Chapter 4

Target Circuit Structure

The goal of the translation system is to convert programs written in the source language, a dialect of OCCAM, into circuits. In this case, the target circuit is not an assembly of individual transistors, or even gates, but rather a collection of predefined data and control modules, wired together appropriately. The modules may be as simple as a single gate or as complex as a few hundred gates as required to implement some interesting or commonly used operation. This is a similar approach to that taken by design tools that use a set of standard cells as their primitive circuit elements. Circuits are constructed using these modules rather than simpler gates in part because the functions of the modules correspond closely to programming constructs, thus simplifying the translation, and in part because signaling conventions between modules allow reasoning about optimizations and correctness without being overwhelmed by the complexity of thousands of transistors. Most of the signalling to and from these modules is insensitive to delays in the wires or the components which allows simple reasoning about their connection. Because of this, the designer can reason about high level communication protocols rather than voltage levels on specific wires.

This module library traces its ancestry to Macromodules [19, 70], designed at Washington University in the late 1960's. Macromodules were designed to be "building blocks such as registers, adders, memories, control devices, etc., from which it is possible for the electronically naive to construct arbitrarily large and complex computers that work [70]." Macromodule elements were constructed as boxes, typically containing 80 MECL-II integrated circuit packages, that were plugged into a large frame that supplied power and cooling. The boxes were physically wired together using data cables connecting to the faceplates of the boxes. While the physical scale of the resulting circuits was rather greater than that of the VLSI implementation of the target circuits presented here, the philosophy was quite similar. Modules larger than a single gate were implemented so that common circuit structures could be connected together without becoming overwhelmed with the large number of gates needed to implement those structures. Careful design of the interfaces of the circuit elements enabled large circuits to be constructed based only on their functionality. A small number of other cell sets for asynchronous design similar to the macromodules approach have also been explored [18, 74].

The asynchronous nature of the desired circuits requires that special modules be implemented in addition to standard digital logic gates. The circuit modules used here include special modules like those described in [10, 94, 95]. The modules form a library of standard cells implemented using the MOSIS scalable CMOS design rules. Once the netlist for the desired circuit is generated, whether by hand or by automatic translation, these modules may be assembled into a circuit using a layout editor such as Magic [71], or used as input to a place and route system such as Fusion [3, 4] for automatic assembly.

4.1 Signaling Protocols

The correct functioning of an assembly of circuit modules into a larger system depends upon each of the modules adhering to a set of rules regarding signals passed between the modules. For an asynchronous system, these rules should make no assumptions about the timing of the signals and rely instead only on the sequence of signals. This allows circuit modules to operate at any speed, and communication to take as long or as short a time as required. This is also quite different from traditional synchronous signaling conventions where signal events occur at specific times and may remain asserted for specific time intervals. In asynchronous systems it is important only that the correct sequence of signals be maintained. The timing of these signals is an issue of performance that can be handled separately. Systems that will continue to function correctly regardless of delays in circuit elements or the wires that connect them are known as *delay-insensitive* [100].

The simplest form of asynchronous communication between two modules is a request-acknowledge transaction as shown in Figure 4.1. One module, the sender, sends a request event (*Req*) to another module, the receiver. Once the receiver has completed the requested action, it sends an acknowledge event (*Ack*) back to the sender to complete the transaction. In order to maintain this sequence of events, the communicating modules must obey the following rules:

RA-Rule 1 *The sender must not produce a new request event until the previous request event has been acknowledged.*

RA-Rule 2 *The receiver must not produce an acknowledge event unless it has received a request event.*

RA-Rule 3 *After initialization, the sender may produce a new request event.*

These rules define the operation of the modules which follows the common idea of passing a token of some sort back and forth between two participants. Imagine that a single

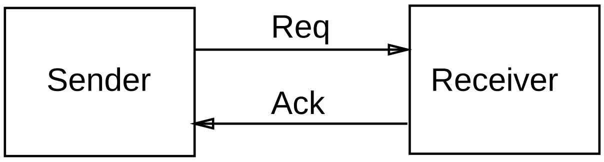

Figure 4.1: A Request-Acknowledge Interface

token is owned by the sending module. To issue a request event it passes that token to the receiver. When the receiver is finished with its processing it produces an acknowledge event by passing that token back to the sender. The sequence of events in this communication transaction is an alternating sequence of request and acknowledge events. The sequence of events in a communication transaction is called the protocol. In this case the protocol is simply for request and acknowledge to alternate, although in general a protocol may be much more complicated and involve many interface signals. Proper operation of a protocol requires the cooperation of more than one module; a single module cannot enforce all the rules for orderly communication. Notice that in the rules for this simple protocol RA-Rule 1 must be observed by the sender, and RA-Rule 2 observed by the receiver.

4.1.1 Transition Signaling

The simplest communication event that can be generated and sensed by digital circuit modules in MOS or TTL technology is a transition of the voltage on a wire. For signaling purposes the direction of the transition, either rising or falling, is not important, only the fact that wire has made a transition from one state to another is important. Taken as a primitive, both rising and falling transitions have the same meaning. A variety of protocols constructed from simple signal transitions can be used to pass information between modules. The protocol should be constructed to be sensitive only to the sequence of signal transitions and thus be insensitive to the relative timing of the transitions.

Using a transition as a basic event, a sender can issue a request by causing a transition on the request wire (*Req*), and the receiver can acknowledge by causing a transition on a second acknowledge wire (*Ack*). This form of signaling is called *transition*, *two-phase*, or non-return-to-zero signaling [82]. The term two-phase is used because there are two distinguishable states in each transaction: the state where the sender is active and the receiver has not yet responded, and the state where the receiver has responded and is waiting for the sender to become active again.

A common convention, and one that will be followed here, is to initialize the signal wires to the same state, always low in this case, so that in the communication phase with no active request the wires will be in the same state. This convention allows the two phases of the communication protocol to be distinguished by examining the levels of the *Req* and

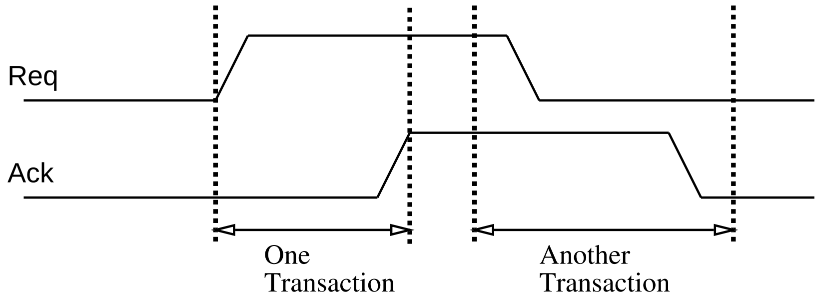

Figure 4.2: Transition Signaling Protocol

Ack wires. Figure 4.2 shows the transition signaling protocol as defined here. Note that because rising and falling transitions mean the same thing, the communication phase with no active request is represented by the state where both signal wires are at the same level, whether that level is high or low. The phase with an outstanding request that has not yet been acknowledged is detected when the signal wires are at different logic levels.

4.1.2 Four-Phase Signaling

Another common signaling protocol, called *four-phase* or *return-to-zero* signaling also uses transitions for primitive events but adds two more transitions to each transaction. The extra transitions return the *Req* and *Ack* wires to their initial state after every transaction. Again, it is common to initialize the signal wires to a low state. This means that after a complete four-phase transaction the signal wires will again both be low.

Figure 4.3 shows a complete four-phase transaction with each of its four distinct phases. The four phases are:

1. Quiescent: *Req* low, *Ack* low. This is the initial condition of the interface and indicates that no transaction is in progress.

2. Requesting: *Req* high, *Ack* low. The sender has issued a request to the sender, and the sender has not yet acknowledged.

3. Acknowledging: *Req* high, *Ack* high. The sender has acknowledged the request.

4. Clearing: *Req* low, *Ack* high. The requester has initiated the clearing phase of the protocol by dropping its request signal. The receiver has not yet responded by lowering its acknowledgment.

Both the transition and four-phase protocols are delay-insensitive and are thus very useful for communication between asynchronous modules. Which protocol is used for

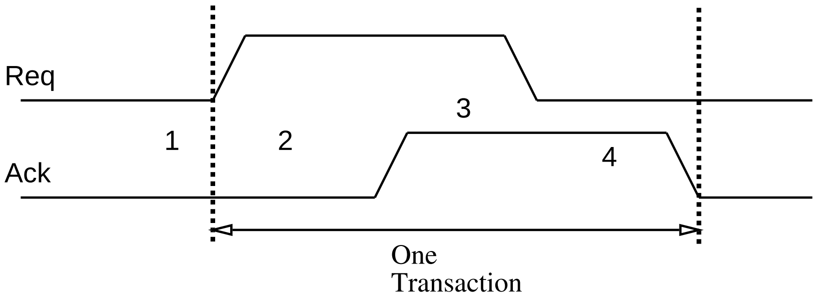

Figure 4.3: Four-Phase Signaling Protocol

individual modules will influence the design of the modules or the system, but is largely a matter of taste. Transition signaling can be faster because it requires only half of the transitions that four-phase requires. In addition to speed, less power is used when fewer transitions are required. However, transition signaling may require special circuits not normally available as packaged parts. This is not as severe a problem with VLSI as with designs using commercially available integrated circuits where four-phase designs are more common. Even in VLSI, however, four-phase modules are perhaps a better match for modules that deal with data rather than just control signals. Most control modules used as targets for translation in this work use transition signaling interfaces, although specification of individual modules will show precisely which protocol is being used.

4.2 Data Passing Protocols

In order to pass data between asynchronous modules more events than a single request-acknowledge pair need to be used. Two different events need to be distinguished to communicate a single bit: sending a 1 and sending a 0. There are a variety of ways to encode the value of data being communicated from module to module.

4.2.1 Delay-Insensitive Data Communication

One choice is to use two request-acknowledge pairs denoted as R_0/A_0 and R_1/A_1, one for each value of the bit being passed. A transaction on the R_0/A_0 signal pair would pass a 0 from sender to receiver, and a transaction on the R_1/A_1 pair would indicate a 1 being passed. This protocol is known as *four-wire* data signaling. The two signal pairs may use either transition signaling or four-phase signaling to implement their requests and acknowledgments, but in either case the requests must be mutually exclusive. A transaction may not start until the previous one, regardless of which of the two signal pairs it uses, has finished.

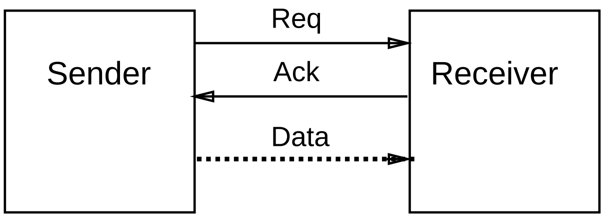

Figure 4.4: A Bundled Data Interface

A variant of the four-wire protocol uses a single acknowledgment wire for both values passed. The signal wires for this *three-wire* protocol are $R_0/R_1/A$. Requests on the R_0 or R_1 wires indicate a value to be passed, and a signal on the A wire indicates that some value has been received. Again, the protocol may specify transition signaling or four-phase signaling be used. This scheme has the advantage of using fewer wires per bit of data passed, but may require slightly more complex circuits than the four-wire protocol.

