-phase Q-flops may be combined to make a two-phase Q-flop as shown in Figure A.9. This circuit operates by alternately sampling into the top and bottom four-phase Q-flops. When the request to sample arrives, one of the four-phase Q-flops starts sampling, and the other, having been used the last time, resets its input stage. When the half that is sampling has a valid output and the half that is clearing is finished clearing, the C-element switches the output to the sampled side and announces an acknowledge. Thus

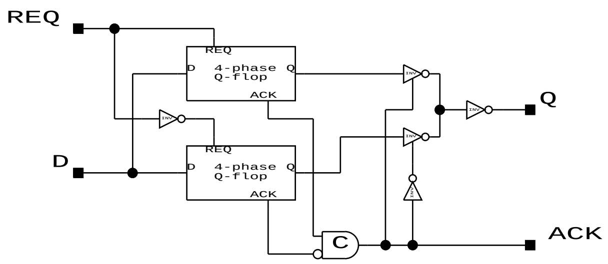

Figure A.9: A Two Phase Q-Flop

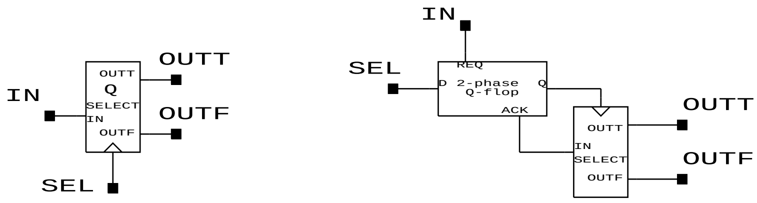

Figure A.10: A Q-select Module

the two phase Q-flop will sample a new input on every transition of the control input.

Note that this circuit could also have been implemented simply by attaching a two-phase to four-phase converter to the interface of single four-phase Q-flop at the expense of some speed of operation.

Q-Select

Once a two-phase Q-flop is available, it can be used to implement a Q-select module. This module has an external interface that matches the standard transition select module from the previous section, but requires no bundling constraint on its *sel* input. This circuit is built, as shown in Figure A.10, by using a Q-flop to sample the *sel* input whenever an input request is seen. When the sampled value of the *sel* signal is valid, indicated by the Q-flop acknowledge, the transition select module chooses where to route the input transition. Note that the bundling constraint required by the transition select module is met because the acknowledge from the Q-flop will not be generated until after the *Q* output is stable.

A.1.5 Toggle

A toggle module can be built from a pair of latches as shown in Figure A.11. In this circuit, the *out0* output will receive the first output transition after a *clear* signal. Note that the bottom latch has a positive *D* input and a negative *G* latch input. The top latch, on the

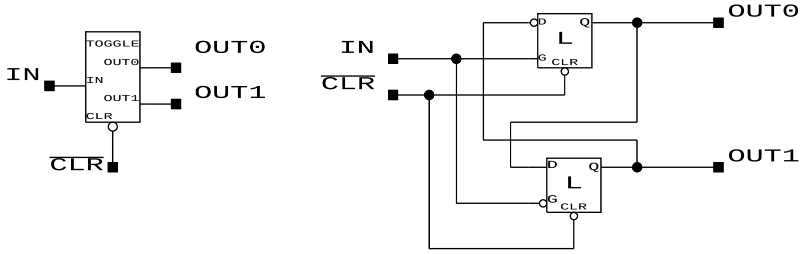

Figure A.11: A Transition Toggle Module

other hand, uses a positive *G* latch signal and a negative *D* data input. The combination of these two latch types implements a toggle module by letting each transition of the *In* signal change which latch is transparent and which latch is opaque.

After a *clear* signal, all inputs and outputs of the toggle will be low. This means that the top latch will be opaque, storing the current low value, and the bottom latch will be transparent passing the *out0* low signal to the *out1* output. When the input signal makes a transition to a high level, the top latch turns transparent allowing the inverted signal from *out1* to change the value of *out0*, and the bottom latch changes to opaque storing its current low value and ignoring the high value that is now generated by *out0*.

Obviously there are several timing constraints involved in this circuit. If the two latches are ever both transparent for any length of time the circuit will oscillate — a most undesirable behavior. Careful timing analysis must be used in designing this circuit, or more a more conservative circuit design can be used that implements a non-overlapping signal to the *G* inputs of the latches to ensure that they are never transparent at the same time.

A.1.6 Call

A Call module remembers which client made a request by using the storage inherent in a C-element as shown in Figure A.12. When a request is made on *R1* or *R2*, a transition is generated on the *Rs* subroutine request, and one input of the appropriate *A1* or *A2* C-element is changed. When the subroutine acknowledge *As* occurs, it changes the other input of the C-element to match, thus causing a transition on the correct acknowledge signal. The subroutine acknowledge also resets the other C-element to the state it was in before the transaction. The Call module is now ready to accept another request on either *R1* or *R2*. Recall that requests to a Call module must be mutually exclusive. An entire request-acknowledge transaction must occur before the next transaction may begin. Call modules with more client interfaces can easily be built by cascading multiple two-input Call modules.

