A Comparison of Modular Self-Timed Design Styles

Franklin Prosser, David Winkel, and Erik Brunvand

UUCS-95-025

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

February, 1995

Abstract

State-machine sequencing methods in modular 2-phase and 4-phase asynchronous handshake control are compared. Design styles are discussed, and the sequencers are tested against each other using a medium-scale minicomputer test design implemented in FPGAs. Seven 4-phase sequencers are tested. In these comparisons, 2- phase control is faster than 4-phase. Within the 4-phase designs, speed is enhanced when work is overlapped with handshake restoration. Performance of asynchronous and synchronous designs is compared.

A comparison of modular self-timed design styles

by Franklin Prosser, David Winkel, and Erik Brunvand

Abstract

State-machine sequencing methods in modular 2-phase and 4-phase asynchronous handshake control are compared. Design styles are discussed, and the sequencers are tested against each other using a medium-scale minicomputer test design implemented in FPGAs. Seven 4-phase sequencers are tested. In these comparisons, 2-phase control is faster than 4-phase. Within the 4-phase designs, speed is enhanced when work is overlapped with handshake restoration. Performance of asynchronous and synchronous designs is compared.

Introduction

This paper reports a comparison of design methods for asynchronous self-timed circuits with handshake control. We compare asynchronous and synchronous methods of state-machine control, with emphasis on the management of the sequencing process in modular 2-phase and 4-phase asynchronous designs. Simulations of a series of FPGA implementations of a benchmark processor show intriguing systematic trends.

Asynchronous design holds a promise of several potential benefits over its well-established synchronous counterpart: higher speed, lower power, and modularity of control structures. The outcomes for speed and power are still uncertain, but asynchronous handshake methods of control certainly provide attractive modular implementations.

Classical models of sequential state-machine control [15, 25] are deeply ingrained in most designers. In these models, control is embodied in a global circuit that is derived as a unit from the state-machine algorithm. Sequencing of events occurs through the state generator's production of the appropriate next state, usually through

an encoded state assignment. Such an implementation is not modular. Changes in the control algorithm usually require extensive re-derivation of the global control circuit. Even with good structured design techniques, the simplest change is likely to require backing up several steps in the circuit development. When studying the control circuit, one cannot clearly identify the individual steps of the algorithm.

One-hot control techniques [21, 25], in which one flip-flop is associated with each step (state) in the algorithm, have a certain modularity. However, a change in control algorithm requires recomputation of the input equations for one or more of the flip-flops.

These classical techniques are dominant in synchronous design, and are common in asynchronous work, especially in automated design systems [for instance, 14, 24, 28]. They are successful and work well, but are not modular.

On the other hand, asynchronous handshake techniques for state-machine control are naturally modular: the acknowledgement of the completion of one step in the sequence leads directly to a request for the next step. There is a strong flavor of passing a token downstream (or, temporarily, multiple tokens for parallel actions). The circuit diagrams show the algorithm steps in a direct way. Changes in the algorithm may be implemented by changing only the affected modules in the circuit and their connections to neighbors, without recourse to the recomputations of logic that characterize the traditional methods. This seems to us an extremely attractive feature of this form of asynchronous design.

Modularity encourages the development of libraries of useful modules that can be reused in other designs, similar to software libraries of objects or functions. Modularity also allows the designer to optimize the circuit incrementally for a variety of figures of merit. An initial design is likely to emphasize functionality, but once the circuit is functioning properly, localized pieces may be modified to improve other parameters such as timing performance without destroying the functionality. In traditional synchronous state-machine design, one must usually redesign the entire circuit.

The modular nature of the handshaking approach has led several workers to develop modules for the frequently used elements of control design. Clark, Molnar, Chaney, et al. at Washington University [13] were the first to use

this approach. Their macromodules constituted a significant advance in design technology; unfortunately the technology available at the time was inadequate to support efficient implementations, and the macromodule method did not achieve great commercial success. Sutherland's landmark Turing Award lecture [27] popularized the notion of a modular approach to control, with emphasis on pipeline operations with 2-phase handshaking. Martin [22, 23] and van Berkel [29, 30, 32] have developed automated systems for module-based asynchronous design, using 4-phase handshaking. Brunvand [7] and Gopalakrishnan and Akella [3, 17] have developed automated 2-phase systems, and Brunvand [6, 11] has produced libraries of 2-phase modules for a variety of technologies including CMOS, GaAs, and, recently, FPGAs.

Each group working with self-timed handshake circuits has adopted a particular style for managing the sequencing of steps in control algorithms. The style is strongly influenced and often determined by the nature of the chosen high-level description language and the target hardware technology. There have been few attempts to compare the practical effects of the various sequencing styles; the emphasis has been on other aspects of the design process. In our work, we focused explicitly on sequencing methods, and found an unexpected richness of behavior and a sometimes baffling subtlety.

Most systems for automation of the asynchronous design process make use of source languages that encourage a hierarchical view of control. The typical control program consists of a procedure that invokes other more detailed procedures. In such a view, it is important to have a mechanism whereby each level may initiate complex work and receive an explicit indication of when the entire work is done. Van Berkel's Tangram [31] and Martin's CSP-inspired production-rule system [22, 23] encourage a hierarchical viewpoint, as does Brunvand's Occam system [7, 12] .

On the other hand, typical state-machine algorithms may be described as “flat” in the sense that the highest level of control takes the form of a sequence of steps connected in a loop. Aside from the necessity of starting somewhere, there is no dominant state. In the hierarchical viewpoint, each sequence of activities has a definite end, whereas, in the flat viewpoint, there is no “end,” just one or more loops.

For reviews of basic asynchronous technology, see [8, 16, 26, 27]. Recent work in asynchronous design and

implementation appears in [4, 5, 19].

Asynchronous protocols for control and data:

For asynchronous control, the two dominant handshaking protocols are

> 2-phase (transition) signaling, in which each transition on the request (REQ) or acknowledge (ACK) signals represents an event;
>
> 4-phase signaling, in which only a positive-going transition on REQ or ACK initiates an event, and each signal must be "restored to zero" before the handshake cycle is completed.

In 2-phase signaling, each transition is meaningful, but for convenience of implementation it is assumed that each REQ and ACK pair operates "in-phase," and all signals start at zero. These assumptions are not required, and do not introduce inefficiencies. This protocol requires basic modules that are sensitive to transitions rather than levels. At the gate level, such modules tend to be more complex than level-driven modules.

In 4-phase signaling, the necessity to restore the signals to zero has interesting implications for design and performance. The restoration steps are unproductive activity, and proposals to overlap them with other productive work in order to improve performance have often been suggested [5]. This is a level-driven protocol that in theory can make use of simple circuitry at the gate level.

For this paper, we have studied both 2-phase and 4-phase protocols.

In addition to the management of control, one must assure that valid data has arrived at its destinations before it is used. This is an important problem where data paths may be long or of unknown or uncontrollable length. Two widely used data-handling protocols in asynchronous design are

the dual-rail data convention, in which each data bit is represented by two signals;

the bundled-data convention, in which each bit is represented by a single signal and delays are inserted in the control paths to assure that data has settled before its use.

In the dual-rail convention, a data bit is represented by one of the signal values 00 (meaning invalid data), 01 (bit is a valid 0), and 10 (bit is a valid 1). This convention has the advantage of providing a definite indication of the status of the bit, thereby permitting true delay-insensitive movement of individual data bits or groups of bits. Its main disadvantages are the doubling of the number of signal paths required for each data bit and the delays for circuit operation and completion detection. Dual-rail arithmetic units must be custom-designed, since they are unlikely to be found in standard data path libraries. Rigorous automated asynchronous design systems often use the dual-rail convention.

In the bundled-data convention, the designer must determine worst-case estimates of each data path for individual bits and groups of bits, and must insert appropriate ad hoc delays in the handshake control signals to assure that data is stable before a request initiates its use (the “bundling constraint”). The advantage lies in smaller data-path size (half the signal paths of dual-rail data), less data-management circuitry, and the likelihood of finding standard data-path modules in circuit libraries. The main disadvantage is its lack of assurance of delay-insensitivity on the data paths.

The disadvantages of each method of data handling are onerous. However, in many implementations, the data paths are sufficiently localized so that worst-case wire delays are close to typical delays. In these cases, bundling delays can be accurately estimated without seriously compromising performance [8]. For economy of chip area and simplicity of design, we have chosen to use the bundled-delay convention for most of our data paths, restricting the use of dual-rail data to cases where the time needed to develop a multi-bit result can vary widely. Unless explicitly stated, the bundled-data convention is assumed in this paper.