More than one bit of information can be communicated either by using one communication path serially, or by duplicating the communication wires for each bit and passing the bits of an entire word concurrently. Wires needed for these protocols are $4n$ and $3n$ for each protocol respectively passing an n bit word.

As a final optimization for passing data bits in parallel, two request wires may be used for each bit, and the acknowledgments combined into a single signal from the receiver for the entire data word. The signals would be $R_0^0/R_1^0 \ldots R_0^{n-1}/R_1^{n-1}$ for the data being passed and a single A for the entire n bit data word. The acknowledgment should not be generated by the receiver until an event has been seen on each of the data request pairs. As before, transition or four-phase signaling could be used for the transitions, but the number of wires is now further reduced to $2n + 1$ for n bits of data. This protocol is known as *two-plus-wire* communication. If four-phase signaling is used with a two-plus-wire protocol it is also commonly referred to as *dual-rail* signaling [82, 68].

4.2.2 Bundled Data Communication

Although delay-insensitive data communication is guaranteed to work properly in the presence of arbitrary delays in circuit elements and wires, the number of wires needed may be prohibitive in some cases. In addition, the circuits required to implement delay-insensitive signalling are larger than those used in a standard datapath. If some assumptions about the timing of communication signals are acceptable then the number of wires and the size of the circuits may be reduced even more. One method uses a set of standard data wires, one wire per bit, and a request-acknowledge pair to indicate validity of those data wires. The configuration is shown in Figure 4.4 and is known as a *bundled data* protocol [94].

The sequence of events in a bundled data protocol is as follows. First the sender places the desired data value on the data wires. After all the transitions on the data wires have completed the sender initiates a transaction on the request-acknowledge control pair with the receiver. The sender must keep the data stable until it receives an acknowledgment from the receiver.

The timing constraint requires that the data bundle and control wires be constructed such that the transitions on the bundle are observed at the receiver before a signal appears on the control wire. Note that if the data bundle is delivered to multiple receivers, each receiver must see valid data on that bundle before the control signal is produced. This is known as the *bundling constraint* and is similar to, but weaker than, Seitz's equipotential constraint [82]. For example, a set of wires with non-zero delay will not meet the equipotential constraint, but can meet the bundling constraint if the delay of the control wire is guaranteed to be longer than the longest delay in the data bundle.

There are several techniques for meeting the bundling constraint. A technique used frequently in asynchronous I/O bus design, and applicable to VLSI design as well, is to fabricate the data and control wires with as nearly equal delays as possible and then impose an extra deskewing delay on the request wire. The DEC Unibus is an example of an I/O bus designed in this way [6]. A module using this technique must delay the request by a specified amount in order to mask any differences in transmission times of data and request wires. For connections on a single integrated circuit it is often the case that the extra circuitry involved with the request wire is enough to mask the transmission delays of the simpler data path, but the analysis should be done carefully to ensure that this is true. Note that this analysis requires careful examination of the circuit after layout. Delay values for individual wires cannot be determined until the actual circuit layout, with all the physical placement information, is known.

Although analyzing circuit delays is a complex procedure in general, the problem is simpler here because of the locality of the delay constraint. The constraint involves only the modules doing the communication and not the entire system as would be the case with a global constraint such as needed for a system clock. In OCCAM, communication channels are restricted to be point to point connections between exactly two processes. This means that the path whose delay must be analyzed can not span more than two circuit modules. Checking to make sure that the delay requirements have been met is still a difficult task, but the scope is extremely limited.

As with the delay-insensitive data delivery protocols, the control transaction for a bundled data path may use either transition or four-phase signaling. The sending module is responsible for keeping the information on the data wires valid for as long as the receiver desires. The receiver signals that the data is no longer needed by issuing an acknowledge event. The signals used for bundled transition data communication are shown in Figure 4.5. Notice that the receiver must keep the data valid during the time after the request is issued and before the acknowledge is seen.

If four-phase signaling is used with a bundled data path there is a choice of which signal

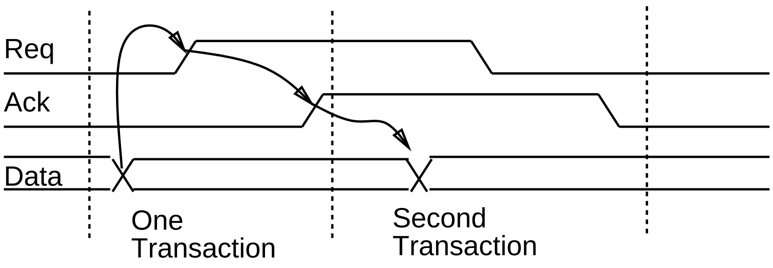

Figure 4.5: Bundled Transition Signaling

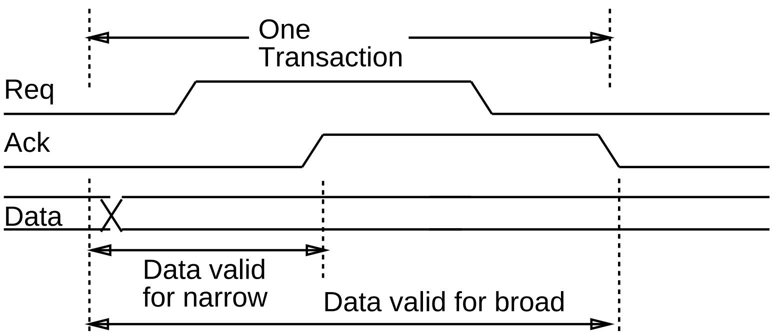

Figure 4.6: Bundled Four-Phase Signaling

from the receiver allows the signals to be removed from the data wires. The most common choice, known as the *narrow convention*, requires that the sender hold the data valid after the request rises and before the acknowledgment rises. The falling transitions of request and acknowledge are assigned no meaning and may be overlapped with the next transition on the data wires. The other choice, known as the *broad convention*, requires that the data wires be maintained after request rises until the acknowledgment falls. Unlike the narrow convention, no control events may be overlapped with data transitions which results in a slower protocol. The advantage of broad convention signaling is that some circuit modules may be simpler to design. The two forms of bundled four-phase data communication are shown in Figure 4.6.

4.3 Circuit Module Specification

In order to specify the behavior of the modules used for the translation it is necessary to specify the precise behavior of the signals at the interface to that module. In general a module's interface will have several input and output signals connected to an environment. This environment may be a single other module, or a collection of modules connected in complex ways. When specifying the behavior of a module it is also necessary to specify the behavior of this environment because a module is not expected to behave correctly in the face of random signals applied to its inputs. Only if the environment obeys the specification as well will the connected system operate correctly. Thus a specification of the behavior at the interface of a module will constrain the module's environment as well as the module itself.

4.3.1 Traces

The model used to describe the behavior of the modules is known as *trace theory* [24, 76, 81, 104]. A *trace* of a circuit represents a history of execution of a module by listing all the transitions of signals at the module's interface. The set of all possible traces of a module, known as its *trace set*, completely specifies the behavior of the module at its interface.

More formally, a trace structure is defined using the communication alphabet of a module Σ. This alphabet consists of a finite number of symbols used to represent the wires over which the module communicates. A trace structure is defined as a triple:

$$\mathcal{T} =< I, O, X >$$

where $I \cup O = \Sigma$, $I \subset \Sigma$ is a finite set of input symbols, $O \subset \Sigma$ is a finite set of output symbols, and $X \subseteq \Sigma^*$ is the set of all possible traces of the module. The set Σ^* is the set of all finite length sequences of symbols in Σ. Although an individual trace represents a single execution history of the module, the trace set X captures all possible execution histories of an interface and thus completely specifies the behavior of the interface. The set X of traces can never be empty because even the trace that contains no symbols contains the empty symbol ε. This indicates that no transitions have occurred on any of the interface wires.

Traces may be extended by appending a new symbol corresponding to a possible transition onto the end of the trace. This indicates that the transition is allowed to occur immediately following the events already recorded in the trace. For example, using the alphabet $\Sigma = \{r, a\}$ to represent requests and acknowledgments we can describe the trace for a transition signaling interface. After initialization the trace of the interface is ε, the empty trace. RA-Rule 3 tells us that the first event that can happen is a request, so r is the first symbol in the trace. After a request, RA-Rule 2 requires that the receiver produce an acknowledgment before the sender issues a new request, so the trace is extended to ra. This process continues as the interface executes producing a sequence of alternating r's and a's.

Trace sets, being sets of simple lists of symbols, are often expressed using familiar regular-expression notation. This notation makes the description of an entire trace set more compact than simply listing all the possible traces. Regular expressions are composed of the symbols in the alphabet Σ of the trace structure, and the special symbols $|$, $*$, (, and). A regular expression is:

- A single symbol from $\Sigma \cup \varepsilon$ where ε is the empty trace containing no symbols.
- $e_1\ e_2$ where e_1 and e_2 are regular expressions. This represents concatenation of the two sequences.
- $e_1 \mid e_2$ which represents an alternative between two regular expressions.
- e^* which represents zero or more repetitions of the expression e.
- (e) which groups all the symbols of the expression e into a single expression.

The trace set of the transition signaling protocol can, for example, be expressed using regular expression notation as $((ra)^* \mid (ra)^*r)$.

Although signaling protocols are often thought of in terms of complete transactions, the trace set must show all partial transactions as well. In order to describe the partial transactions, the trace set must be *prefix-closed*. Intuitively this means that all the partial traces are, themselves, traces of the system. That is, a module can perform the sequence of transitions leading to a complete transaction. Any trace along the way is also a legal trace of the system. Formally, the set of all prefixes of a trace is defined as

$$\text{pref}(X) = s \mid (\exists t : st \in X).$$

A trace structure $\mathcal{T}$ is prefix-closed if pref$(X) = X$ holds. Using this notion of the prefix of a trace, the complete trace set of the transition signaling protocol can be written as pref$((ra)^*)$.