An external timing constraint imposed by this implementation of a Call module is that

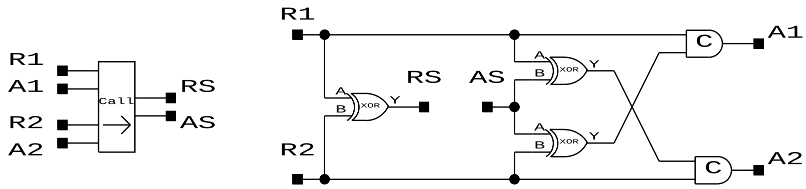

Figure A.12: A Transition Call Module

following a client transaction, a succeeding transaction cannot be initiated before the *As* subroutine acknowledge has reset both internal C-elements. For example, assume that a request occurs on *R1*. This causes a transition on *Rs*, makes a transition on one input of the top C-element, and, through the top XOR, makes a transition at the bottom C-element too. When the *As* subroutine acknowledge arrives, it will cause a transition on the *A1* signal by making a transition at the other input of the top C-element, but also "take away" the transition at the bottom C-element that was caused by *R1*. A transition on *R2* must not occur until this action of removing the transition from the bottom C-element has occurred or an incorrect acknowledge will be generated. In practice, this is usually not a problem because the circuitry to generate a new request will be slower than the XOR gate internal to the Call module.

A.1.7 Arbiter

A transition arbiter can be used to guarantee mutually exclusive access to some shared resource. To implement this guarantee, a mutual exclusion circuit similar to the circuit used in one version of the Q-flop resolver is used. This circuit is shown in Figure A.13. The circuit consists of a set-reset flip-flop followed by a comparator to sense when the flip-flop has settled into a stable state. The action of the comparator is the same as in the Q-flop resolver. The outputs of the comparator, *CA* and *CB*, are pulled high when the flip-flop is metastable and the outputs of the flip-flop are at the same values. When the outputs of the flip-flop differ by more than a transistor threshold, the pass transistors are turned on and one, and only one, of the comparitor outputs is pulled low. This will cause exactly one of the mutually exclusive output signals *MA* and *MB* to go high through the inverters.

This mutual exclusion (mutex) circuit can be used as the heart of a transition arbiter as shown in Figure A.14. When there is no request active, both outputs from the mutex circuit are low and so both input latches are transparent. Requests from both *R* wires go through the latches and the XOR gates to raise the inputs to the mutex circuit. The mutex circuit will choose exactly one of the request signals and raise the corresponding mutually exclusive output. This output will change the input latch of the chosen request to opaque, and the output latch of that request to transparent thus allowing the request to feed through and become a grant signal. When the *D1* or *D2* done transition is received, it causes the

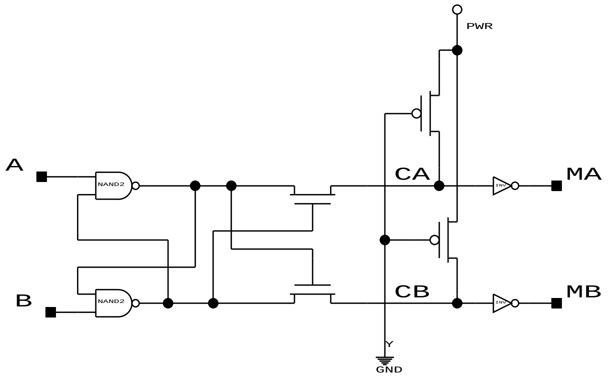

Figure A.13: A Mutual Exclusion Element

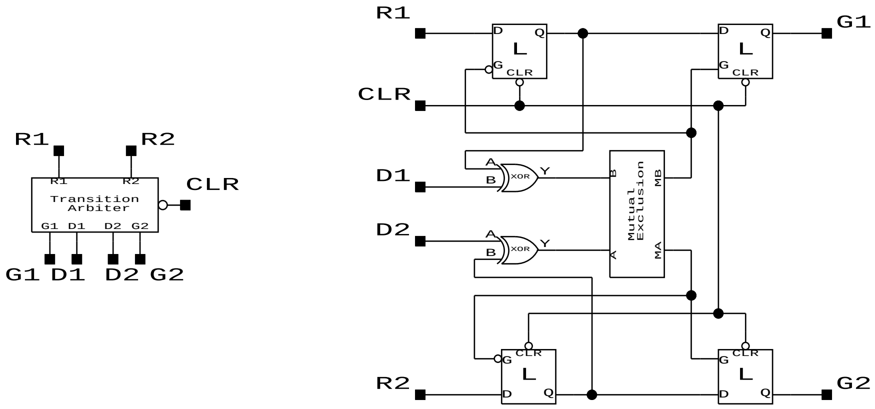

Figure A.14: A Transition Arbiter Module

input to the mutex element to go low, which frees the arbiter to make another choice of requesters.