Asynchronous control:

The implementation of several generic algorithm actions are of special interest for our discussion: performance of alternative (conditionally executed) algorithm steps, concurrent (parallel) execution of steps, and sequencing of steps.

Conditional execution of algorithm steps: The selection of an alternative path determined by the value of a Boolean variable ("if-then-else") is performed in 2-phase and 4-phase signaling by means of an appropriate *select* element [27]; we discuss select elements later. The *merge* of the acknowledge signal paths following completion of the selected step is accomplished in 2-phase technology with an exclusive-or gate. In 4-phase technology, the merge requires a call element in order to merge the acknowledge signals into a downstream request and assure that the acknowledgment of the downstream request is routed back along the path of the proper alternative. Call elements for the various 4-phase signaling disciplines are discussed later.

Parallel (concurrent) actions: In either 2-phase or 4-phase signaling, concurrency is initiated by sending a request to more than one step, thereby spawning parallel execution of the steps. The parallel *fork* is, therefore, trivial to implement. To resynchronize the control upon completion of all concurrent steps requires a *join*. In 2-phase or 4-phase signaling, the join is in practice achieved with a Muller C-element, which retains its output until all inputs have the same value, and then copies the input(s) to the output.

Sequencing: The sequence is an essential part of the execution of an algorithm. In traditional design, whether asynchronous or synchronous, sequencing is embodied in a state generator whose task is to produce the next sequential state. When using handshake signaling, an acknowledgment of work done leads to the passing of control to the next step in the sequence, without regard to the remainder of the algorithm. One may envision passing a token from one step to the next.

Two-phase sequencing:

Sequencing with 2-phase (transition) signaling is straightforward [9, 12]. A sequential step is initiated by a transition on the REQ signal. When the work is completed, the ACK transition initiates the request for the next step in the sequence. In 2-phase, the sequencer is simply a wire connecting the ACK of one stage with the REQ of the next stage. Under the style used here, in which each sequential step stores its results for later use, there is no need for the work's acknowledgment to be communicated back to the previous step.

Four-phase sequencing:

Four-phase sequencing is more complicated, both conceptually and in implementation [23, 32], and gives rise to a number of phenomena that must be identified and managed. This is due to the need to restore request and acknowledge signals to zero, and a desire to exploit opportunities to overlap productive work with the restoration of various REQ and ACK signals.

Four-phase graphic convention:

In our 4-phase designs, we use a uniform sequencer graphic having six terminals, shown in Fig. 1. The six terminals form three request/acknowledge groups for (1) initiating a step (the input group IR and IA), (2) doing the work (the work group WR and WA), and (3) sequencing to the next step (the output group OR and OA). When appropriate conditions hold, the assertion of the input request IR will cause the work request WR to be asserted. The assertion of work acknowledge WA means that the work has been completed; this or subsequent phases of the restoration of the WR and WA signals will stimulate the output request OR. Subsequently, an output acknowledge OA will occur. The decision when to issue the input acknowledge IA is a key factor in the behavior of the different sequencers. Except at the end of a sequence (as explained later) all work that affects the environment is done within a work unit of a sequencer.

Types of 4-phase sequencers:

In each sequential stage, the sequencer's designated work is completed when the work acknowledge WA is asserted. However, in 4-phase sequencing the designer has several points during the WR/WA handshake when the sequencer may push on to the next sequential stage. We call this the "work-release" decision; it bears on when the request for the next step in the sequence (the output request OR) is asserted. We distinguish three types of work-release conventions:

Broad, in which the work's WR and WA must both be restored to zero before the work is released;

"Weak broad," in which the work is released when the work's WR falls;

Narrow, in which the work is released when the work's WA rises.

Figure 2 shows the relationship between work completion and work release for the three conventions. One might presume that speed advantages may be gained by use of narrow or weak-broad protocols instead of the traditional broad; our performance data support this presumption.

Another factor in 4-phase sequencing is the point at which the restorations of the output request OR and input acknowledge IA signals for a sequential step are performed, in relation to other steps in the sequence. The possibility of lingering request and acknowledge signals requires a careful look at such algorithm properties as conditional branching and sharing of resources.

We have examined four handshaking sequencers from the literature, three of them supporting the hierarchical model of algorithm expression, and one designed for pipeline applications. For this study, we developed a series of flat sequencers, herein called Winkel sequencers. We discuss control algorithm implementations using seven 4-phase sequencing elements:

Van Berkel sequencer using his S-element (Fig. 3.a) [32];

Martin sequencer using his D-element (Fig. 3.b) [23];

Martin sequencer using his Q-element (Fig. 3.g) [23];

Brunvand narrow sequencer, designed to support pipelining (Fig. 3.f) [8];

Winkel flat sequencers, in broad (Fig. 3.c), weak-broad (Fig. 3.d), and narrow (Fig. 3.e) forms.

The van Berkel and Martin sequencers assume a hierarchical view of sequencing, in which a supervisor initiates a sequence of work through an initial request and expects to be informed of the completion of the entire sequence through an acknowledge. The hierarchical model fits well with automated design techniques based on high-level languages such as CSP [20], Occam [7, 12], or Tangram [31]. However, the concept of state-machine loops is not well supported by this model.

The flat sequencers assume that the sequence is a part of a state-machine top-level loop, in which the completion of a "sequence" is recognized only by the initiation of another cycle of the sequence. These sequencers exploit opportunities to give an early input acknowledge IA (before the work is completed) and an early output request OR (before full restoration of the WR/WA handshake). Figure 4 summarizes the relationships among the sequencers in this study.

In Fig. 3, boxes enclose the van Berkel S-element (Fig. 3-a) and the Martin D- and Q-elements (Figs. 3-b and 3-g); these elements appear in the literature as building blocks for their respective sequencers. The Martin Q-element circuit is the same as the van Berkel S-element circuit, but the two elements are connected and used differently in their respective sequencers. For comparison, Fig. 3-h shows 2-phase sequencing expressed as a degenerate form of our general 6-terminal model.

In a hierarchical representation, a master process spawns work subprocesses of arbitrary complexity, involving sequencing, branching, and parallel activity. Sequencing within a process is accomplished with hierarchical sequencer elements, which report when all their subprocesses are finished.

In a flat representation, the top-level processor control algorithm is formulated as a branching sequence of work elements. Loops are permitted in the sequencing, and the cycle usually contains several sequential steps or states. No step in the top-level loop is "in charge." Each element of work may itself involve sequencing, branching, and parallel activity.

Only the top level of the algorithm is flat. Below the top level, sequencing is of necessity hierarchical, as can be seen as follows: Any sequencer requires an acknowledgment WA when its work is completed before further action occurs. If the work performed by the sequencer is complex enough to have internal sequencing, then the top-level work request WR will drive the input request IR of a sequencer at the lower level, and the top-level work acknowledge WA will be produced by an input acknowledge IA from the lower level. Since the flat sequencers are designed to give early IAs (before the work is completed), they are not suited for nested operations. For such sequencing below the top level, we use the van Berkel sequencer; these internal sequences tend to be short. Although only the top level may be implemented with flat sequencers, we found that flat implementations were noticably faster.

Accessing a shared resource:

In this context, a shared resource is one that is accessed from more than one point in the control algorithm, but not simultaneously. Examples are system architectural elements such as ALUs, multiplexers, and flip-flops. Actions on these resources are usually initiated and completed by REQ and ACK handshakes produced by the control algorithm, but the control of these elements is by means of level signals (for example, a multiplexer's select signals). Level control signals are assumed to be negated unless explicitly asserted. Assertion of level signals must be guided by the overarching REQ and ACK handshake signals.

In 2-phase handshaking, the *exclusive or* of REQ and ACK provides a convenient level to enable the assertion of architectural control signals. In 4-phase signaling, one may consider broad, weak-broad, and narrow forms of the level control signals: broad is enabled by the *or* of REQ and its ACK, weak-broad by just REQ, and narrow by the *and* of REQ and its negated ACK. In our 4-phase work, for efficiency we have uniformly chosen the weak-broad

(REQ-only) form of control signals for elements of the architecture, independent of the form of 4-phase sequencer.

In most state-machine designs there are a number of resources that are shared among (used by) the various work elements; the ALU, the registers, and the main data paths are examples. One must manage access to a shared resource so that there are no conflicts in its usage. In asynchronous handshake control, access to a shared resource is through a *call* element that allows only one access at a time to the resource and preserves knowledge of the return path so that the resource's ACK can be delivered to the proper point in the algorithm. Different implementations of the call are required for 2-phase and 4-phase broad, weak-broad, and narrow work-release. Figure 5 shows our call elements. (The call element thus may be used to implement an algorithmic control element supporting calls to subprograms or an architectural element supporting access to shared resources.)