4.3.2 Delay-Insensitivity

Using traces to define the behavior of a module allows properties of the module to be expressed in terms of that trace set. A property that is especially important is delay-insensitivity. Modules whose behavior is delay-insensitive are particularly easy to compose and substitute because communication and processing delays can be ignored. A circuit module meets this delay-insensitive requirement if and only if its interface meets the following two requirements:

DI-Rule 1 *Adjacent events are not allowed on the same wire. In terms of traces, this means that the trace set of the module or protocol must not include any traces that have the sequence aa in them, where a is any symbol in the alphabet Σ.*

DI-Rule 2 *The module must be insensitive to the order of arrival of concurrent inputs. If the module's trace set contains a trace with ab in it, where a and b are both inputs, then it must also contain the same trace with the sequence ba substituted for ab.*

DI-Rule 1 prohibits changes on a single wire that might not be seen by a slow receiver. This is related to RA-Rule 1 for transition signaling where another transition on the output request wire could not be made until an intervening acknowledge has been received. DI-Rule 2 is required because the order of arrival of inputs to a module may be different from the order in which the environment produced those signals. This has been characterized as a foam rubber wrapper surrounding the module [80].

To understand delay-insensitivity using a foam rubber wrapper, imagine that the circuit module is encased in a thick coating of foam rubber. The interface as seen by the environment is a set of signals applied to the outer surface of this wrapper. These signals then penetrate the wrapper and make their way to the circuit module inside. Depending on the thickness of the wrapper, the signals may take longer or shorter times to reach the actual circuit module. Because the environment cannot tell how thick the wrapper is at any point on the outer surface, it cannot tell how long any individual signal will take to reach the circuit once it has disappeared into the wrapper. Because signals may take different amounts of time to pass through the wrapper, concurrent inputs may arrive in a different order at the circuit module than the order in which they were applied at the surface of the wrapper. For a circuit module to operate correctly when encased in such a wrapper, any order of arrival for concurrent inputs must be acceptable.

4.3.3 Interface Nets

While regular expressions are a more convenient method for describing trace sets than simply enumerating the possible sets, they are still not easy to understand or manipulate in many cases. An easier way to describe trace sets uses another popular tool for describing the behavior of systems called Petri nets [73]. *Interface nets*, or *I-nets* [64] are a restricted class of Petri Nets used to model the interface behavior of a module. An interpretation rule allows the I-net to be executed to produce the trace set of the interface. I-nets can be thought of as event engines, machines that, when run, generate the events allowed at the interface of the module being described. The I-net can also be analyzed to produce a state graph or other representations of the module, and the graphical nature of the I-net may reveal the symmetry and concurrency of a module's interface more clearly than other approaches.

An I-net consists of a set of places, drawn as circles, transitions, drawn as bars, and arcs, drawn as arrows that connect places and transitions. Each transition in an I-net is labeled with an interface signal of the module being described, and the firing of a transition in the I-net corresponds to a signal transition on the wire represented by the label. *Input places* of a transition are all places that have arcs leading to that transition, and *output places* of a transition are all places that have arcs leading directly from the transition. Each place in the I-net may contain a finite number of tokens shown as dots inside the places. A transition is said to be *enabled* if each of its input places contains at least one token. A *marking* of an I-net is an assignment of tokens to specific places. An I-net starts with some initial marking

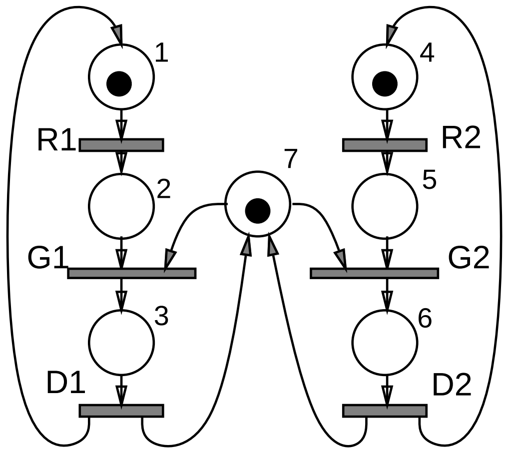

Figure 4.7: An Example I-net

which is then modified by the execution of the I-net.

An example I-net is shown in Figure 4.7 along with its initial marking. In this I-net the initial marking is the configuration with tokens in places 1, 4, and 7. The transitions are labeled with the names of the communication actions they represent. In the initial marking, only transitions *R1* and *R2* are enabled.

An I-net is executed by *firing* enabled transitions. At each step in the execution exactly one of the enabled transitions fires. If more than one transition is enabled the choice of which to fire is arbitrary. When a transition fires it removes one token from each of its input places and adds one token to each of its output places. In the example of Figure 4.7, if transaction *R1* fires, a token is removed from place 1 and a token is added to place 2 and the trace described by executing the I-net is extended by appending an *R1* action at the end of the trace. In the new marking of the I-net, transactions *R2* and *G1* are now enabled.

Firing corresponds to an occurrence of the event that labels the transition. In this case, firing a transition corresponds to a transition on the wire represented by the I-net transition. If no transitions are enabled, the I-net *halts*. Each time a particular transition is chosen from the set of enabled transitions a choice is made between one of several possible sequences of events. If all these sequences are enumerated the result is the trace set of the system modeled by the I-net. As with any trace description of a module, the role of the module and environment are symmetric. The I-net models the constraints imposed on the environment as well as the constraints imposed on the module itself.

4.3.4 Interface State Graphs

In addition to generating the trace set of a module, executing an I-net can also result in a simple state graph structure. This state graph can be used to synthesize a physical circuit to implement the module. The state graph is obtained be exhaustively simulating the I-net and constructing a graph where each node of the graph corresponds to a distinct reachable marking of the I-net and each arc leaving a node corresponds to an interface signal of the module being described. These arcs represent I-net transitions that are enabled to fire by the marking which corresponds to that node. Each arc terminates in a node that represents the new marking produced by firing the transition that labels the arc. The resulting graph is known as an *interface state graph* or *ISG* [64].

Circuits may be generated from ISG descriptions by encoding the nodes of the ISG with the current values of the interface variables. Values are assigned for the interface variables in the initial state, usually zeros, and transitions in the interface variables are echoed in the new state vector for subsequent nodes in the ISG. The resulting encoded ISG (EISG) may be implemented by building a Karnaugh map for each of the output variables. This procedure, while usually simple, may be complicated by the need to add states to the EISG to distinguish between identical states. The resulting circuits may not be delay-insensitive even though the interface has this property because there may be timing relationships internal to the circuit related to the implementation. Careful circuit design can eliminate the effect of possible races and hazards in the resulting circuits and allow them to be used as truly delay-insensitive circuit modules.

4.4 Control Path Modules

Control signals in the target circuits are manipulated using a set of control modules. Unless otherwise stated, these modules all use transition signaling and are delay-insensitive at their interfaces. Each responds to a global *clear* signal that clears the internal state, if any, and sets its outputs to a known state. To keep things consistent, all modules will set their outputs to a low state upon receipt of a *clear* signal. This *clear* signal is not shown in order to make the circuit diagrams less cluttered and easier to understand.

When these control modules are combined in a circuit, there is a special control signal used to initiate the action of the circuit. This signal, called *start*, is a single transition from low to high that is produced by the environment after the circuit has been cleared. Its purpose is to start activity in the circuit. As such, it is generally wired to the *Req* signal of the top level processes in the system.

Control modules used in the target circuits are described using an I-net, drawn with an initial marking, to describe the trace set of the interface. The symbol used to denote each circuit module is also shown. Each of these modules has been implemented using the MOSIS scalable CMOS technology. Circuit schematics for this implementation of the modules can be found in Appendix A.

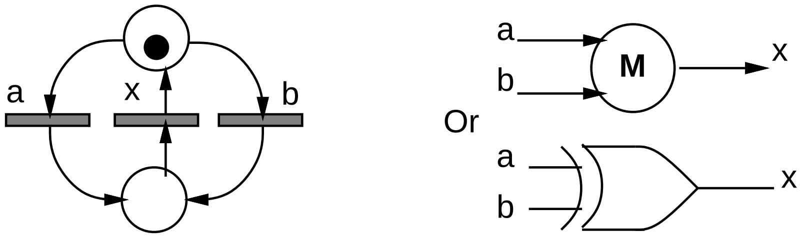

Figure 4.8: A Merge Module

4.4.1 Merge

A Merge module is the "OR" function for transition signals: A transition on either input causes a transition on the output. The I-net and symbol are shown in Figure 4.8. The module has two inputs, a and b and a single output x. It will cause a transition on its output when a transition arrives on either of its inputs. After an event on a or b, a subsequent input event must not occur until after an output event x has been generated. Note that this is a constraint on the environment, not the Merge module itself. Nevertheless this constraint is included in the module's description.

After initialization the first event to occur must be an input event. When cleared, a Merge module will force its output to a low state. If other modules in the system also follow this convention, then a Merge can be implemented using a standard XOR gate augmented with some provision for forcing its output low on a *clear* signal.

A Merge module may be extended to include multiple input signals where an output transition is generated in response to a transition on any of the input signals. In this case the implementation is extended to a parity gate with multiple inputs.

4.4.2 Join

The Join is the "AND" function for transition signals: it will cause a transition on its output only after a transition has arrived on each input. An I-net and symbol are shown in Figure 4.9. When an event has occurred on both the a and b inputs an event is generated on the x output. A subsequent input event must not occur until the output event has been generated and after initialization the first event to occur must be an input event. Note that the behavior of the two input events obeys DI-Rule 2 which says that the order of arrival of concurrent inputs is not important.

When cleared, the Join module clears its internal state and forces its output to a low state. One way to implement a Join module is to use a Muller C-Element [82] enhanced with a clear signal. The behavior of a C-element is that whenever the state of its input signals matches, it copies that state to the output; at all other times its output does not change.

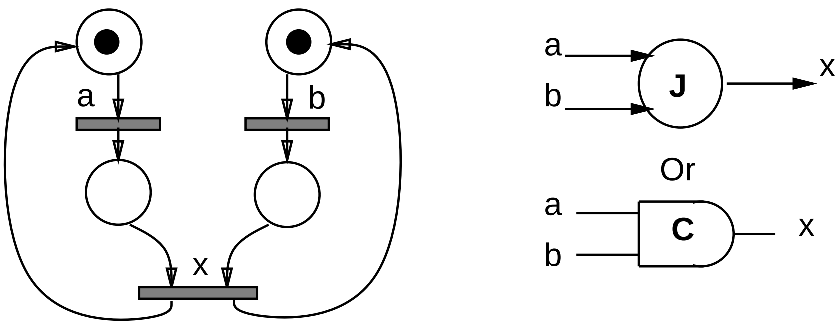

Figure 4.9: A Join Module

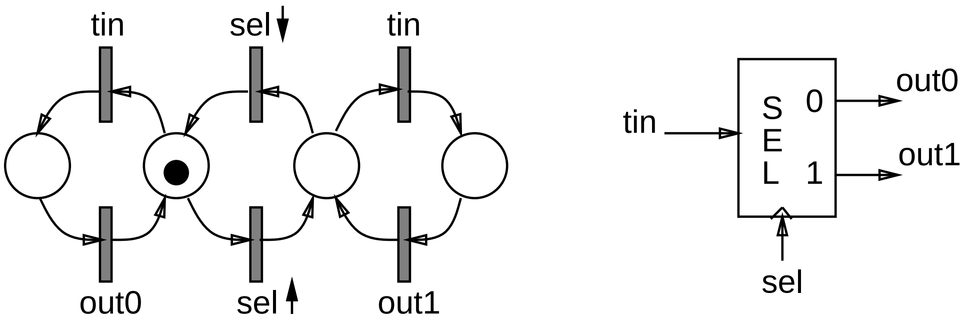

Figure 4.10: A Select Module

A Join module may be extended to multiple inputs by extending the definition so that an output transition will not occur until a single event is received on each of the inputs. The extended Join may be implemented using a generalized C-element with multiple inputs.

