Synchronous handshake circuits

Synchronous Handshake Circuits

Ad Peeters and Kees van Berkel
Philips Research Laboratories
Prof. Holstlaan 4 (WL01)
5656 AA Eindhoven, The Netherlands
Ad.Peeters@philips.com

Abstract

We present the synchronous implementation of handshake circuits as an extra feature in the otherwise asynchronous design flow based on Tangram. This synchronous option can be used in the mapping onto FPGAs or as a fall-back option to provide a circuit that is easier to test and integrate in a synchronous environment.

When single-rail and synchronous realizations of the same handshake circuit are compared, the synchronous versions typically require fewer state-holding elements, occupy less area, have similar performance, but consume significantly more power (in the examples studied up to a factor four).

Synchronous handshake circuits provide a means to study clock-gating techniques based on the synthesis starting from a behavioral-level specification. In addition, the study provides hints as to where the asynchronous handshake circuits may be optimized further.

1 Introduction

The Tangram approach to designing complex asynchronous circuits has established a mature design flow [4, 1, 13]. The key idea in this flow is to hide the asynchronous circuit details for the designer who uses the flow by choosing handshake circuits as intermediate architecture, cf. Fig. 1. Designs are expressed in a high-level programming language called Tangram, and the compilation to netlists is partitioned in a so-called *frontend* step (the compilation from Tangram to handshake circuits) and a *backend*. The backend is further partitioned in a component expansion step, in which the actual asynchronous circuits are generated, and a technology mapping step that depends on the standard-cell library being used.

The Tangram design flow has been used to produce many asynchronous demonstrator ICs, such as for the DCC system [3], an 80C51 microcontroller [6], and (based on the 80C51) ICs for pager, smart card, and wired telephony applications [8, 9].

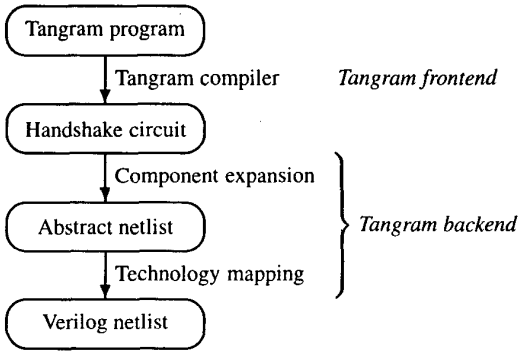


Figure 1. Tangram design flow

This paper presents an alternative backend for Tangram, based on synchronous implementation of handshake circuits. Although this may seem a weird idea at first glance, since operation dictated by a clock typically conflicts with the low-power and low-EMI targets set for handshake designs, a synchronous backend provides interesting opportunities.

One opportunity is the mapping onto various families of FPGAs. For a synchronous design, this is rather straightforward, whereas (from our own experience) for asynchronous designs this is something that for each FPGA architecture has to be handled with new care. For example, the implementation of isochronic forks and of delay matching critically depends on the interconnect architecture of the FPGA at hand.

A second opportunity is that with a synchronous backend a fall-back scenario can be offered to Tangram designers. When a design runs into confidence or test problems, a synchronous implementation may provide an escape without throwing away the investment that has been made in writing Tangram code.

We start the paper with the introduction of a class of

1522-8681/01 $10.00 © 2001 IEEE

handshake protocols that we call *sample based*, of which the synchronous protocol is a representative. Three alternative implementations of this protocol are studied. The first one is truly synchronous, in the sense that each memory element is connected directly to the clock. A second variant then is to apply clock gating in the datapath, a third option to apply this also in the control. Finally, the synchronous (based on clock gating in the datapath only) and single-rail implementations of two larger Tangram designs are compared, and a comparison with related work is presented.