Our 2-phase designs use the call element [6] shown in Fig. 5-a. Since 2-phase events are signaled by transitions, and since the handshaking protocol requires an acknowledgment prior to issuing a subsequent request, there are no problems with conflicting request inputs to the call element.

For 4-phase calls used with broad-work-release sequencers, we use the 4-phase broad call element [8] shown in Fig. 5-b, which assumes that a new request for the resource will not arrive until any prior request and its acknowledgement both have fallen. One is assured that when work commences in the next step of the sequence there is no residual assertion of the preceding WR or WA. This call element cannot accommodate concurrent requests or acknowledges. This is the most compact of the asynchronous call elements.

With 4-phase weak-broad sequencers, we use our weak-broad call element, shown in Fig. 5-c. This element assumes that a new request will arrive only after any prior request to this call module has fallen. However, at the time of arrival of a new request, there is no requirement that the prior acknowledge has fallen. Upon receipt of a request, the call element issues its subprogram request. The call element blocks an acknowledgment of the new request until any prior request's acknowledgment has fallen. This element cannot tolerate concurrent requests, but can accommodate overlapping acknowledges.

With 4-phase narrow sequencers, we use our narrow call element, shown in Fig. 5-d. In this element, a new request

is permitted to arrive while a prior request is still active. Upon receipt of a request, the call element blocks the request until any prior request has fallen, and it blocks the acknowledge until any prior acknowledge has fallen. This element can accommodate concurrently active requests and acknowledges, such as may arise with narrow work-release sequencers. It is the most complex of the asynchronous call elements that we use.

Use of the appropriate call element to access a shared resource assures the resolution of any conflicts that originate from multiple active work request signals WR (in the case of the narrow sequencers) or work acknowledge signals WA (for weak-broad or narrow sequencers).

Control of branching:

The type of sequencer has an effect on how branching is controlled. In 2-phase signaling, the select element [6] is a state-holding device that makes its decision at the time of the transition on the input request, producing a transition on the true or the false output request line; the select condition (a level input) must be stable at the time of the input-request transition but subsequent changes on the select-condition input will not affect the proper operation of the selector.

In 4-phase signaling with the bundled-data convention, the selector can be a simple combinational circuit--a demultiplexer--that copies the behavior of the input request line to either the true or false output request line. However, in this case, the select-condition input must remain stable throughout the assertion of the input request, else the direction of control may change in mid-computation! If subsequent actions may alter the select-condition input before the sequencing protocol has removed the input request, this simple scheme will not work. Such circumstances can arise with most of the 4-phase sequencers if the processor's control algorithm causes such a change in the select condition. As will be seen, the circumstances arise with a vengeance with narrow-work-release sequencing.

Where the integrity of the conditional selection is threatened, one must use a latched selector, which latches the select condition whenever the input request is asserted and passes the condition otherwise. Although a

combinational selector will often suffice for 4-phase handshake control, designs will usually require some latched selectors. An inspection of the detailed design is required. Alternatively, if efficiency is not an issue, latched selectors may be used throughout the design.

In all forms of asynchronous REQ/ACK handshaking, if any select condition may undergo a change near the time of issuing of the selector's input request, then fundamental mode may be violated, and one must use a Q-select element [6] or its equivalent that deals with possible metastable behavior. A example is the examination of an interrupt-request signal or other signal arising from the outside, where the local circuit has no knowledge of the timing of the incoming signal.

Figures 6 and 7 show a typical segment of a state-machine algorithm implemented in 4-phase flat broad style and in 2-phase style. The illustrative segment shows sequencing, conditional branching, and parallel execution. In the 4-phase segment, within the complex work in stage 2 a hierarchical (van Berkel) sequencer is required. The top level in both the 4-phase and 2-phase illustrations will eventually loop back. In a pure 4-phase hierarchical implementation, the top level would, as would all levels, come to an end where the final acknowledge would feed back to the source of the sequence.

Characterization of the 4-phase sequencers:

Figure 8 shows Petri net descriptions of the behavior of the 4-phase sequencers assuming a valid operating environment. Each Petri net is comprised of three full handshakes, for the input request and its acknowledge, for the work request and acknowledge, and for the output request and acknowledge; these are shown in Fig. 8.h. We assume that the environment causes no violations of the handshake conventions on any of the three handshakes. The sequencers differ in the details of the constraints imposed by each of the three handshake sequences on their two counterparts.

Work typically occurs under the jurisdiction of a sequencing element through its WR/WA handshake. However, when using the hierarchical sequencers the last work in a sequence may be performed without an explicit

sequencer; the work may be driven directly from the OR/OA of the previous stage.

As we have seen, a concern is whether a combinational 4-phase select element will suffice, or whether a latched element must be used. Frequently, select elements lie between sequencing elements, as in the "SKIP" select on the top row of Fig. 6. Trouble arises if SKIP changes while the select element's input request signal is active, and this is strongly dependent on the behavior of the sequencer. For instance, in van Berkel sequencing, input request signals linger until the completion of all steps in the sequence. Any later step in the sequence that modifies SKIP may cause the old branch path to change, with disastrous results. Clearly, latched selects will be needed unless the integrity of the select condition can be assured. Similarly, with the Winkel flat narrow sequencer, it is possible (though unlikely) that input request signals linger throughout all stages of the loop. On the other hand, with the Winkel flat broad and weak-broad sequencers, requests may linger until the completion of one additional sequential stage.

Here we summarize the behavior of each of the sequencers. An examination of the Petri nets and the circuits for the sequencers will reveal the detailed signal behavior.

Van Berkel S-element sequencer (Figs. 3.a, 8.a):

This is a hierarchical sequencer, with broad work release. It stacks sequential requests until all elements in the sequence are completed. IA rising means this work and all subsequent work in this sequence are broad-released. IA falling means this and all subsequent elements in this sequence have restored to zero. OR rising means this work is broad-released. This sequencer is useful with complex work sequences where a hierarchical sequencer is needed (nested control) to assure completion of all elements of the sequence before any acknowledgment is issued.

Martin D-element sequencer (Figs. 3.b, 8.b):

This sequencer is hierarchical, with broad work release. It gives an early acknowledge, which persists until the end of the sequence. IA rising means this work has been acknowledged. IA falling means that all subsequent sequence elements have restored to zero. OR rising means that this work is broad-released. Sequence requests tend to drop

early; sequence acknowledges linger. This sequencer is useful in hierarchical control when a quick acknowledge is desired, while retaining an indication of when the entire sequence is finished.

Winkel flat broad sequencer (Figs. 3.c, 8.c):

Work release is broad, with a quick input acknowledge IA after the work is acknowledged. WR rises with IR. IA falls when the current work is broad-released (WR and WA are fully restored). OR rising means both this sequencer's WR/WA and IR/IA handshakes are fully restored.

This sequencer is useful for flat sequencing as in a sequential state machine; it and the other flat sequencers support cyclic control flow. This sequencer provides an early input acknowlegement while waiting for broad-released work before issuing the output request OR. Proper operation of the next stage in the sequence requires that no new input request appear before OA has risen; this condition will hold within state-machine cycles of length 2 or greater. This and the other flat sequencers are not useful within complex work sequences that themselves involve sequences of steps. Here, a hierarchical (nesting) sequencer is needed to assure completion of all steps in the sequencer's work before the work is acknowledged.

Winkel flat weak-broad sequencer (Figs. 3.d, 8.d):

This sequencer proceeds after weak-broad-released work, with a quick input acknowledge IA after the work is acknowledged. WR rising means that IR has risen and OA is low; this prevents deadlock in cyclic sequences with very slow restoration of OA. IA falls when the current work is broad-released. OR rising means that this sequencer's work request WR has fallen (weak-broad release) and that OA is low (assuring that there are no lingering parts of the restore phase of previous cycles of the state machine).

This sequencer is useful for flat sequencing. It provides an early acknowlegement while waiting for weak--broad-released work before issuing the output request OR. Proper operation of the next stage in the sequence requires that no new input request appear before OA has risen; this condition will hold within state-machine cycles of length 2 or greater.

Winkel flat narrow sequencer (Figs. 3.e, 8.e):

Work release is narrow, with a quick input acknowledge IA after the work is acknowledged. WR rising means that IR has risen and OA is low; this prevents deadlock in cyclic sequences with very slow restoration of OA. IA falls when the current work's WR and WA are fully restored. OR rising means that this sequencer's work has been acknowledged (narrow release) and that OA is low (assuring that there are no lingering parts of the restore phase of previous cycles of the state machine).

This sequencer is useful for flat sequencing. It responds to completion of the work (WA rising) by providing early input acknowlegment IA and output request OR signals. Proper operation of the next stage in the sequence requires that no new input request appear before OA has risen; this condition will hold within state-machine cycles of length 2 or greater.