4.4.3 Select

A two-way transition Select module will steer an input transition to either of two outputs depending on the value of a boolean data signal. This boolean data signal, called *sel*, is a bundled data signal that uses the transition input *tin* as its control signal. That is, the *sel* signal must be valid from before a transition occurs on *tin* until after an output transition is generated at one of the outputs. Note that because the Select module reads, but does not write, the value of the bundled boolean, the bundling constraint is a constraint on the environment. The environment must avoid making a transition on *tin* until after the boolean *sel* bit is valid. There is no way that the select module itself can enforce this constraint. The I-net and symbol are shown in Figure 4.10. A transition Select module will clear its internal state and force its outputs to low on a *clear* signal.

Note that although the transitions of an I-net correspond to transition events on signal

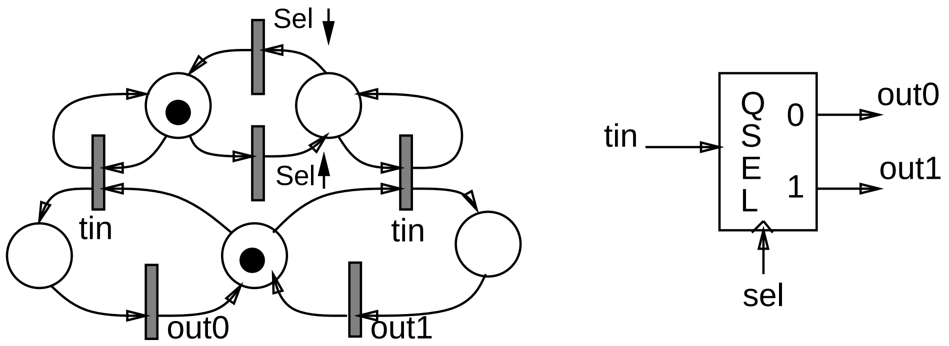

Figure 4.11: A Q-Select Module

wires, an I-net can still capture the behavior of a bundled data path. The transitions on the *sel* wire change the value on that wire from low to high or high to low. This is modeled in the I-net as separate transitions. Transitions that denote a bundled signal changing value are marked with an arrow to indicate the direction of change. The bundling constraint is described in the I-net by arranging places such that after a transition on *tin* further transitions on the *sel* wire are not allowed until a transition on one of the outputs has been generated.

4.4.4 Q-Select

This module is like a Select module except that it is delay-insensitive: there is no bundling constraint on the *sel* signal. This means that the select signal might be changing when the input request occurs which, in turn, means that a special circuit must be used to sample the input that might be changing. The modified I-net to allow this unbundled behavior is shown in Figure 4.11. Note that, unlike the I-net for the Select module, transitions on the *sel* wire may occur at any time. A transition on *tin* will immediately enable another transition on *sel* even before an event on one of the outputs occurs.

The special circuit used to sample the *sel* signal in a Q-Select is known as a Q-flop [79]. A Q-flop is a circuit that allows the value of a signal to be sampled on request, and generates an acknowledge when a reliable data value is available at the output. Because the sampling circuit may enter a metastable state, the acknowledge must be delayed until the circuit has exited metastability and a reliable result is reported.

The Q-Select module will sample the value of the *sel* input using a Q-Flop when an input transition *tin* is received. It then uses that sampled value to steer the transition to one of two outputs.

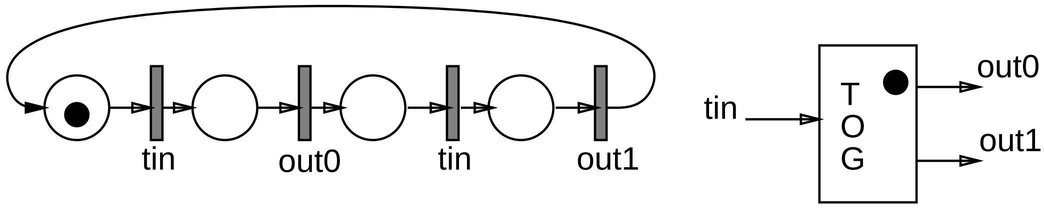

Figure 4.12: A Toggle Module

4.4.5 Toggle

A Toggle module routes an input transition *tin* alternately to its two outputs. The outputs of a Toggle module are forced to low on a *clear* signal and after initialization the first input transition will be routed to the *Out0* output and subsequent input transitions will be routed to alternate outputs. When a Toggle module is drawn as a circuit element the output that receives the first transition after a *clear* is marked with a dot. This symbol, and the I-net for a Toggle are shown in Figure 4.12.

4.4.6 Call

A transition Call module implements the hardware equivalent of a subroutine call. In response to a request on either of the two request lines *R1* or *R2*, the call module starts a "subroutine" process with the *Rs* signal. The process making a request to the Call module is known as a *client* of that call module. When the subprocess completes and acknowledges with *As*, the call module acknowledges the appropriate client on *A1* or *A2*. It is the responsibility of the call module to remember which of *R1* or *R2* made the request so that the acknowledge gets routed to the correct client.

A full request-acknowledge transaction must be completed before either side may request again and the input request signals must be mutually exclusive. A Call module can generalize to any number of input request/acknowledge pairs. A Call with only a single input pair is simply a pair of wires connecting the input request to the subroutine request and subroutine acknowledge to input acknowledge. The I-net and symbol for a two-input call are shown in Figure 4.13. The arrow in the Call module's symbol indicates the direction of the call. That is, requests are routed to the subroutine along the direction of the arrow. A Call module will force its outputs to a low state on a *clear* signal.

4.4.7 Arbiter

A two-way transition Arbiter is used to guarantee mutually exclusive action of two independent requesters. If only one of the users requests action, access will be granted promptly.

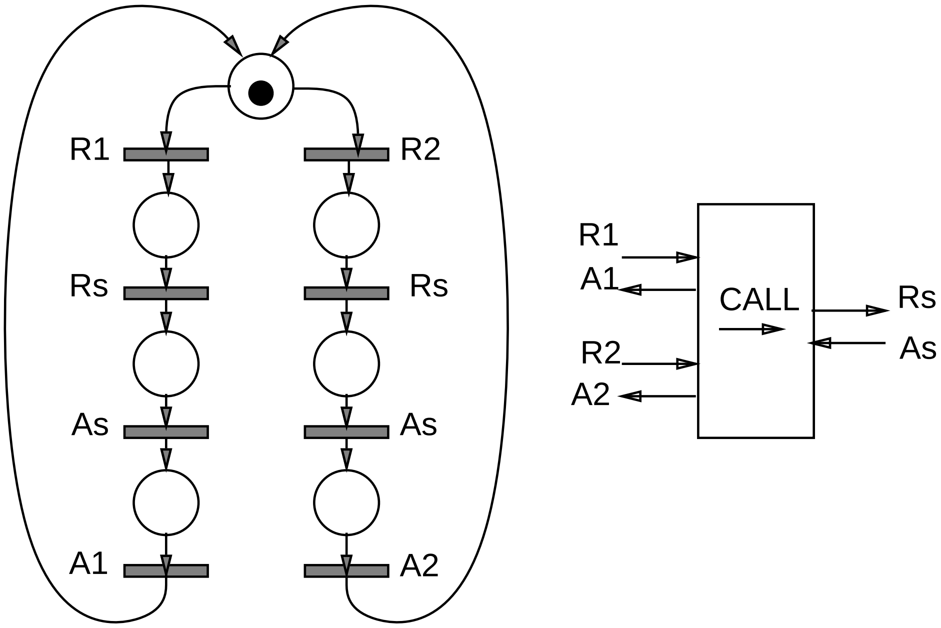

Figure 4.13: A Call Module

If both users request action, the Arbiter will grant access to only one of the two requesters. After a user receives a grant it performs the desired action, and then tells the Arbiter that it is finished which allows the Arbiter to grant access to the next requester. The signals, shown in Figure 4.14, are *R, G,* and *D* for each of the requesters for request, grant, and done respectively. The place marked *ME* is the part of the I-net that specifies that only a single grant is allowed at any time. Because a single place with a single token is an input place for both grant transitions, only one may occur at any time, and another grant may not occur until a done signal has deposited a token back into the *ME* place.

In fact, the difficult part in the design of a transition Arbiter is designing a circuit to select one of two concurrent input requests. Ordinarily the first request to arrive is selected, but if two requests arrive in a short enough time interval the arbiter must decide which one to service. This decision is made using a circuit called a *mutual exclusion element* to make the difficult choice. This circuit, along with the surrounding circuitry to make the Arbiter module, is shown in Appendix A.

An arbiter will set its outputs low and clear its internal state on a *clear* signal. The first event after initialization must be an event on one of the two request lines.

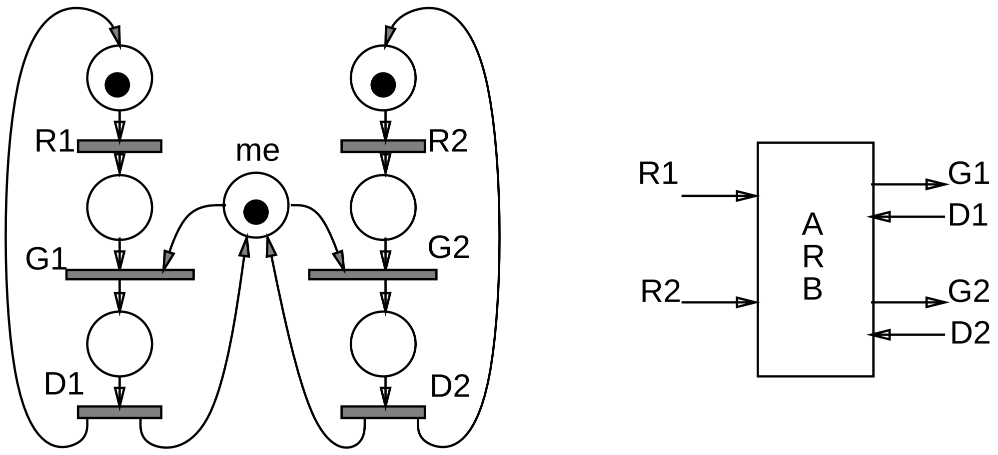

Figure 4.14: An Arbiter Module

4.5 Data Path Modules

In addition to control modules, modules are required that deal with modifying data or passing data around in a circuit. Unless otherwise mentioned, the target data path modules all use bundled data paths with transition signaling used for the control lines. These modules generally set their control state upon receipt of a *clear* signal, but do not clear the state of the data path unless specifically mentioned. As with the control modules, the clear signal is not drawn for clarity. Circuits for these modules implemented in MOSIS scalable CMOS can be found in Appendix A.