2 Sample-based handshake protocols

Components in a handshake circuit communicate by means of handshake signaling. Various ways to implement handshaking have been identified and developed throughout the years, most of these using a separate request and acknowledge wire, on which events strictly alternate and are interpreted in either a four-phase or a two-phase manner [5, 16, 17]. More exotic implementations of handshaking have also been also been proposed, such as single-track [2] (in which request and acknowledge events are mapped onto the same wire) and pulse-mode [7, 14, 15] (in which separate request and acknowledge pulses are used, which are generally non-overlapping).

What all these implementations of the handshake protocol have in common is that they are defined directly in terms of events on the handshake wires that implement the protocol. The protocols also typically put restrictions on the events that are allowed on these handshake wires. In a four-phase protocol on two wires, for instance, one typically selects the state with both request and acknowledge wire low as the initial state, and from that state only allows the sequence in which first the request goes high, then the acknowledge, then the request goes low, after which the acknowledge going low returns the channel to the initial (idle) state. From this idle state, spurious pulses on the acknowledge wire are typically not allowed.

A different class of implementations is obtained if we define the implementations (that is, the definition of request and acknowledge events) in relation to a non-handshake event, such as a clock. This paper proposes a protocol in which the state of the handshake wires is observed at the rising edges of the clock, and from these states the handshakes are (re)constructed. An advantage of such a protocol is that the behavior of the handshake signals between observation events need not be restricted. In such a *sample-based* handshake protocol, glitches and hazards at request and acknowledge wires can be allowed, thus simplifying technology mapping and the exploitation in this of mainstream CAD tools.

The definition of a sample-based handshake protocol requires three issues to be handled.

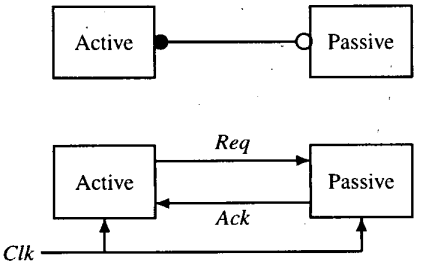


Figure 2. Handshake channel: abstract symbol (top) and implementation (bottom)

1. The *sample event*, that is, at which moments the status of the handshake wires is to be interpreted.
2. The *sequences of events* that are allowed on the handshake wires at the moments of sampling, including the initial state.
3. The allowed behavior of the handshake wires *between* observations (glitches).

In addition to these three issues, data-valid schemes including set-up and hold times around the sampling event, have to be taken into account. For the definition of the handshake protocol proper, however, this is not relevant. The discussion of these issues is postponed until the implementation of handshake components in subsequent sections later in this paper.

3 A synchronous protocol

In the synchronous protocol reported in this paper, we have chosen the rising edge of the clock as the sample event. We consider the situation shown in Fig. 2, in which the active and passive partner share the same clock signal (*Clk*), and have separate request (*Req*) and acknowledge (*Ack*) wires. (Throughout the paper, in handshake circuit diagrams, fat dots denote active handshake ports, open circles represent passive handshake ports.)

At the rising edge of the clock, the handshake signals may be in any of the following four states.

- $\neg Req \wedge \neg Ack$, the *idle* state, which also is the initial state.
- $Req \wedge \neg Ack$, the *pending* state. A request has been issued by the active side of the channel, and the passive side has not completed its action during the immediately preceding clock period.
- $Req \wedge Ack$, the *handshake* state. If both the request and the acknowledge are high, we consider the handshake on the channel completed.

- $\neg Req \wedge Ack$, the *forbidden* state. Although in principle one could allow such 'spurious acknowledgements', it turns out that the implementation of some components is actually simpler if this state is avoided. In this paper we consider such so-called *quick return-to-zero* protocols only. With this choice, the dependency between request low and acknowledge low must always be implemented such that no clock ticks are required.

The allowed sequences of events upon the observation edge of the clock (the sample event) are shown in Fig. 3. The initial state is 'Idle'. At each sample event, a transition is made to the state that is then observed. Between the three allowed states, all transitions are possible, except withdrawing a non-acknowledged request. The transition from 'Pending' to 'Idle' thus is not allowed.