Brunvand's narrow sequencer (Figs. 3.f, 8.f)

Work release is narrow, with a quick input acknowledge after the work is acknowledged. OR rising means the current work is narrow-released. IA falls when the work's WR and WA have fully restored.

This sequencer is useful with pipelines (FIFOs), in which highly parallel processing on unshared resources is desired. In such an application, many elements of the pipeline may be active at once, although they do not share common resources such as ALUs. It is also useful for flat sequencing, but, like the flat sequencers, is not useful within complex work sequences.

Martin Q-element sequencer (Figs. 3.g, 8.g):

This sequencer has radically different properties from the preceding sequencers. Work release is broad. Input requests persist; all requests become active before any work is performed. In practice, this behavior precludes conventional branching within a sequence, since the hardware has committed to a sequence of steps before any work has been performed, and there is no opportunity for work within the sequence to determine the values of

select conditions used later in the sequence. Thus, this sequencer is not likely to be useful for sequential state-machine control, except under special conditions. We have not used this sequencer in our tests.

Testing the sequencers:

To examine the properties of asynchronous sequencers, we chose a well-known processor of moderate size and complexity--the Digital Equipment Corporation PDP8. The PDP8 is a commercially successful processor whose design is not undergoing revision. The complete processor, with its simple and sparse instruction set, limited register set, small (12-bit) word, and rich set of diagnostic programs, has proven to be ideal as a testbed [25]. It is small enough to be assembled in a MOSIS tiny chip, an Actel FPGA integrated circuit, or a small number of PALs, yet large enough to incorporate the basic elements of more complex processors.

For the present work we built a series of asynchronous PDP8 designs (and one synchronous design) implemented in Actel FPGAs, and from these came interesting observations on the performance of self-timed handshake-driven state machines. We built over a dozen self-timed designs, each using a form of 2-phase or 4-phase request/acknowledge handshaking protocol for control. Our designs use bundled-delay data elements (one wire per bit) [8, 26], except that to maximize the speed of arithmetic our ALU uses a dual-rail carry chain to support carry-completion-detection addition [11].

Our PDP8 designs follow as closely as practical the architecture and control algorithm presented in [25], modified where necessary to assure an efficient implementation. The main data path consists of a set of registers that can be admitted to the two inputs of an ALU, where an arithmetic, logical, or shifting operation may be performed, with the result stored in one or more of the registers. Several Boolean variables are stored in flip-flops. The algorithm derives from an ASM (algorithmic state-machine) view that is inherently cyclic, consisting mainly of a loop with instruction fetch states and execute-phase states. The main data path is heavily used throughout the algorithm. Where possible, actions are performed in parallel, but the ALU is not pipelined and can support only one action at a time.

The designs were drafted and simulated using ViewLogic's Powerview Release 5.1, particularly the ViewDraw and ViewSim tools [33], with the aid of Actel's Action Logic System (ALS) chip place-and-route software [1]. In the 2-phase designs we used Brunvand's 2-phase Actel building blocks [10] where possible; where necessary, we constructed additional building blocks for 2-phase and 4-phase signaling. Brunvand [11] has discussed techniques for building asynchronous systems in FPGAs, and has described Actel implementations [2] of many of the common asynchronous building blocks.

Performance of the designs:

We have verified our Actel FPGA designs for the PDP8 with detailed delay-back-annotated simulations. We report times for the simulations using two commercial PDP8 diagnostic test programs executed from a simulated RAM of negligible delay. The diagnostic test INST1 performs intensive testing of the processor instructions without using input/output or interrupts. The diagnostic INST2 performs testing of the “auto-indexing” capabilities (RAM-based index registers with automatic incrementing of contents) and the interrupt system. In this test, input/output is used only to generate interrupt requests, and takes negligible time. Times are for one simulated execution of the main loop in each diagnostic routine.

Table 1 shows simulated timing results and the number of Actel modules required for the processor chip. In addition to the asynchronous designs, we also report timings for an Actel FPGA version of Prosser and Winkel's standard synchronous design [25], simulated with both typical-delay and worst-case-delay module timing parameters.

The results for the Actel FPGA implementations show definite trends: The synchronous design is substantially faster and smaller than any asynchronous design. The 2-phase design is faster than any 4-phase design, and is comparable in size to the typical 4-phase design. The flat 4-phase designs are faster but larger than either hierarchical 4-phase design. The flat 4-phase designs show the anticipated speed relationship (narrow fastest, broad slowest), even though there is a substantial inverse size relationship.

We doubt that fine-tuning the designs or changing hardware technologies would change the observed timing trends in the asynchronous 4-phase designs. It is tempting to conclude that there are indeed significant speed gains to be found by exploiting overlaps of productive work with the restoration of various REQ and ACK signals.

One might ask if the observed trends between synchronous and asynchronous performance and between 2-phase and 4-phase performance have general applicability.

First, the 2-phase versus 4-phase comparisons: The standard module used within the Actel FPGA is an enhanced 4-input 1-output multiplexer. From this module can be constructed a wide variety of complex multi-input gates, multiplexers, latches, and flip-flops, each such construction having roughly the same propagation delays. Of particular importance for our asynchronous technology, the 2-input exclusive-or, the Muller C-element, and the asymmetric C-element [23] are each easily constructed from a single module. Compared with custom CMOS, this technology may give a slight advantage to 2-phase over 4-phase control implementations, since the exclusive or, which appears frequently in 2-phase designs, is a single Actel module. On the other hand, in the Actel FPGA, 4-phase elements such as the selector and the broad call are implemented more efficiently than their 2-phase counterparts.

The 2-phase PDP8 implementation is straightforward and intrinsically streamlined. Once a request or acknowledge transition occurs, there is no further adjustment of the signal. No special hardware is used to perform sequencing. See Figs. 3.h and 7. We believe that this factor is responsible for the speed advantage of our 2-phase circuit over the 4-phase circuits. However, the advantage is only about 20%, which may not be significant.

Second, the synchronous versus the asynchronous designs: Using standard typical-delay simulation parameters, the synchronous design is 38% faster than the fastest asynchronous design, and less than half the size of the smallest asynchronous design. The synchronous control circuitry is only about one-third the size of the asynchronous control. Such significant differences were a surprise. Are they real?

Whereas the asynchronous designs are self-timed, the timings for the synchronous design were achieved by increasing the simulated clock speed until the diagnostic programs failed. In practice, one must derate the clock

speed to assure proper performance under the normal variety of real conditions, thus sacrificing some of the speed advantage of the synchronous design. A sensible way to derate the simulated synchronous timings is to use Actel's maximum (worst-case) module-delay parameters rather than the typical-delay parameters used in the standard simulations. With synchronous technology, we must be prepared for worst-case conditions, whereas we should expect the typical asynchronous circuit to run at typical-delay speeds. Running with Actel's maximum module-delay parameters increases the simulated times by 38%--by coincidence making the derated synchronous times essentially the same as the fastest asynchronous typical-delay times.

In all our asynchronous designs, resources such as the ALU and the registers are shared from many places in the control algorithm. The control of the access to these resources results in call elements with many request inputs, approaching 30 inputs for the ALU in our PDP8 designs. Since the size of the call elements grows linearly with the number of inputs (the time grows logarithmically), considerable chip real estate is consumed with these control structures. This is probably a principal reason the synchronous design is smaller and faster than its asynchronous counterparts. It is a price one pays for modularity, at least in systems with heavily shared resources.

One might adopt a design technique that minimizes the size of the call structures, with an associated increase in size of random logic and selectors, and at the cost of reducing the clarity of the design. In fact, our designs use this technique in several places where the application is rather obvious, and where good structured design practices may be preserved. For instance, our initial 2-phase design (not reported in Table 1), which adhered most closely to the spirit of modular design, used a 37-input call to the ALU. Designing so that the size of the ALU call is lowered to 23 (in the reported design), with more modest lowerings in size of register-load calls, resulted in a 9% increase in speed and a 12% decrease in size. These modifications were made to operations in the critical path of the processor's instruction-execution algorithm. We expect continued improvements in speed and size through additional shrinking of the calls, but with decreasing benefits.

We have made no systematic attempt to shrink the data bundling delays in the asynchronous designs to their minimum. Such an effort might marginally improve the speed of the asynchronous designs.

From the module counts for the Actel designs reported in Table 1, one sees that the control circuitry dominates

each design. The small 12-bit word of the PDP8 results in a relatively small commitment of hardware for architecture. In processors with larger register sizes and more extensive data paths, the relative commitment to control would be less.