4.5.1 Bundled Data Path Elements

In general, any combinational function can be used as a bundled data path element. The only restriction is that the control signal be delayed enough to account for delay through the combinational function and possible transmission delay to the receiver. The model is shown in Figure 4.15. The request transition is simply delayed to become the acknowledge. This bundling delay is required to ensure that the module obeys the bundling constraint with respect to the receiver of the data.

The main advantage of this method of building logic functions is that standard techniques, or existing cell sets, can be used to transform the data and still be used in the framework of an asynchronous system. The disadvantages include that a careful examination of the worst-case delay through the logic block and the data delivery to the receiver is required to guarantee that the bundling constraint is met under all conditions. Guaranteeing that the bundling constraint is met under a variety of different power supply characteristics, temperature ranges, and process variations will often require the bundling delay to be large

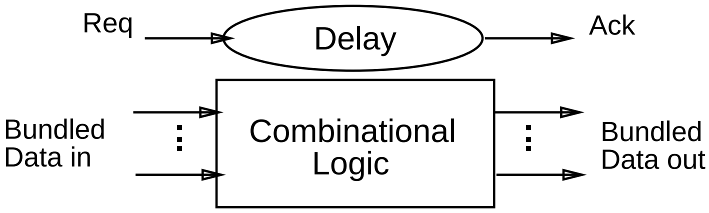

Figure 4.15: A General Bundled Data Module

compared to the average case performance of the logic. This not only slows down the module, but also the entire system that uses this module.

4.5.2 Protocol Converters

When bundled data is used in an asynchronous system it is sometimes the case that different circuit modules will use different signaling conventions for the control parts of the bundled data interfaces. For example, most of the control might use transition signaling, but some data path element might use four-phase bundled signaling with the narrow convention. In cases like this it is helpful to have a set of protocol converters to connect modules whose interfaces use different delay-insensitive signaling conventions. This allows different parts of a system to be built using the convention that best suits each application and still allow the parts to be connected together into a complete system.

The I-nets and symbols for six possible protocol converters are shown in Figures 4.16 to 4.21. Note that different converters are required for each of the narrow and broad conventions used with a four-phase bundled protocol. Also, because a four-phase broad convention actually meets the specification for a narrow convention, no converter is really necessary to convert from broad to narrow. If a converter is used, however, some concurrency is gained by overlapping the last two transitions of the broad and narrow sides of the converter.

4.5.3 Enable

An Enable module is used to construct multiplexers for bundled data by gating data onto a shared bus. 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 disable request (*Rdis*) is received, the data outputs are placed in a high impedance state and a disable acknowledge (*Adis*) is generated. Correct use of this module requires that enable and disable requests alternate.

The I-net for the control signals and the symbol for an Enable are shown in Figure 4.22. For clarity, the bundled data path is not shown in this I-net, but because the Enable module