Note that the latch on the *R1* and *R2* inputs allows the requester to make another request as soon as it receives a grant. In particular, it can make a new request after receiving a grant and before producing a done signal. Once the done signal is received by the arbiter, the new grant will be allowed to influence the mutex element immediately.

A.1.8 Protocol Converters

Although the current compiler uses only two-phase signalling for control modules, independently designed modules may use different protocols and require converters to communicate

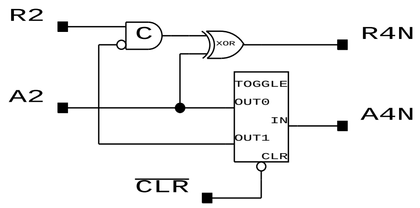

Figure A.15: Two-Phase to Four-Phase Narrow Converter

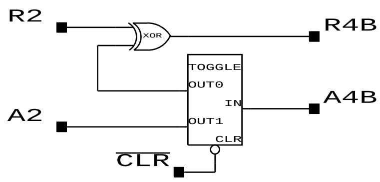

Figure A.16: Two-Phase to Four-Phase Broad Converter

to the two-phase modules. Circuits designed by hand without using the compiler can also make use of these converters. Recall that there are two forms of four-phase protocol: narrow and broad conventions. Converters are shown in Figures A.15 through A.18 for all possible conversions.

A.2 Data Path Modules

A.2.1 Enable

Enable modules are used to drive bundled data onto a shared bus in response to transition control signals. After receiving an enable request *Ren*, the module will enable its input data bundle on the output bus. The enable acknowledge, *Aen*, is generated after the output data bundle is valid at all the receivers connected to the bus. When a *Rdis* disable request is received, the data outputs are placed in a high impedance state and a disable acknowledge,

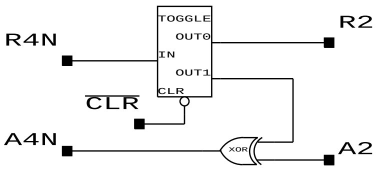

Figure A.17: A Four-Phase Narrow to Two-Phase Converter

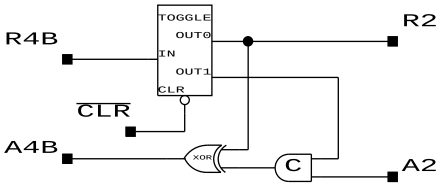

Figure A.18: A Four-Phase Broad to Two-Phase Converter

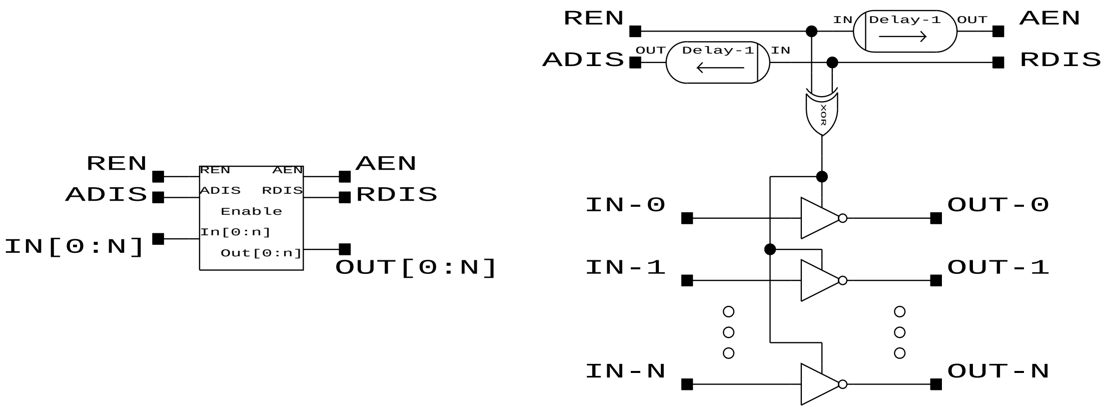

Figure A.19: An Enable Module

Adis, is generated. Correct use of this module requires that enable and disable requests alternate.

One circuit for an Enable module is shown in Figure A.19. In this circuit, tri-state inverters are used to drive the bundled data bits onto the Enable module's outputs. These tri-state inverters are controlled by the XOR gate such that a transition on the *Ren* wire will enable the inverters and drive the outputs. After some bundling delay, the transition is seen at *Aen* which signals that the data are correctly driven. When a transition occurs on *Rdis*, the output of the XOR goes low and the tri-state inverters no longer drive the output. After another delay, *Adis* reports that the outputs are, in fact, no longer driven.