We are currently constructing custom CMOS versions of our PDP8 designs in order to make similar comparisons within this technology. Among the design differences caused by technology is the absence in Actel of three-state elements within the chip. In Actel, the bussing of the registers is done with multiplexers rather than with three-state buffers. Although we are not certain of the overall effect of this factor, we suspect that the custom CMOS designs will gain a speed advantage through the use of three-state technology. However, this advantage will apply to all the designs, including the synchronous versions.

Summing up the results of our study of PDP8 designs in Actel FPGAs:

- Modular design with handshake signals is flexible and elegant.
- Two-phase design is easier and more attractive than 4-phase.
- Four-phase design demands careful attention to subtleties of conditional branching and sharing of resources.
- Our flat 4-phase designs are faster than the hierarchical designs.
- In 4-phase design, early work release gives faster circuits.
- In this study, conventional synchronous design still beats asynchronous handshake design (although the times may be close).

Within the 4-phase designs, the timings show a trend in favor of sequencers that exploit early input acknowledge and early work release: the flat sequencers outperform the hierarchical sequencers, and the flat sequencers that use narrow work release outperform their weak-broad and broad counterparts. This seems strong justification for pursuing the overlap of useful work with 4-phase signal restorations.

Our best asynchronous design averages about 40% slower than our non-derated synchronous design and about the same speed as a derated version. It is intriguing that the synchronous design performs so well compared with its asynchronous counterparts. One might speculate that the most important role of asynchronous design technology is in implementing obviously asynchronous processes such as in pipelines and inter-process communication, and in situations where the modularity of the asynchronous handshake paradigm is an important factor, rather than in the implementation general processors.

Even though most asynchronous methods seem inherently more difficult than synchronous methods, we find modular 2-phase asynchronous handshake designs easy to construct and extremely easy to modify in comparison with conventional synchronous state-machine synthesis. Whether this ease of design can overcome the other advantages of synchronous design remains an open question.

References:

[1] Actel Corporation, *Action Logic System (ALS) Release 2.11 User's Guide,* April 1992.

[2] Actel Corporation, *Actel FPGA data book and design guide,* 1994.

[3] V. Akella and G. Gopalakrishnan, "Specification, simulation, and synthesis of self-timed circuits," *Proc. 26th Annual Hawaiian International Conf. on System Sciences,* Vol. 1, pages 399-408, 1993.

[4] *Proc. Async94: International Symposium on Advanced Research in Asynchronous Circuits and Systems,* Erik Brunvand and Alan Davis, general chairs, Salt Lake City, Utah, November 3-5, 1994.

[5] *Proc. VII Banff Workshop: Asynchronous Hardware Design,* G. Birtwistle, convener, Banff, Canada, August 28-September 3, 1993.

[6] E. Brunvand, "Parts-R-Us: a chip apart(s)...," Technical report CMU-CS-87-119, 1987.

[7] E. Brunvand and R.F. Sproull, "Translating concurrent programs into delay-insensitive circuits," *Proc. International Conf. on Computer-Aided Design '89,* IEEE, pages 262-265, November 1989.

[8] E. Brunvand, Translating concurrent communicating programs into asynchronous circuits, Ph.D. thesis, Carnegie Mellon University, 1991. Available from the Computer Science Department as Technical Report CMU-CS-91-198.

[9] E. Brunvand, "Examples of translating concurrent communicating programs into asynchronous circuits," class notes, Computer Science Department, University of Utah, 1991.

[10] E. Brunvand, "A cell set for self-timed design using Actel FPGAs," Technical Report UU-CS-91-013, Computer Science Department, University of Utah, 1991.

[11] E. Brunvand, "Using FPGAs to implement self-timed systems," *J. VLSI Signal Processing 6,* pages 173-190,

1993.

[12] E. Brunvand, "Designing self-timed systems using concurrent programs," *J. VLSI Signal Processing 7,* pages 47-59, 1994.

[13] W.A. Clark, "Macromodular computer systems," in *Proc. Spring Joint Computer Conf.,* AFIPS, April 1967.

[14] B. Coates, A. Davis, and K.S. Stevens, "Automatic synthesis of fast compact self-timed control circuits," *Proc. IFIP Working Conf. on Design Methodologies,* Manchester, England, pages 193-208, April 1993.

[15] A.D. Friedman, *Fundamentals of Logic Design and Switching Theory,* Potomoc, Maryland: Computer Science Press, 1986.

[16] G. Gopalakrishnan and P. Jain, "Some recent asynchronous system design methodologies", Technical Report UU-CS-TR-90-016, Computer Science Department, University of Utah, 1990.

[17] G. Gopalakrishnan and V. Akella, "VLSI asynchronous systems: specification and synthesis," *Microprocessors and Microsystems 16,* No. 10, pages 517-527, 1992.

[18] G. Gopalakrishnan and L. Josephson, "Towards amalgamating the synchronous and asynchronous styles," in *Proc. TAU 93: Timing Aspects of VLSI,* ACM, Malente, Germany, September 1993.

[19] G. Gopalakrishnan and E. Brunvand, eds, "Special issue on asynchronous systems," *Integration, The VLSI J. 15,* 1993. Proc. Minitrack on Asynchronous Circuits and Systems, held at 26th Annual Hawaiian International Conf. on System Sciences, Maui, Hawaii, Jan 6-8, 1993.

[20] C.A.R. Hoare, *Communicating Sequential Processes,* Englewood Cliffs, N.J.: Prentice-Hall, 1985.

[21] L. Hollaar, "Direct implementation of asynchronous control units," *IEEE Trans. on Computers, C-31,* (12), pages 1133-1141, 1982.

[22] A. Martin, “Compiling communicating processes into delay-insensitive circuits,” in *Distributed Computing 1*, No. 3, 1986.

[23] A. Martin, “Synthesis of asynchronous VLSI circuits,” *in Proc. VII Banff Workshop: Asynchronous Hardware Design,* 1993.

[24] S.M. Nowick and D.L. Dill, “Automatic synthesis of locally-clocked asynchronous state machines,” *Proc. 1991 IEEE International Conf. on Computer-Aided Design, pages 318-321, 1991.*

[25] F. Prosser and D. Winkel, *The Art of Digital Design,* Englewood Cliffs, N.J.: Prentice-Hall, 1987.

[26] C.L. Seitz, “System timing,” in C. Mead and L. Conway, *Introduction to VLSI Systems,* Chapter 7, Reading, Mass.: Addison-Wesley, 1980.

[27] I. Sutherland, “Micropipelines,” *Communicat. ACM 32,* pages 720-738, 1989.

[28] S.H. Unger, *Asynchronous Sequential Switching Circuits,* New York, N.Y.: Wiley-Interscience, 1969.

[29] K. van Berkel, M. Rem, and R. Saeijs, “VLSI programming,” in *Proc. International Conf. on Computer Design '88,* 1988.

[30] K. van Berkel and R. Saeijs, “Compilation of communicating processes into delay-insensitive circuits,” in *Proc. International Conf. on Computer Design '88,* 1988.

[31] K. van Berkel, J. Kessels, M. Roncken, R Saeijs, and F. Schalij, “The VLSI programming language Tangram and its translation into handshake circuits,” *Proc. European Design Automation Conf.*, pages 384-389, 1991.

[32] K. van Berkel, “Handshake circuits: an asynchronous architecture for VLSI programming,” in *Proc. VII Banff Workshop: Asynchronous Hardware Design,* 1993.

[33] Viewlogic Systems, Inc., Powerview Release 5.1, 1993.

Table 1. Performance of sequencer test designs

Sequencer		Execution time * INST1	Execution time * INST2	Actel modules ** Total	Actel modules ** Control
2-phase		3.52	3.49	1000	695
4-phase hierarchical:					
Van Berkel S-element		5.26	5.09	864	583
Martin D-element		5.45	5.26	889	608
4-phase flat:					
Winkel flat broad		5.21	5.09	915	634
Winkel flat weak-broad		4.78	4.61	1095	814
Winkel flat narrow		4.41	4.33	1223	942
Brunvand narrow		4.25	4.16	1223	942
Synchronous	(typical delays)	2.58	2.51	403	245
	(worst-case delays)	3.54	3.43	403	245

* Simulated time in msec for execution of PDP8 diagnostic main loop, using Actel's typical-delay module timing parameters unless noted.

** Using Actel 1280 series 160-pin PQFP chip (1232 total modules)

Manuscript received _____

Affiliation of authors:

Franklin Prosser and David Winkel are with the Computer Science Department, Indiana University, Bloomington, IN 47401. This work was done while Prosser and Winkel were on leave at the University of Utah.

Erik Brunvand is with the Computer Science Department, University of Utah, Salt Lake City, UT 84112.

Figure captions:

Fig. 1. Four-phase sequencer graphic.