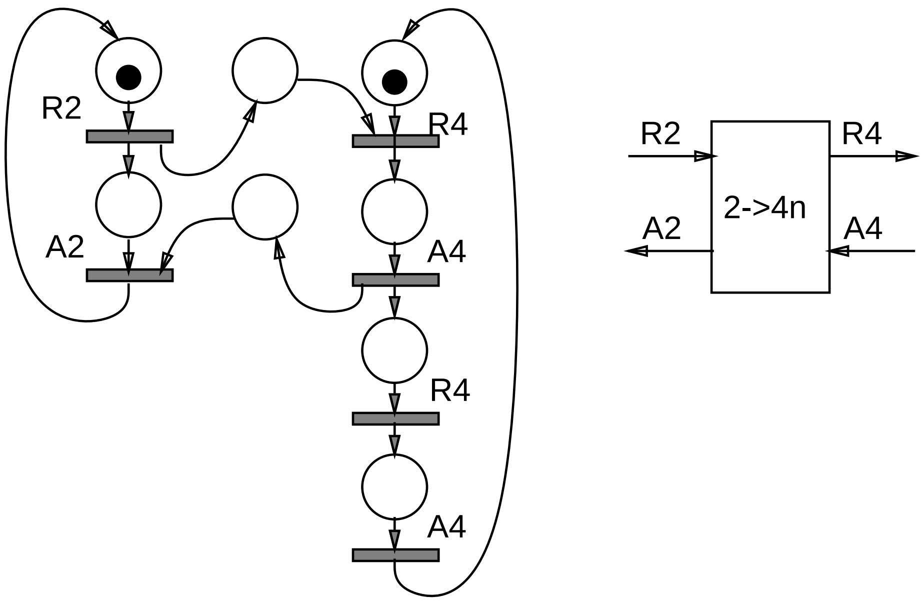

Figure 4.16: A Transition to Four-Phase Narrow Converter

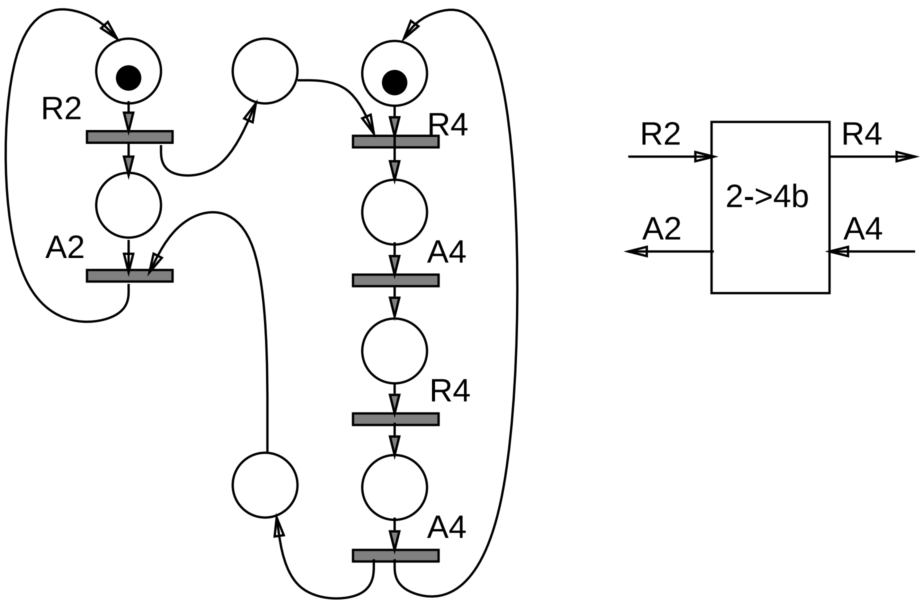

Figure 4.17: A Transition to Four-Phase Broad Converter

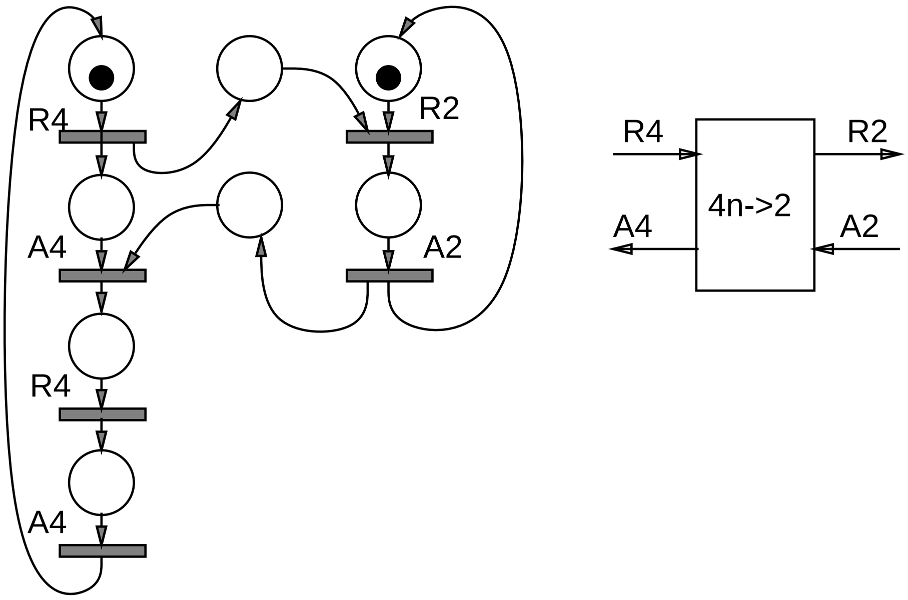

Figure 4.18: A Four-Phase Narrow to Transition Converter

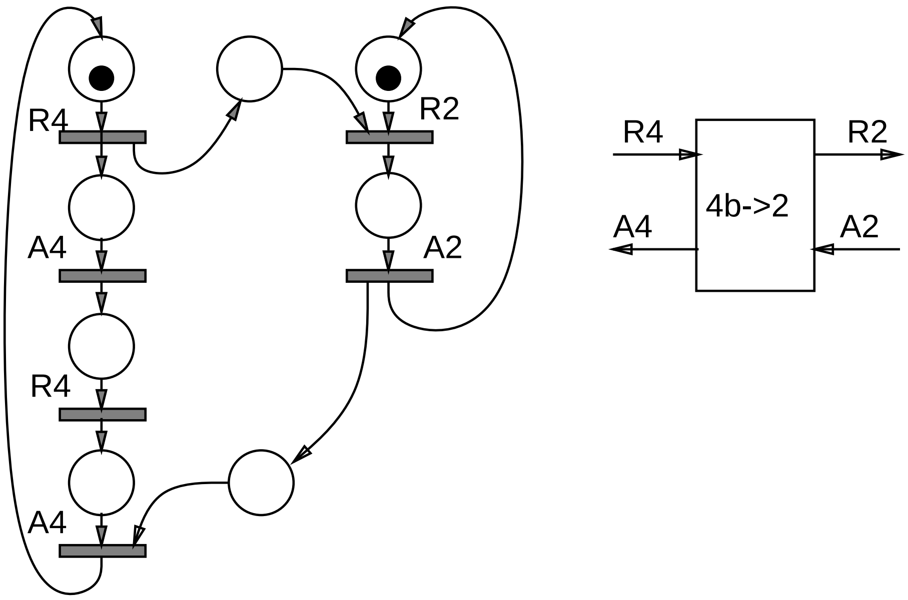

Figure 4.19: A Four-Phase Broad to Transition Converter

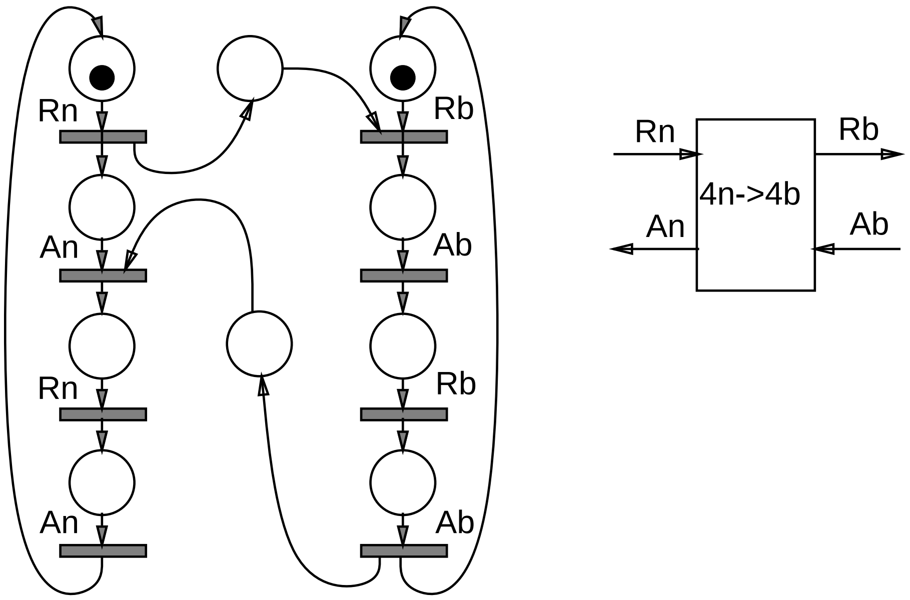

Figure 4.20: A Four-Phase Narrow to Four-Phase Broad Converter

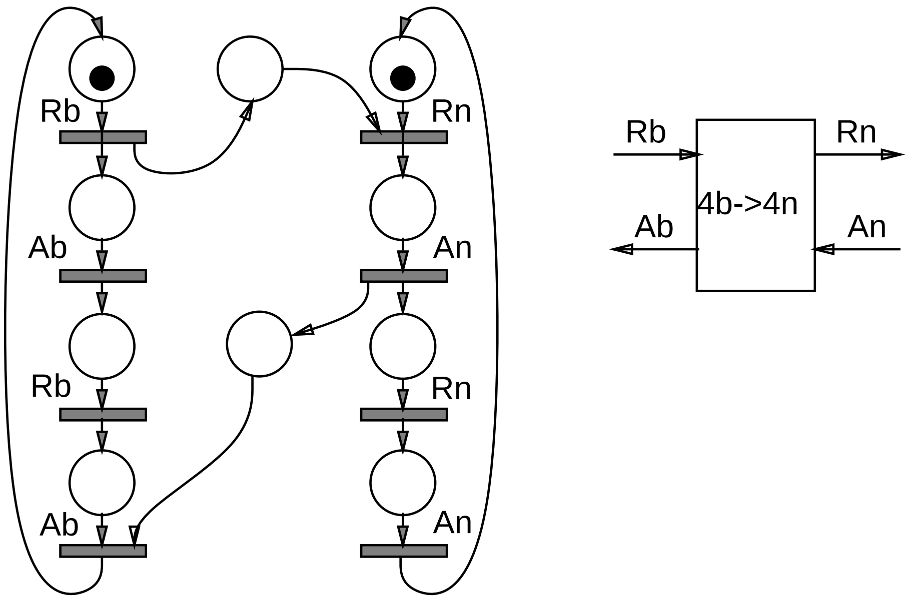

Figure 4.21: A Four-Phase Broad to Four-Phase Narrow Converter

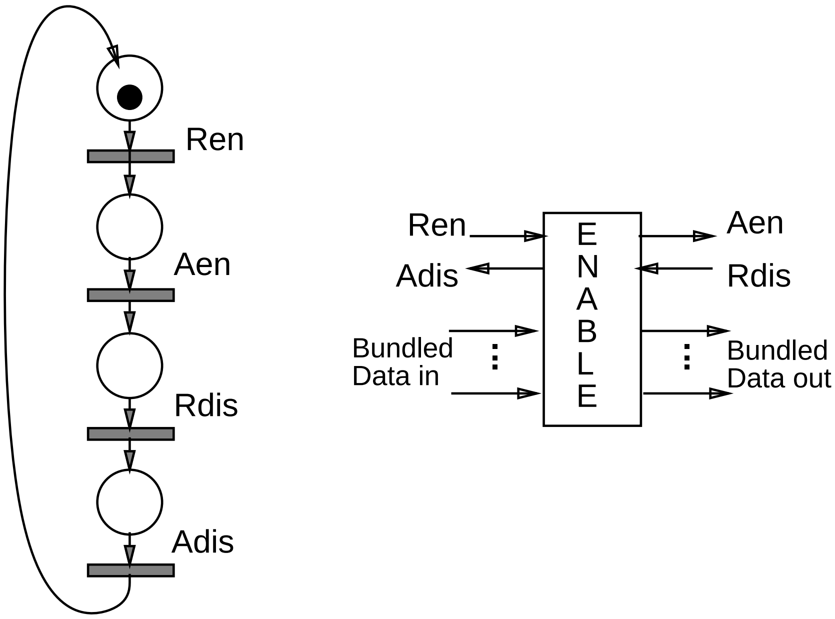

Figure 4.22: An Enable Module

uses bundled data there are bundling constraints that must be met. The constraint on the environment is that the data at the input of the Enable must be valid before an *Ren* signal is received, and the data must stay valid until a complete transaction has occurred and an event on *Adis* is seen. This is because there is no storage in the enable module. The data must remain valid at the sender during the entire time that the enable module is driving its outputs. The bundling constraint enforced by the Enable module is that the enabled data value at the output of the Enable must be valid on the bus before the *Aen* signal is generated.

4.5.4 Register and Latch

An asynchronous Register module is used to store bundled data on the direction of the control signals belonging to the bundled data path. The control signals could, of course, use any of the possible signaling forms but in this case the control uses transition signaling. When a request transition arrives at the Register, the input data are latched. The bundling constraint on the environment requires that the data be stable at the input to the Register before the request is received. An acknowledge transition is generated when the data are latched and, according to the bundling constraint on the Register, the output data are valid at all recipients of the data. The I-net for the control and the symbol for a Register module are shown in Figure 4.23.

A Latch serves a similar purpose to the Register, but uses a slightly different set of

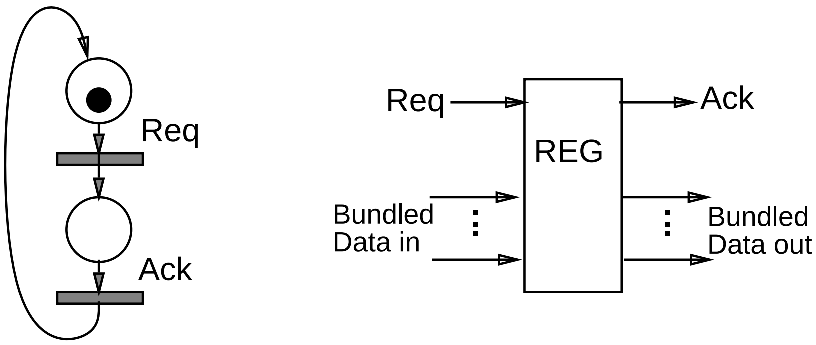

Figure 4.23: A Bundled Transition Register

control signals. Rather than a simple request-acknowledge interface, a latch has explicit control over the transparent and opaque states of the latch [94]. The *Cin* signal tells the Latch to capture the data and become opaque and the *Cout* signal signals to the environment that the data have been latched. The usual bundling constraints hold here that the data must be stable before a *Cin* and the outputs must be valid before a *Cout* is issued. The *Pin* and *Pout* signals put the latch into a transparent state and pass the data from input to output. Correct operation of the Latch requires that the *C* and *P* transactions alternate as shown in Figure 4.24.

There are actually two types of Latch modules that can be used that differ only in their initial conditions. The latch shown in Figure 4.24 is normally transparent. This means that after initialization the Latch is transparent and the first transition must be on the *Cin* wire to capture some data. Another form of Latch is normally opaque. The only difference is that the initial marking of the I-net would not have a token in the top place as in Figure 4.24 and instead, would have a token in the place above the *Pin* transition. This Latch is opaque after initialization and the first event must be a *Pin* to turn the Latch transparent and pass data through.

4.5.5 Boolean Variable

A Boolean register is a single-bit register, outfitted with transition signaling interfaces for changing the register's contents. A transition on the *Rset0* wire will set the value of the register to zero and acknowledge on the *Aset0* line. The interface for setting the value to one is similar. The use of these interfaces must be mutually exclusive. A third delay-insensitive interface allows the value of the register to be tested. A transition on the *test* input will produce a transition on either the *A0* or *A1* output depending on the current value of the register. A delay-insensitive implementation of a boolean register similar to that shown in Figure 4.25 may be found in [89].

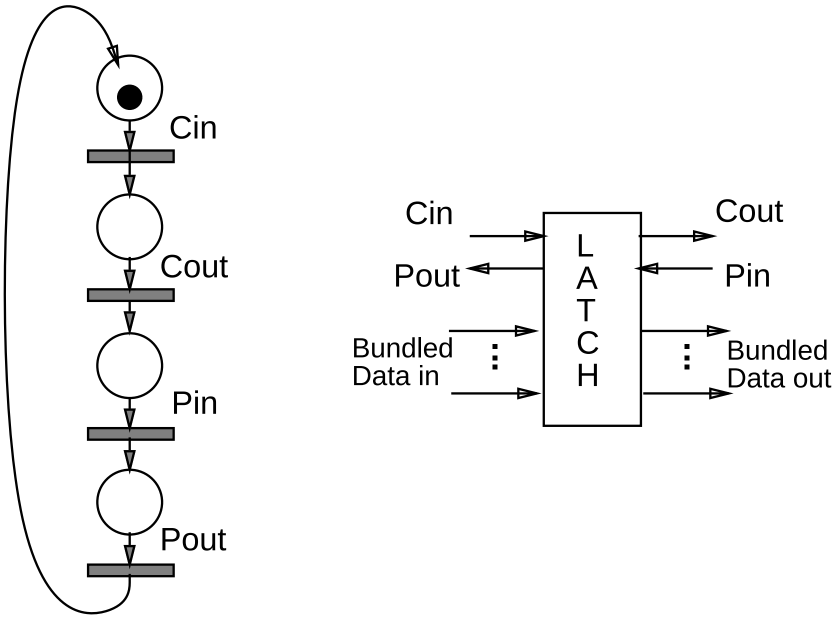

Figure 4.24: A Bundled Transition Latch [94]

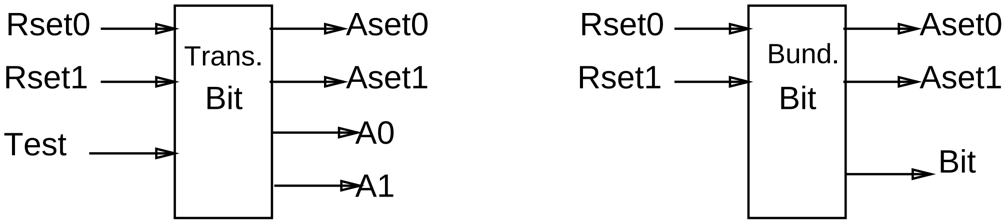

Figure 4.25: Boolean Variables

Another form of single bit register also shown in Figure 4.25 uses transition control paths to set and reset the value of the bit, and a bundled data bit as output. In this form, the request-acknowledge interfaces for changing the value of the variable are the same, but the acknowledgments must enforce the bundling condition that the data value is stable wherever it is seen before the acknowledge is issued. This type of bit is useful for generating boolean values sensed by Select modules.