While the delays in Figure A.19 are not difficult to determine, a slightly different circuit can be used to reduce the need for adding explicit delays. Shown in Figure A.20, this circuit holds the *Ren* transition at the input of the latch until the high signal at the output of the XOR has propagated to all of the tri-state inverters. Once this occurs, the latch is made transparent and the transition is allowed to become the *Aen* transition. This circuit requires that the physical layout be taken into account to assure that the output of the XOR has propagated to all of the tri-state inverters before arriving at the *G* input of the latch. The latch on the *Rdis—Adis* line operates in the same way. In each of these circuits, explicit

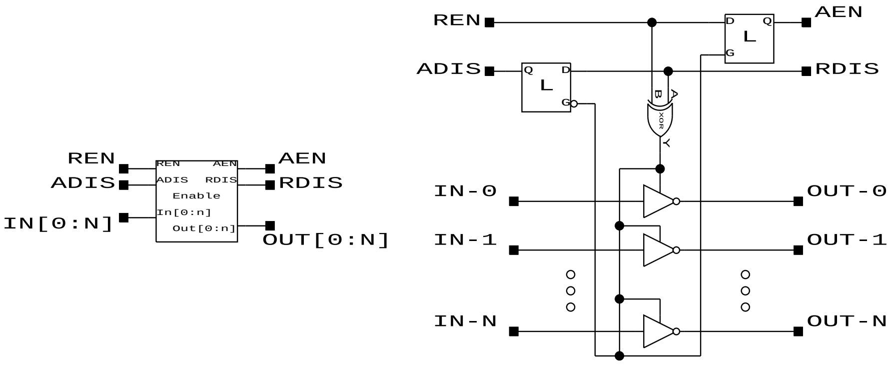

Figure A.20: An Enable Module using Latches

delay may need to be added to the *Aen* and *Adis* wires to guarantee that the data delivery bundling constraint has been met. These control signals must not be seen at the receiving circuit before the data are valid at the receiver.

A.2.2 Register and Latch

Register and Latch modules are used to capture and store bundled data in response to transition control signals [94]. The bundling constraint on the environment requires that the data be stable at the input to the Latch or Register before the request is received. An acknowledge transition is generated when the data are latched and, according to the bundling constraint on the module, the output data are valid at all recipients of the data.

An example of a possible implementation of a Latch module is shown in Figure A.21. This is a Latch which uses tri-state inverters to latch a bundled data signal in response to transition control signals. The control signals for this module are called *C* for *capture* and *P* for *pass*. After initialization, when the two control signals are at the same level, the latch is transparent from input to output. The first transition occurs on the *C* signal which causes a loop to form either at the top or bottom of the circuit in Figure A.21 to hold the value of the data signal. When a transition occurs on the *P* line, the loop is broken and the input is once again connected to the output. A Latch of this form, where the first signal transition must be on *C* is known as a *normally transparent* transition latch.

Variations of this basic latch are easily imagined. Rather than be transparent when the input signals are at the same level, the latch could be opaque in that initial state with suitable mixing of the control signals. This circuit is shown in Figure A.22. In this case, the first signal to the latch must be a *P* transition to allow the new input value through the latch, followed by a *C* transition to form an inverter loop and capture that value. This is known

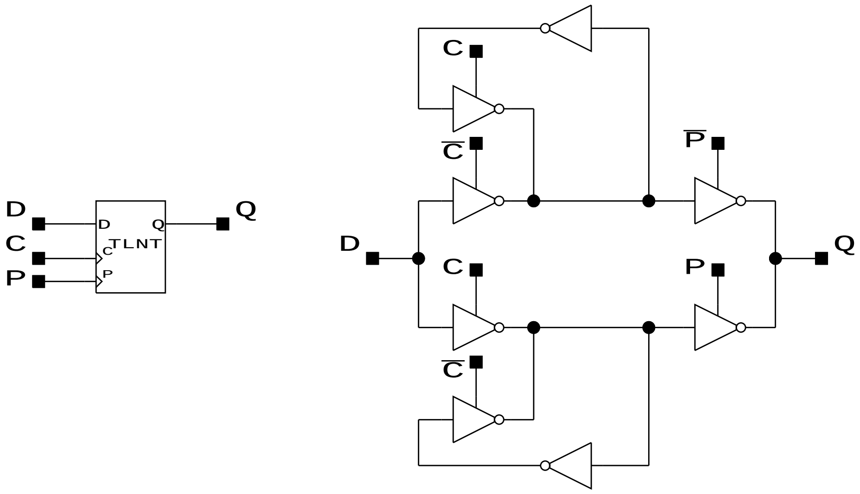

Figure A.21: A Normally Transparent Latch

as a *normally opaque* transition latch.

These transition latches can be connected in groups to form transition registers. These registers are simply groups of latches with their *C* and *P* signals connected in parallel. They can also be configured to present standard request and acknowledge signals at the Latch interface. This form of the latch is shown in Figure A.23. This Latch is normally opaque. When a request transition is received by the Latch, it becomes transparent. Note that the *R* wire connects to the *P* inputs of the Latch and thus passes the data through the latch. After some bundling delay, this same transition controls the *C* wires and captures that data in the now opaque Latch. This transition is sent to the *Ack* signal of the Latch to report that the current data have been captured in the Latch.