Fig. 2. Work-release conventions.

Fig. 3. Four-phase sequencer elements.

Fig. 4. Sequencer taxonomy.

Fig. 5. Call elements.

Fig. 6. Typical 4-phase control flow.

Fig. 7. Typical 2-phase control flow.

Fig. 8. Petri nets describing 4-phase sequencer behavior.

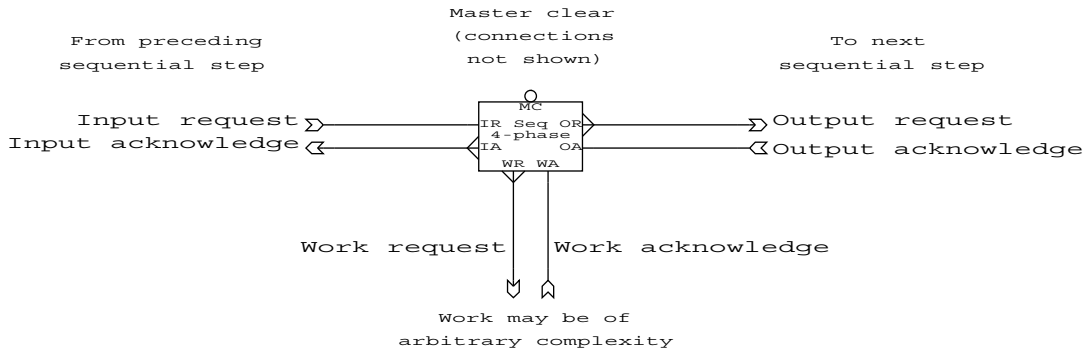


Fig. 1. Four-phase sequencer graphic.

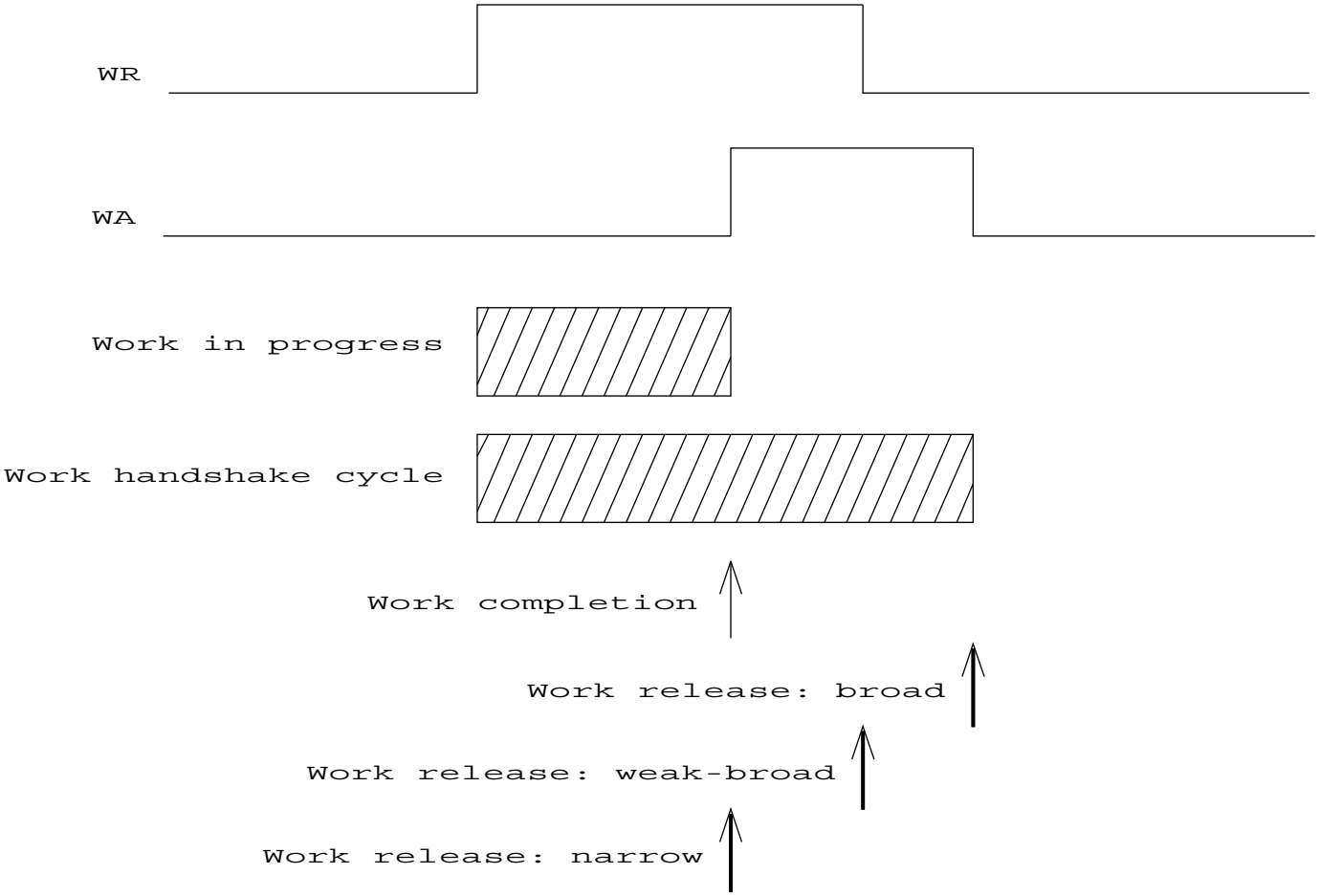
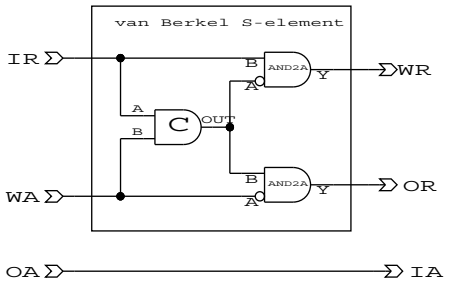
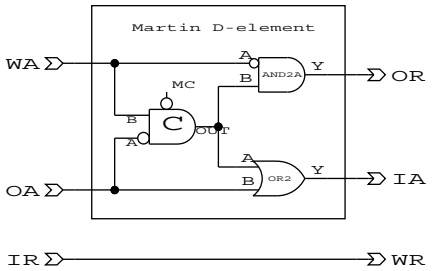


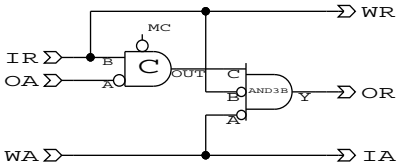
Fig. 2. Work release conventions.



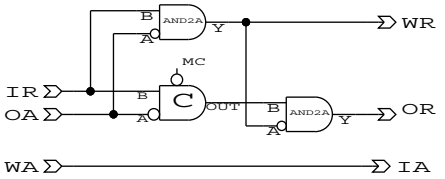
(a) van Berkel sequencer
(hierarchical)



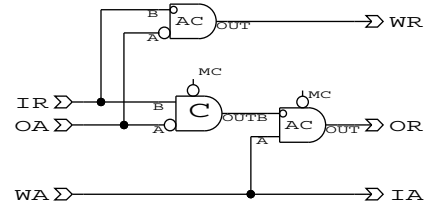
(b) Martin D-element sequencer
(hierarchical)



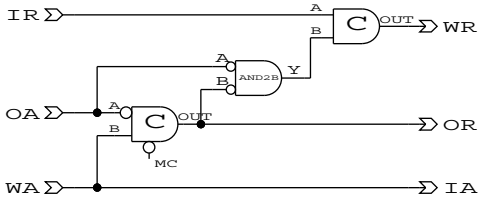
(c) Winkel broad sequencer
(flat)



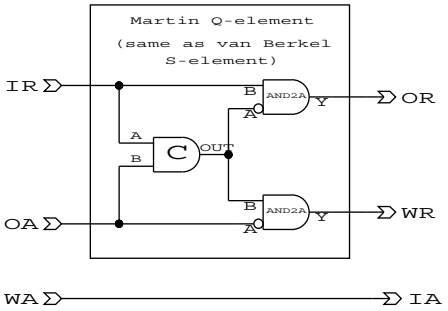
(d) Winkel weak-broad sequencer
(flat)



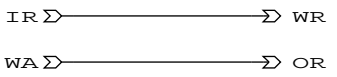
(e) Winkel narrow sequencer
(flat)



(f) Brunvand narrow sequencer
(pipeline)

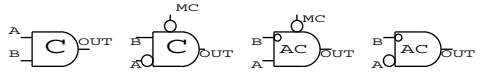


(g) Martin Q-element sequencer
(hierarchical)



(h) 2-phase sequencing
using the 6-terminal model

C-element behavior



A	B	OUT′	OUT′	OUT′	OUT′
L	L	L	OUT	L	L
L	H	OUT	H	OUT	H
H	L	OUT	L	L	L
H	H	H	OUT	H	OUT

Fig. 3. Four-phase sequencer elements.