4.5.6 Carry Completion Addition

One data path function that seems essential in any real system is addition. A simple way to implement addition in a self-timed system is to treat addition as a combinatorial function and build a general bundled module around that function. Any particular implementation of an addition operation might be used from ripple-carry addition to full carry-lookahead addition. The challenge is to identify the worst-case path through the resulting circuit and use the timing of that path to develop a delay model that meets the bundling constraint. The problem is that for a function like addition, the difference between the best-case delay, average delay, and worst-case delay may be dramatic, resulting in huge speed penalties in most cases because the worst-case delay must be used every time. Using simple ripple-carry addition, for example, the bundling delay must model the case where the carry propagates from the low order bit through the entire adder to the high order bit. Because this case is rare the delay model will be pessimistic most of the time.

One solution is to use a form of adder circuit that can identify when each of the bits in the addition is completed. This form of addition is called *carry-completion sensing* or *CCS* addition [41]. The truth table for this form of addition, shown in Figure 4.26, reveals the trick. The carry is propagated using two separate signals: *Cout* to indicate a carry, and *Dout* to indicate the absence of a carry. 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.

A self-timed addition module can be constructed using a CCS adder module. The modules are chained together with the *Cout* and *Dout* of one stage connected to the *Cin* and *Din* of the next. The *En* signals from each bit of the adder are connected together and become the request line, and the acknowledges from each bit of the adder are connected into a tree of Join, or C-element, modules to signal the condition that all the adder bits have cleared, and that all the adder bits have produced their carry. The CCS adder module constructed in this fashion is shown in Figure 4.27 and uses four-phase narrow signaling. The rising *Req* transition initiates addition by allowing each bit to compute its sum and carry. The rising *Ack* indicates that each bit has finished computing its carry to the next stage, and thus that the addition is complete. A falling *Req* transition is required to initialize the CCS adder for the next addition by pulling all the carry signals low, and the falling *Ack* indicates that each bit is reset and the CCS adder is ready to compute a new result.

4.6 Examples

The following simple circuits are used as examples to show how these basic modules might be connected into larger modules or systems.

En	Cin	Din	AB	Sum	Cout	Dout
1	0	0	00	x	0	1
1	0	0	01	x	0	0
1	0	0	10	x	0	0
1	0	0	11	x	1	0
1	0	1	00	0	0	1
1	0	1	01	1	0	1
1	0	1	10	1	0	1
1	0	1	11	0	1	0
1	1	0	00	1	0	1
1	1	0	01	0	1	0
1	1	0	10	0	1	0
1	1	0	11	1	1	0
1	1	1	xx	x	x	x
0	x	x	xx	x	0	0

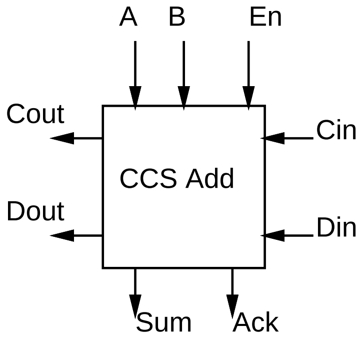

Figure 4.26: Truth Table and Symbol for a CCS Addition Bit

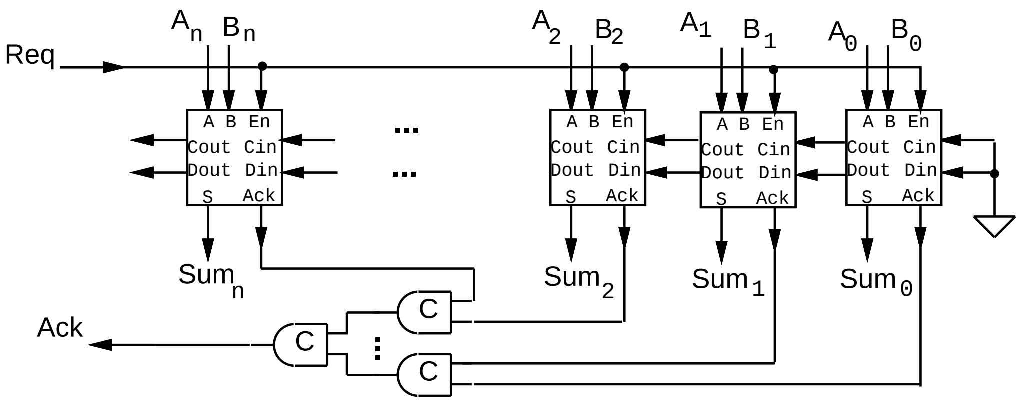

Figure 4.27: CCS Adder Module

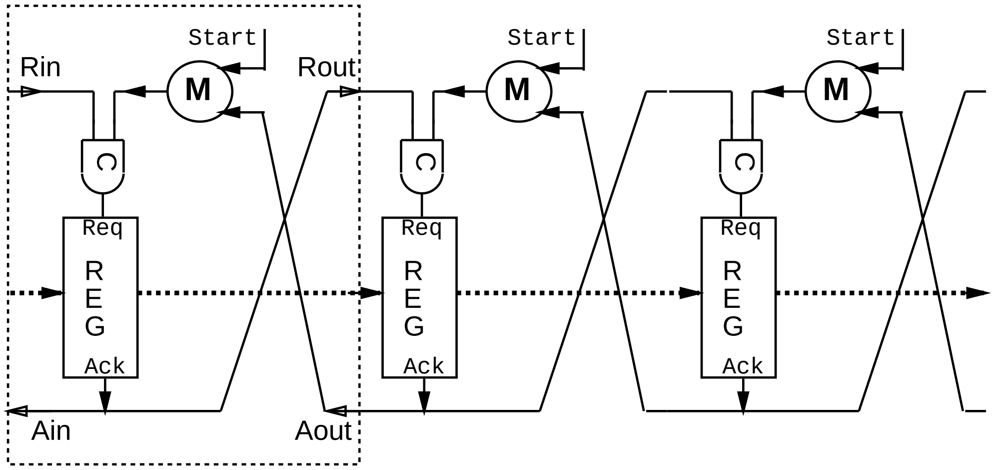

Figure 4.28: A Flow-Through FIFO Circuit

4.6.1 Branch and Join FIFO control

A simple flow-through first-in first-out (FIFO) buffer can be constructed, in the same style as in [94, 91], by using registers to hold data in the FIFO, and Join modules, implemented as C-elements, for control. The control for a single stage of FIFO requires a request-acknowledge interface at both the input and output. The input request signals that there are new data available at the bundled input data path, and the output request indicates to the next stage that there are new data presented at the bundled output data path. The operation of a single stage of FIFO is to accept new input data after initialization, and thereafter accept subsequent data only after the next stage has taken the current data presented at the output.

The circuit is shown in Figure 4.28. The Merge module will produce a transition at the C-element after a *start* signal. Thereafter, the C-element makes sure that new data are latched only after the next stage is finished, signalled by a transition on *Aout*, and there are new data available from the previous stage, signalled by *Rin*.

As an extension of this simple FIFO, a FIFO-branch module can be used to deliver alternate input values to two different FIFO buffers. A corresponding FIFO-merge module will take values alternately from two different FIFO buffers and merge them into a single FIFO again. The overall block diagram of this system is shown in 4.29. The symbols use the same dot convention as the Toggle module to identify the first data path that is active after initialization. Using the circuit modules from this section, modules for the FIFO-branch and FIFO-merge can be constructed as shown in Figures 4.30 and 4.31.

The circuit in Figure 4.30 for the FIFO-branch uses a register to hold the value of the data from the input port. Once these data are latched, a Toggle is used to send these data to alternate output ports by routing the transition to the appropriate place. When whichever output port the data are sent to responds with an acknowledge, the C-element is enabled to