Finally, a master-slave Register can be implemented that uses the same transition control signals. This circuit is shown in Figure A.24. This circuit uses a trick similar to that used in the request-acknowledge Latch of passing a transition around to the internal control points of the Register. In this case, data are first captured in a loop on the left of the circuit when the request transition is seen by the *A* controls of the Register. This value is then transmitted to the output when the transition reaches the *B* wires, and finally, captured in a loop on the right of the circuit through the *C* signals.

A.2.3 Boolean Variable

A Latch can use a special form when the value is a Boolean variable. These Boolean variables have separate interfaces to set and reset the value of the variable. These interfaces are standard request-acknowledge transition interfaces. One method for implementing a Boolean variable is in the same way as other variables. This circuit is shown in Figure A.25.

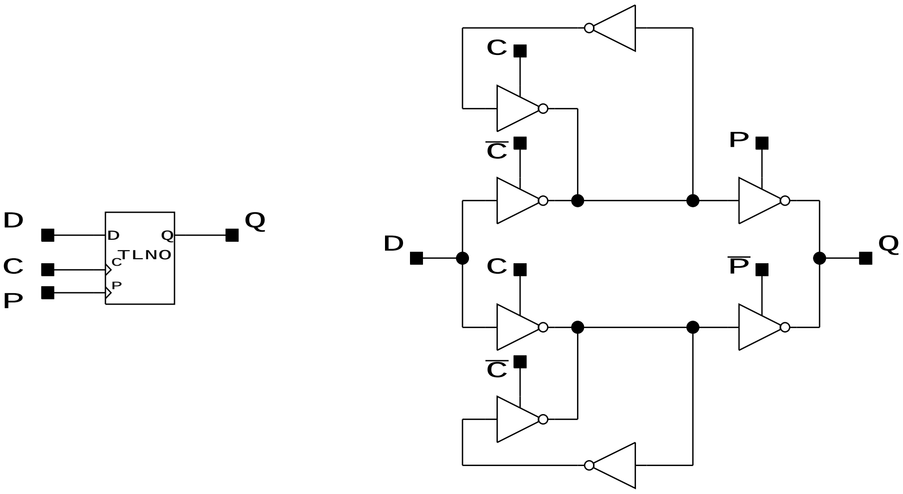

Figure A.22: A Normally Opaque Latch

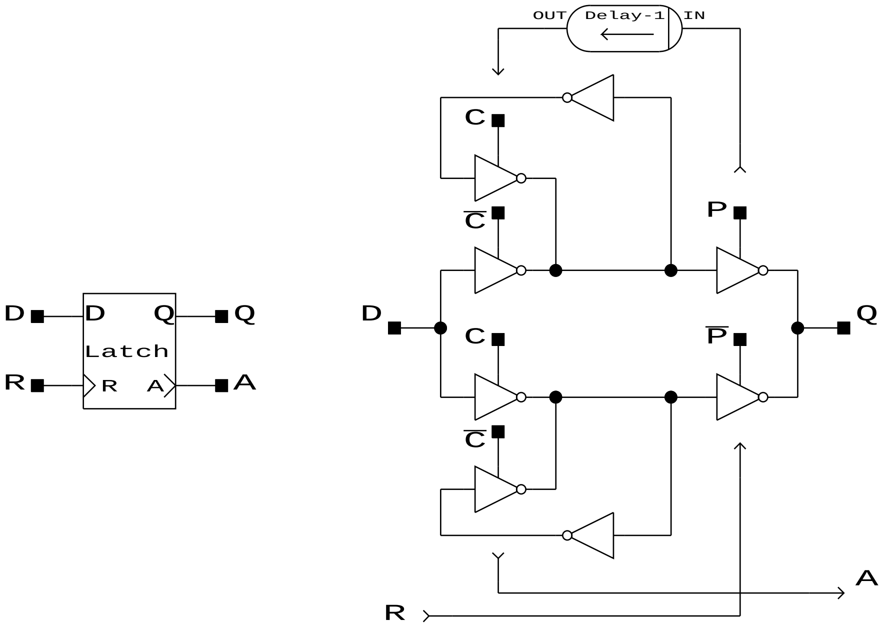

Figure A.23: A Request-Acknowledge Latch

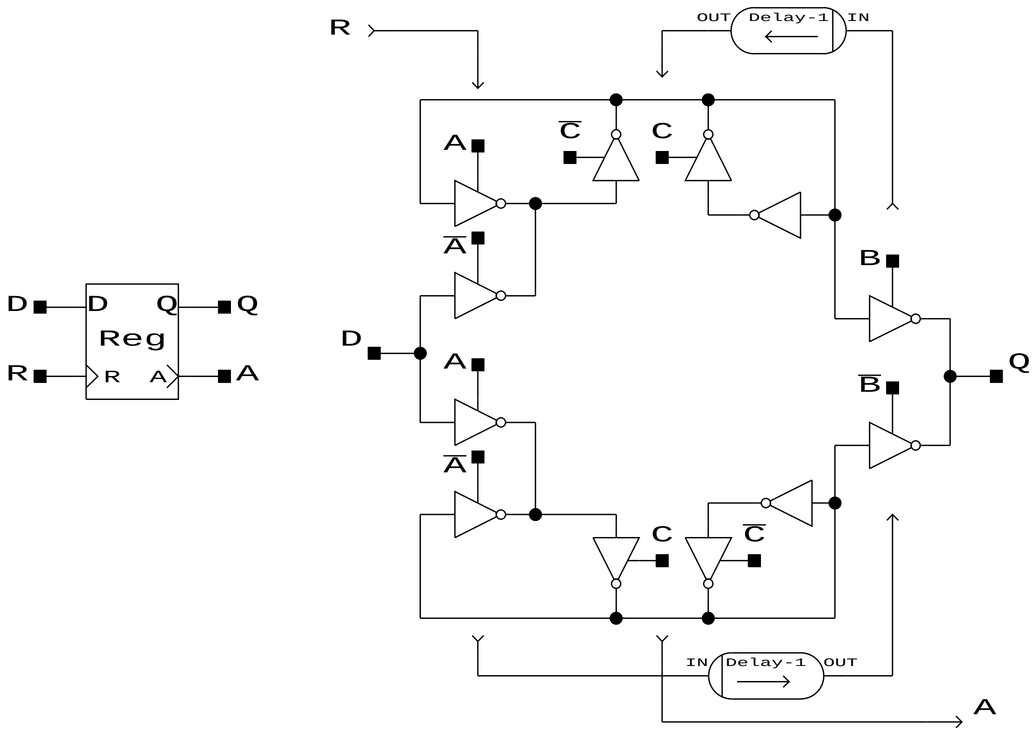

Figure A.24: A Transition Register

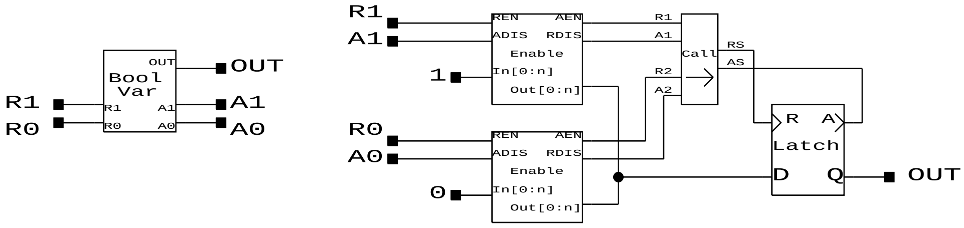

Figure A.25: A Boolean Variable Using Enable Modules