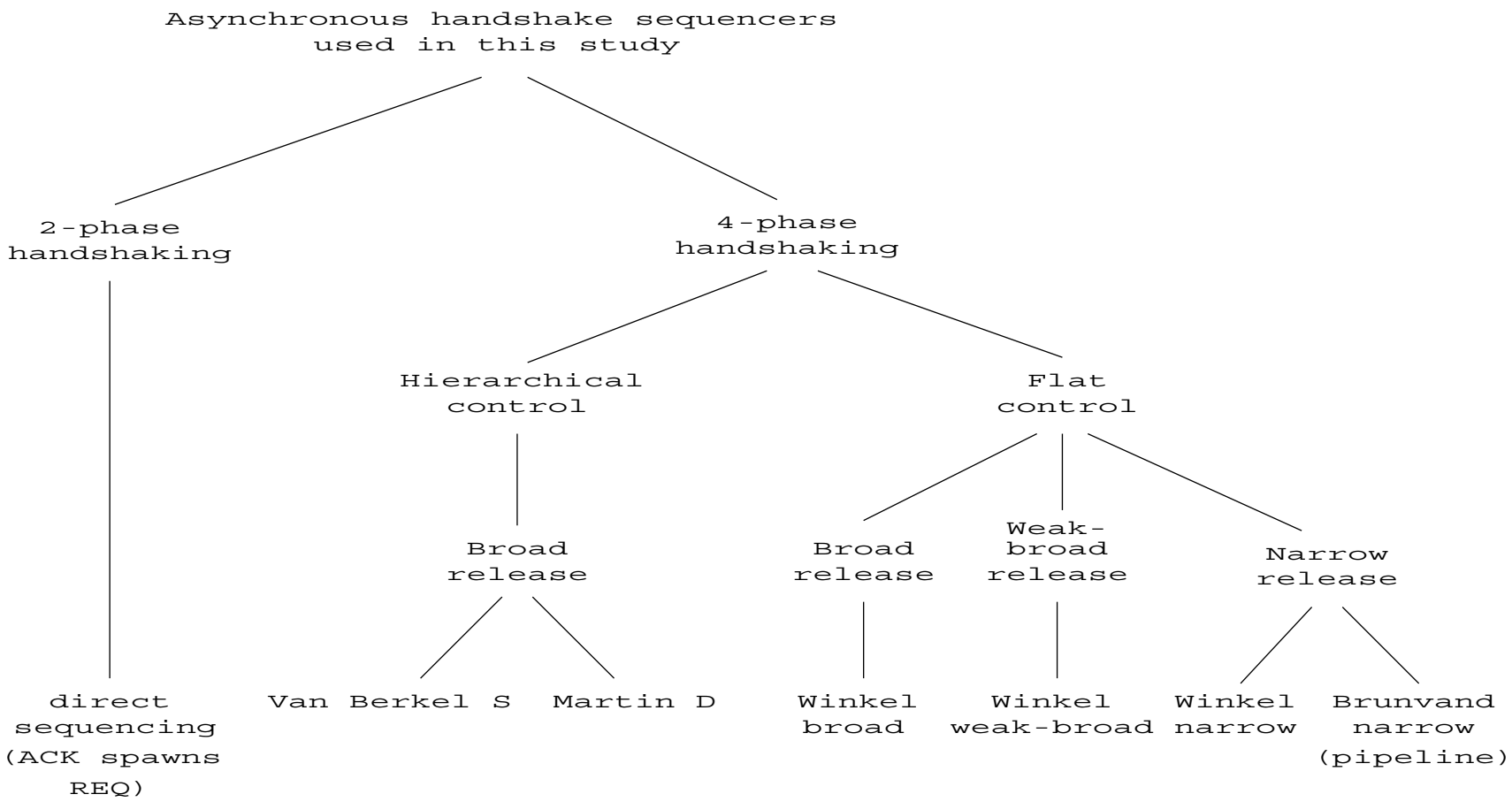
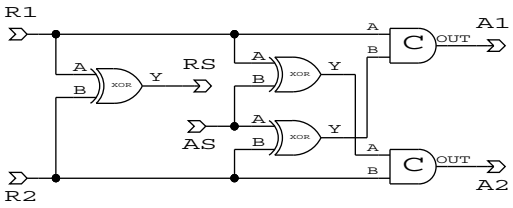
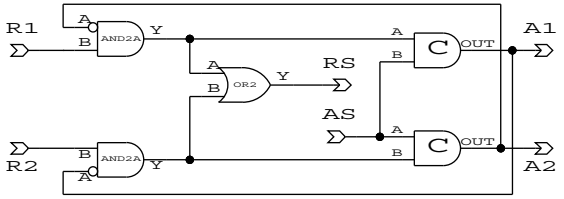


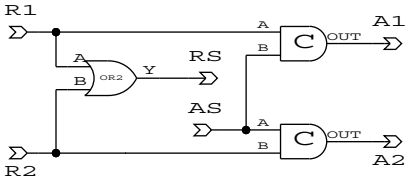
Fig. 4. Sequencer taxonomy.



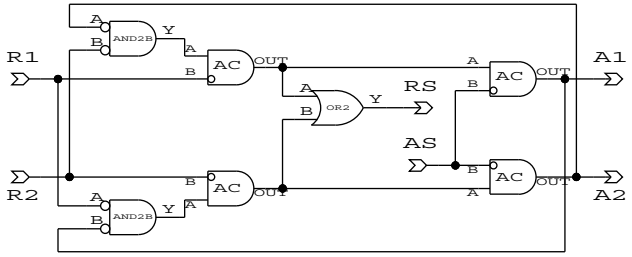
(a) 2-phase call



(c) 4-phase weak-broad call



(b) 4-phase broad call



(d) 4-phase narrow call

Fig. 5. Call elements.

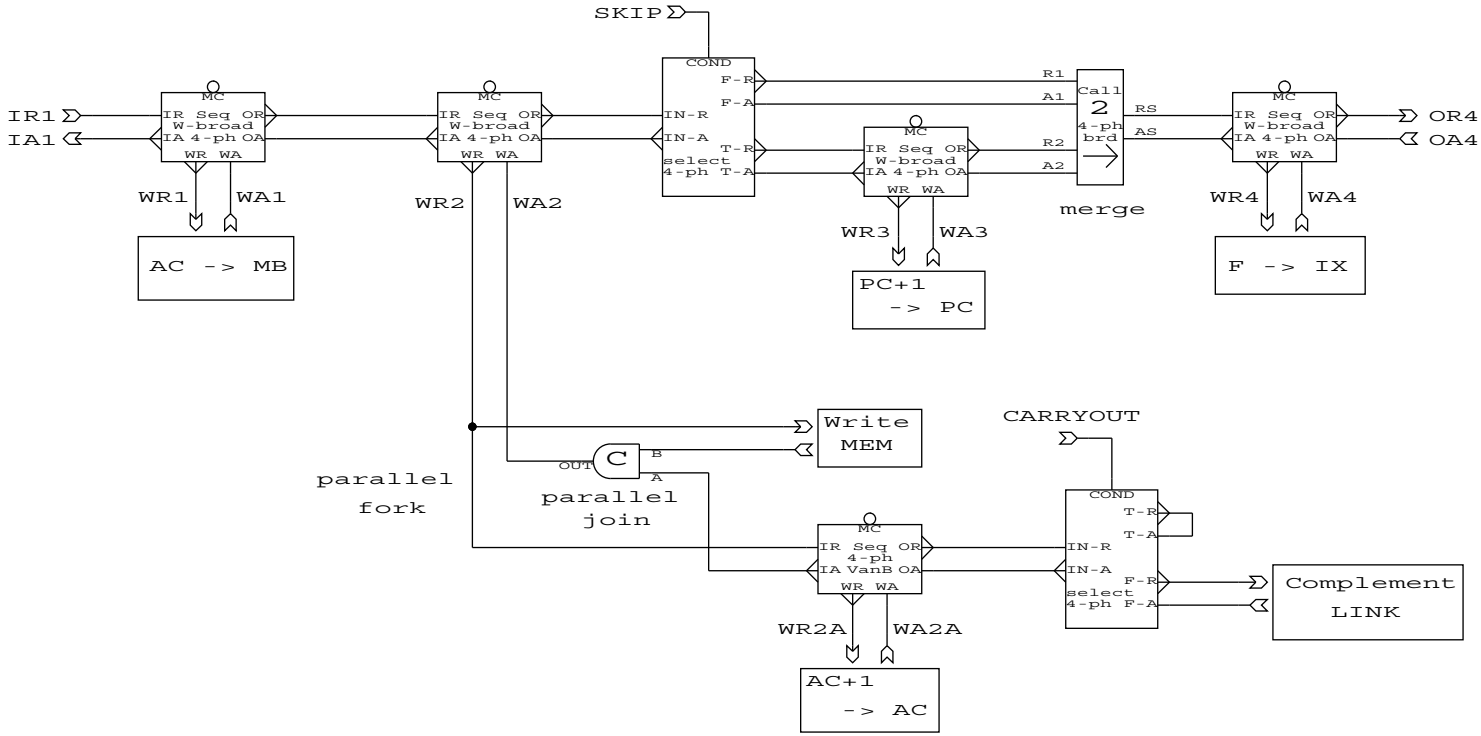


Fig. 6. Typical 4-phase control flow.