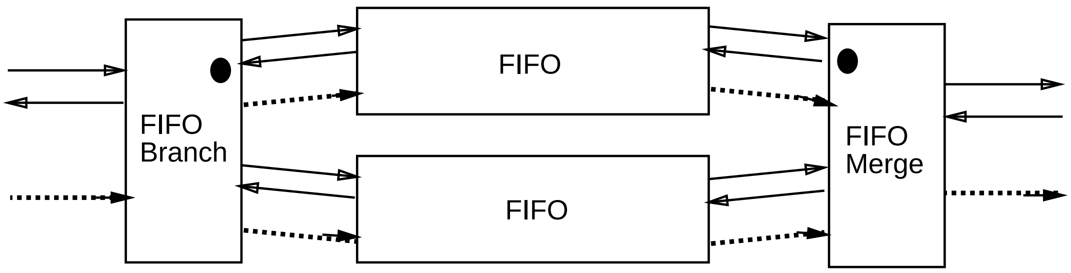

Figure 4.29: Branching and Merging FIFO Buffers

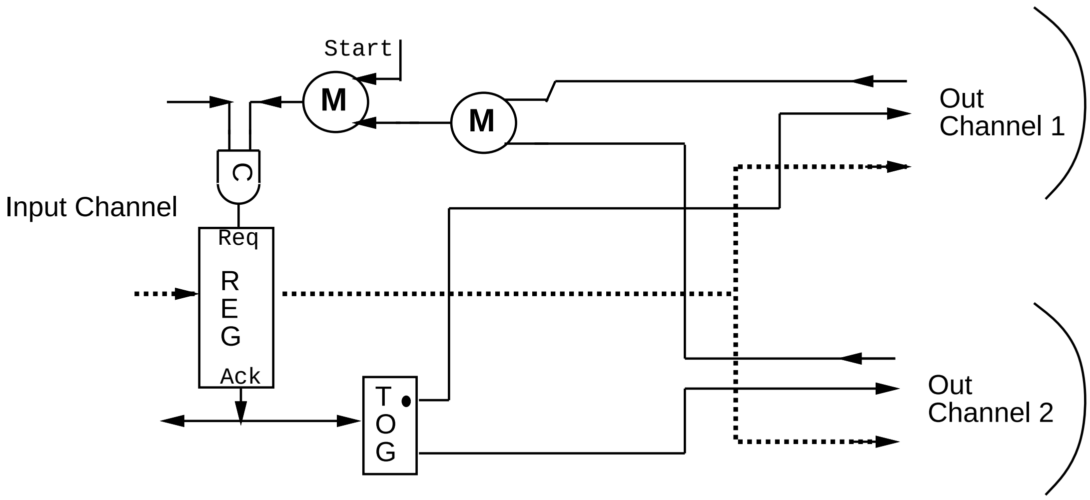

Figure 4.30: A FIFO-Branch Module

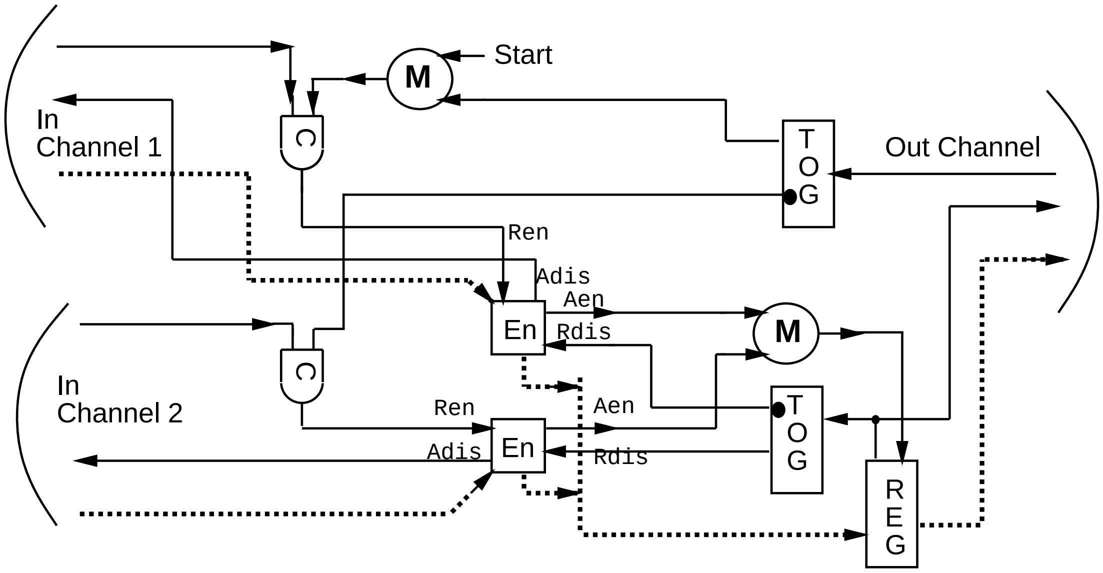

Figure 4.31: A FIFO-Merge Module

allow another value to be latched when it becomes available.

The implementation for a FIFO-merge shown in Figure 4.31 shows how Enable modules may be used to construct multiplexers. Each of the two bundled input paths that can be written to the register go through an Enable module. Separate control lines for each data path control what data are presented to the input of the Register. After initialization, data are accepted from the top channel when they become available. The control signal passes through the Enable module resulting in the data from the top data path being presented at the input to the register. This signal is routed through a Merge module to the control interface of the Register. After the value has been latched into the register, the top bundled data path may be acknowledged through the Toggle module, and a request sent to the output channel. When the output channel has been acknowledged, the control enables the lower C-element to allow data from the lower path to be latched. Note that because the signal from the Register to the Enable modules is routed through a Toggle module, the two data paths will be acknowledged alternately. Also, the Toggle module at the output channel will enable the input channels in the same alternating sequence.

4.6.2 A Self-Timed ALU

Using the CCS adder module described in section 4.5.6 a full ALU can be constructed. Although the CCS adder is a four-phase module, the external interface to the ALU should use transition signaling to fit better with the rest of the control modules.

First, a logic unit is designed as a general bundled module. That is, a circuit is built which performs all sixteen logical combinations of two bits, and a delay model is built to model the worst-case delay through the combinational logic. The logic block is a simple two-level multiplexer circuit that is easy to characterize in terms of delay so this is not a serious problem or a serious speed penalty. The logic unit uses a four-bit wide bundle of function information to determine which function to compute. The environment must make sure the function bits are valid at the logic unit before sending a request transition to the delay model.

The CCS adder is a four-phase module that clears its sum output along with the carry chain during the return to zero phase of the handshake. Because it would be nice to allow the adder to clear and prepare for the next addition while some other module is using the value at the output of the ALU, a register is included in the ALU module to hold the output value while the adder clears for the next addition. The ALU uses a bundled selection bit, *addsel*, to determine if the output of the ALU is taken from the logic unit or the adder and thus computes the function

$$\text{if } \textit{addsel} \text{ then} F(A,B) + B \quad \text{else} \quad F(A,B)$$

.

A circuit that implements an ALU is shown in Figure 4.32. The ALU must be initialized using a *start* signal just like the FIFO circuits. The request transition first travels through the delay model of the logic unit, and then arrives at the C-element which, having already received a transition on the other input, passes the transition through the merge and to the CCS adder. This signals the adder to begin an addition. When the addition is acknowledged, the toggle delivers the transition to the register latching the result. When the result is latched, the request to the output bundle indicates that data are available, and also goes back to the CCS adder to begin the clearing phase. The C-element ensures that only when the adder has cleared will the next ALU cycle be allowed to start.

An extension of this circuit uses the bundled *addsel* signal to bypass the adder when the addition is not required. The *addsel* signal is true when the adder is being used, and false otherwise so a Select module can be used, as shown in Figure 4.33, to deliver a request to the CCS adder only when the ALU function requires an addition. The Call module in front of the Register shows that the register needs to be latched in either case. Note that both outputs of the Call module go to the same merge module. This would seem to indicate that the Call module is not needed. In this case, however, the CCS adder must be cleared with a second request signal only in the case that the adder is actually being used and the Call element is required.

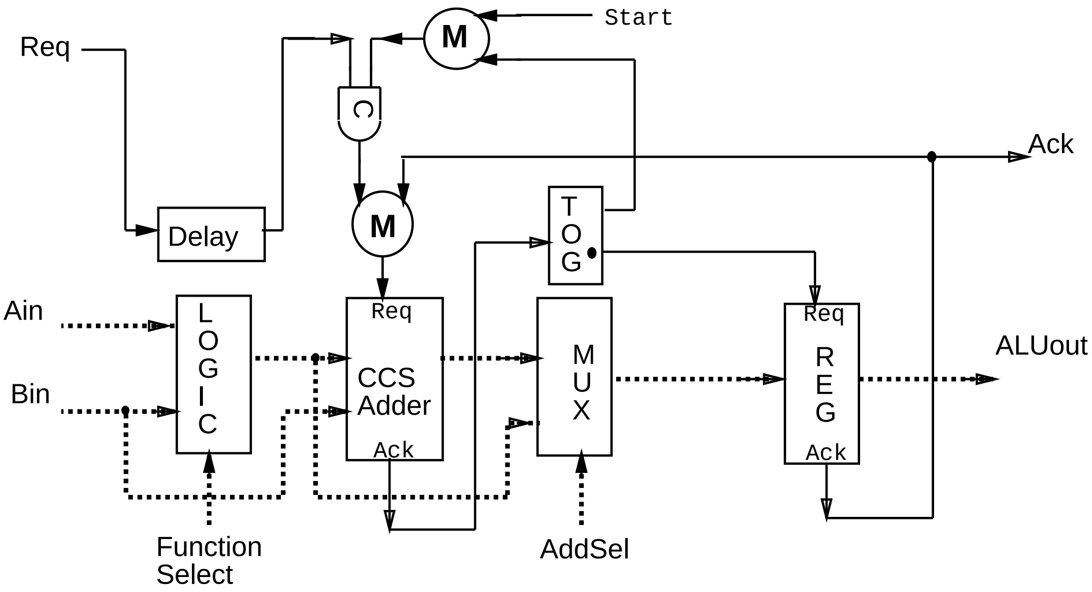

Figure 4.32: A CCS Arithmetic Logic Unit

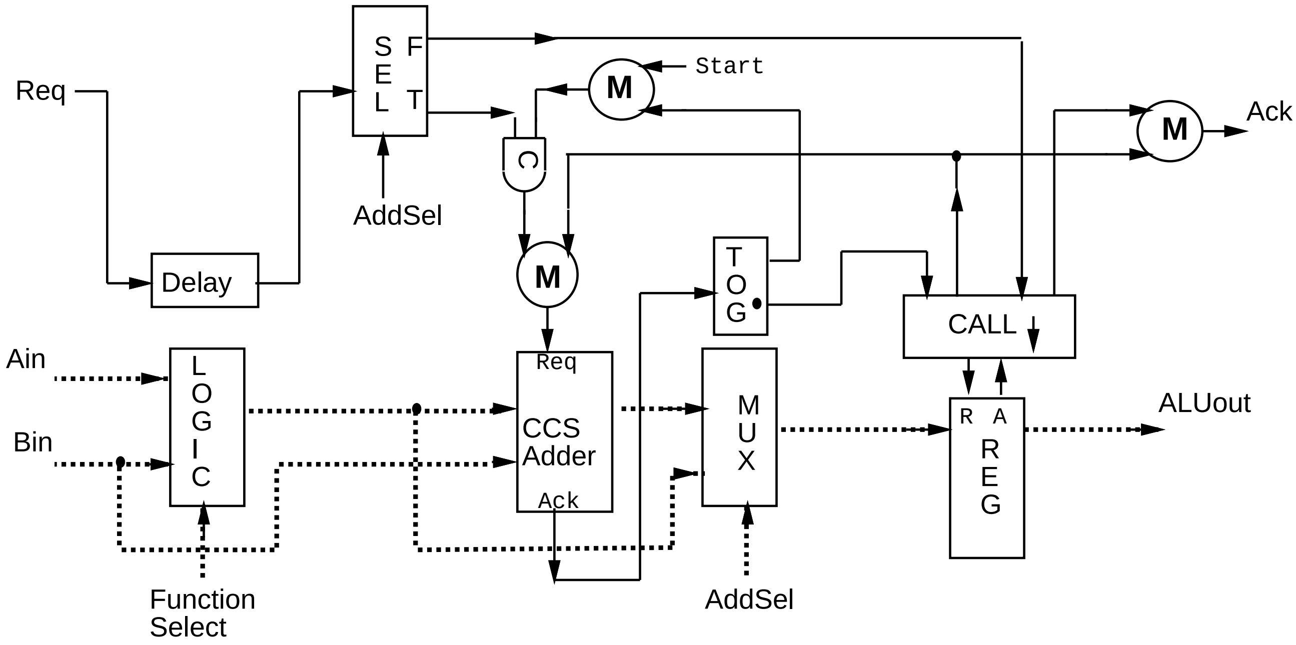

Figure 4.33: An Improved CCS Arithmetic Logic Unit