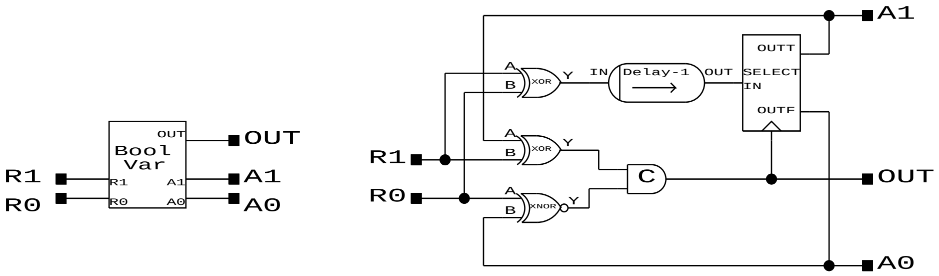

Figure A.26: A Boolean Variable Using a Select Module

A standard Latch is used to hold the value of the Boolean variable. This Latch is either written with a 0 to reset the variable, or written with a 1 to set the variable. These values are presented to the input of the Latch through Enable modules.

Another circuit that implements a Boolean variable is shown in Figure A.26. This circuit does not use a standard Latch and instead uses a C-element to hold the value of the Boolean variable. The Select module is used to steer the request transition to the appropriate output depending on the value of the variable. This acknowledge also resets the inputs of the C-element to allow the next request to set or reset the value. The delay shown in the circuit is the bundling delay for the Select module. This assures that the value of the variable is stable before the transition is seen at the input to the selector.

Another method of steering the acknowledge transition is shown in Figure A.27. In this circuit, the latches are opaque until the Boolean variable reaches the correct state. When the variable is set, for example, the upper Latch becomes transparent and allows the *R1* transition become the *A1* acknowledge and also reset the input to the C-element.