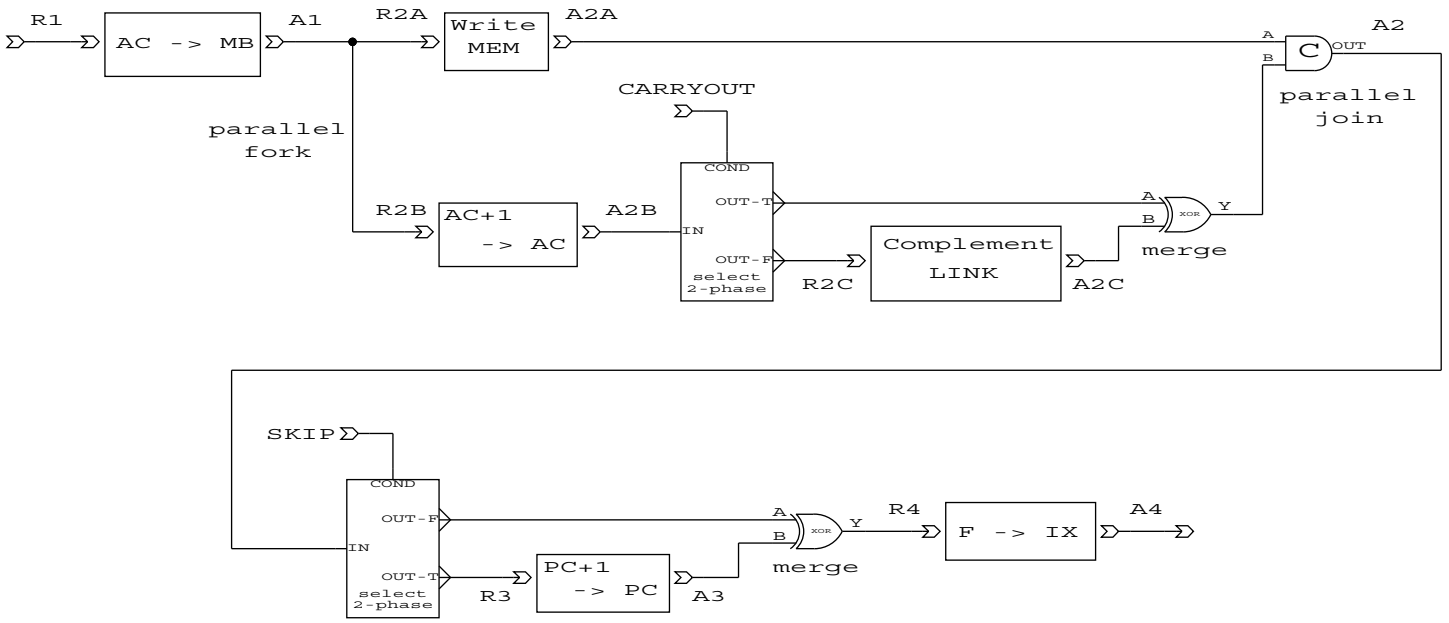
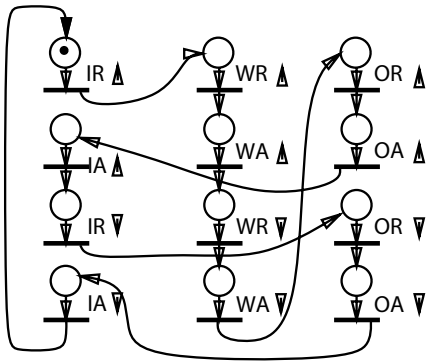
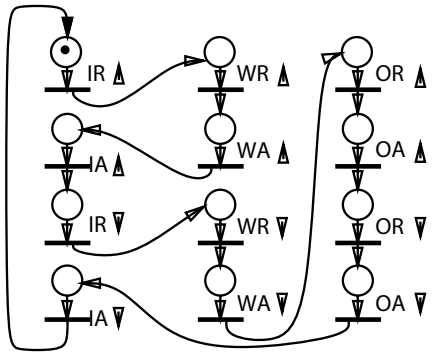


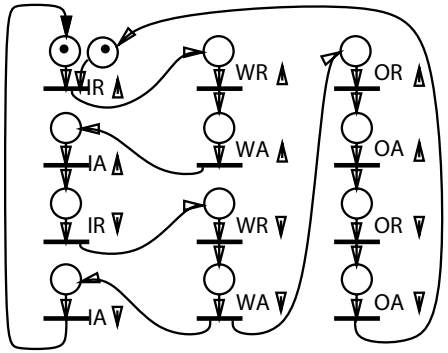
Fig. 7. Typical 2-phase control flow.



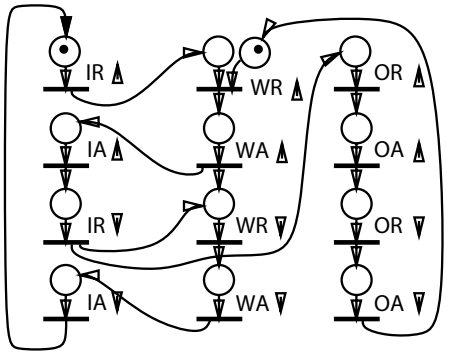
(a) van Berkel sequencer



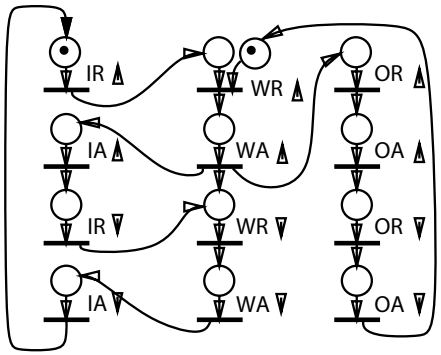
(b) Martin D-element sequencer



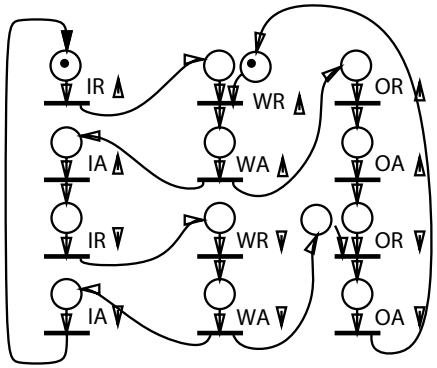
(c) Winkel broad sequencer



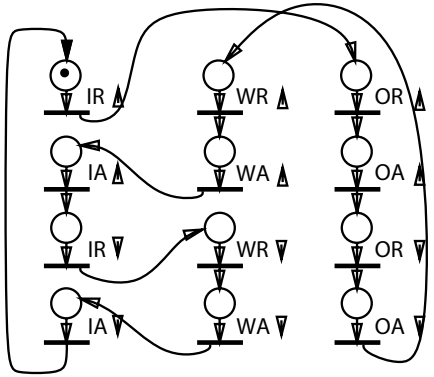
(d) Winkel weak-broad sequencer



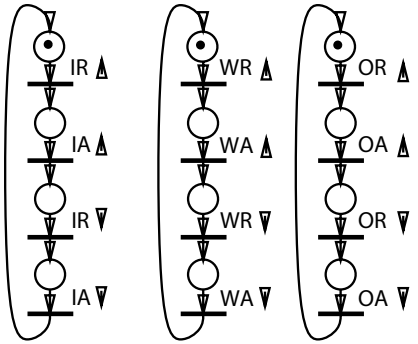
(e) Winkel narrow sequencer



(f) Brunvand narrow sequencer



(g) Martin Q-element sequencer



(h) Three 4-phase interfaces

Fig. 8. Petri nets describing 4-phase sequencer behavior.