One final optimization is shown in Figure A.28 where the *R1* and *R2* transitions are simply delayed long enough to guarantee that the C-element has changed to the correct state before acknowledging the transaction.

A Boolean variable with a different interface is shown in Figure A.29. In this version, the value of the Boolean variable is tested using transitions rather than being a bundled signal. A transition on the *Rt* line will test the value of the variable and cause a transition

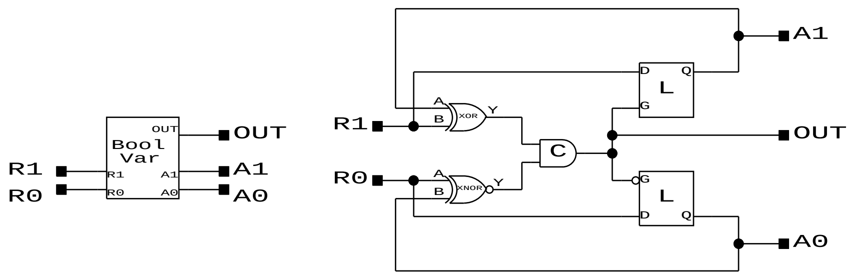

Figure A.27: A Boolean Variable Using Latches

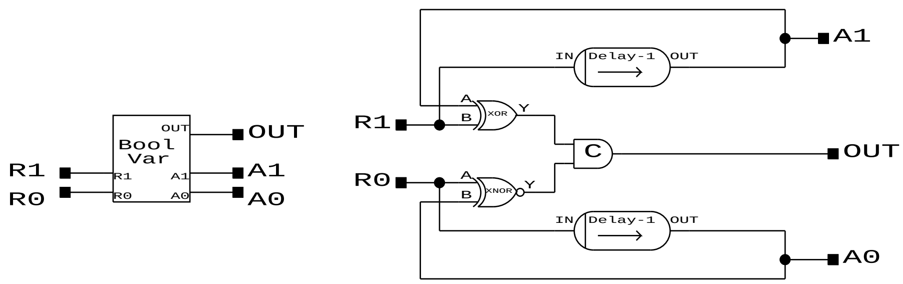

Figure A.28: A Boolean Variable Using Delays

on either the *At0* or *At1* line depending on the current value. A transition on the *Rc* line will complement the value of the variable. The implementation is simply a Select module and a bundling delay for the *sel* signal to that select module. In the initial state the *Rc* signal is low and the Boolean variable is *false*. Every time a transition occurs on the *Rc* line, the value of the *Rc* signal changes, and so does the value of the Boolean variable. The delay simply waits until the value of the *Rc* signal has been seen by the Select module before acknowledging that the value has been complemented.

This Boolean toggle variable can be used to build a form of Boolean variable that uses

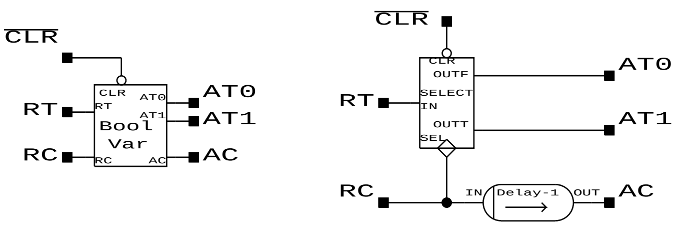

Figure A.29: A Boolean Toggle Variable

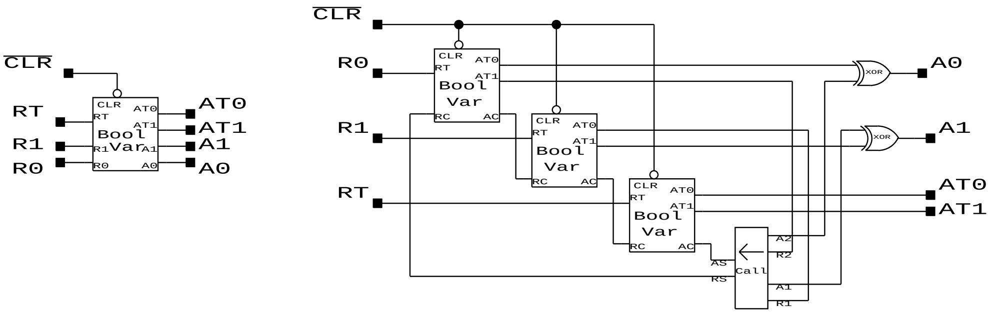

Figure A.30: A Boolean Test and Set Variable

request and acknowledge signals to set and reset the value of the variable and also uses transitions to test the variable's value. This circuit is shown in Figure A.30.

A.2.4 Carry Completion Adder

A carry completion sensing adder can sense when the internal carry chain of the adder is complete, and thus, when the addition is complete. The carry is propagated using two separate signals: *Cout* to indicate a carry, and *Dout* to indicate the absence of a carry. Carry completion adders use a four-phase protocol based on the *En* signal that enables the adder to execute. When the *En* enable signal is not asserted, all the *Cout* and *Dout* signals are set low. When *En* is asserted, if any individual bit can generate a carry it asserts *Cout* to the next stage. If it can kill a carry it asserts *Dout*, otherwise the bit must wait for the arrival of a *Cin* or *Din* from the previous stage before deciding. A simple *OR* gate on the *Cout* and *Dout* signals from each bit can tell when that bit has finished computing the carry to the next bit. When all the bits have computed their carries, the addition is complete. It should never be the case that both *Cout* and *Dout* are asserted at the same time in any bit.

One circuit for a single bit of a carry completion sensing adder is shown in Figure A.31. Note that the sum is computed independent of the *En* signal. The two carry signals are gated with *En* so that they are reset when *En* is low and will produce signals when *En* is high. There are obviously many different ways to implement this form of adder depending on the technology used.

If the desired function is to increment the argument by one rather than by an arbitrary amount, the simpler circuit of Figure A.32 can be used.

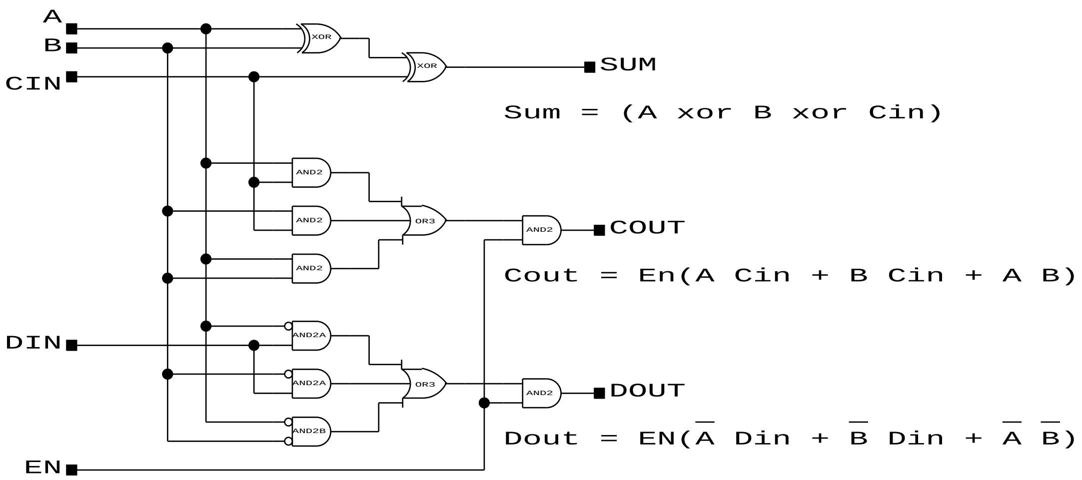

Figure A.31: A Carry Completion Sensing Adder Bit

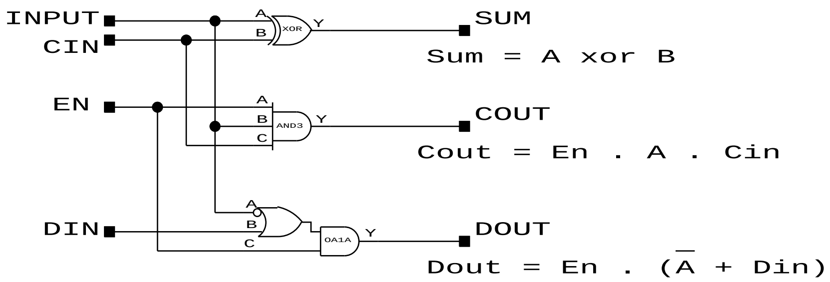

Figure A.32: A Carry Completion Sensing Incrementer Bit

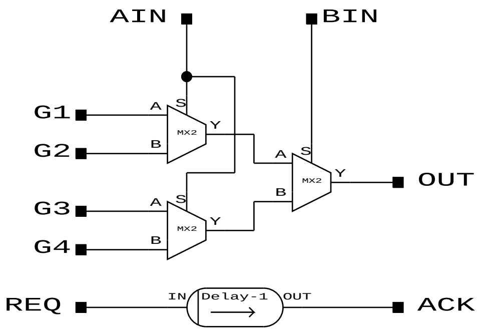

Figure A.33: A Bundled Logic Generator

A.2.5 Logic Generator

The carry completion sensing adder in Chapter 4 is combined with a logic generator to produce an ALU. The logic unit implemented for this circuit, and used in the memory controller from Chapter 6, is shown in Figure A.33. This is simply a bundled logic generator constructed using two-way multiplexers. The desired logic function is set on the *G* inputs with *A* and *B* used to select the appropriate bits. *A* and *B* are bundled signals with respect to the *Req* signal and the bundling delay is simply the worst-case time for a signal to pass through two two-way multiplexers.