The transitions in Fig. 3 are labeled with the handshake events (*Req* and *Ack*) that are interpreted. One may observe that it is possible to generate a complete handshake during each clock-cycle by remaining in the 'Handshake' state.

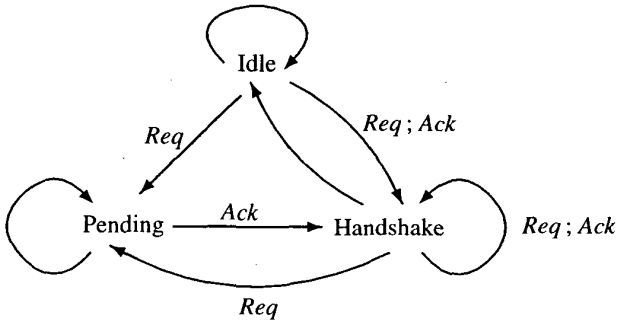


Figure 3. Allowed sequences of observation at clock edges

The transitions on the request and acknowledge wires *between* sample events are not restricted, apart from the fact that set-up and hold times with respect to both the registers used in the control and those in the datapath need be obeyed. Phrased differently, this sets an upper limit to the speed of the clock to which the observation events are related.

The choices made in the synchronous protocol are discussed with reference to Fig. 4. This timing diagram shows eight clock periods, including eight rising edges, which have been numbered I through VIII.

State I is an idle state, in which no activity is observed on the handshake channel. In state II the request wire has gone high but no acknowledge has been given yet, a condition that continues to hold in state III. This already highlights one choice that we have made, namely that requests *must* remain high until the corresponding acknowledge has been observed high, which is in contrast to choices made in [12] and [10, 11]. The pros and cons of these alternatives are discussed later.

In state IV the request is acknowledged and thus the handshake completed. One could say that it took three clock ticks for the handshake to complete. Between state IV and V several transitions on both *Req* and *Ack* are shown, illustrating one of the advantages of this sample-based protocol, namely the insensitivity to such glitches and hazards. State V illustrates a handshake that takes only one clock tick to complete. The transition to state VI shows that a handshake can be completed each clock cycle without making transitions, which is optimal for power, as far as the handshake signaling is concerned.

A forbidden state is state VII: when there is no request, the acknowledge must be low. This choice leads to quick return-to-zero logic that simplifies the design of some handshake components using this protocol. The transition to handshake state VIII again shows correct behavior: it is allowed for the acknowledge to glitch before settling, although set-up times should of course be obeyed.

In the next sections we discuss several implementation issues regarding the synchronous protocol that is proposed above. We start in Section 4 with an implementation in which every register is connected *directly* to the clock. In Section 5 we then introduce clock-gating, which adds at most an AND gate between the global clock and the local gated clock that in itself directly connects to the registers involved. A subsequent extension would be to distribute the clock along with the handshake signals.

4 Synchronous handshake components

This section presents synchronous implementations of some relevant handshake components for control (repeater, mixer, and sequencer) and data (variables).

Repeater

A repeater with passive port a and active port b executes handshakes along b forever, once activated along a (the request along a thus is never acknowledged). It implements Tangram's infinite-looping construct. Its specification in terms of request-acknowledge events is $a_r; (b_r; b_a)^*$. Its symbol is shown in Fig. 5(a).

The four-phase implementation of the repeater is shown in Fig. 5(b). Once a_r is made high, effectively a transition is injected in the loop formed by b_r and b_a in conjunction with the passive side of channel b. The synchronous implementation, shown in Fig. 5(c), is actually simpler. Once activated it will always keep signal b_r high. No logic is required in this element due to the sampling nature of the protocol.

Both from a structural and from a power perspective, this circuit has advantages over the four-phase implementation. The circuit does not introduce a combinational cycle (b_a is

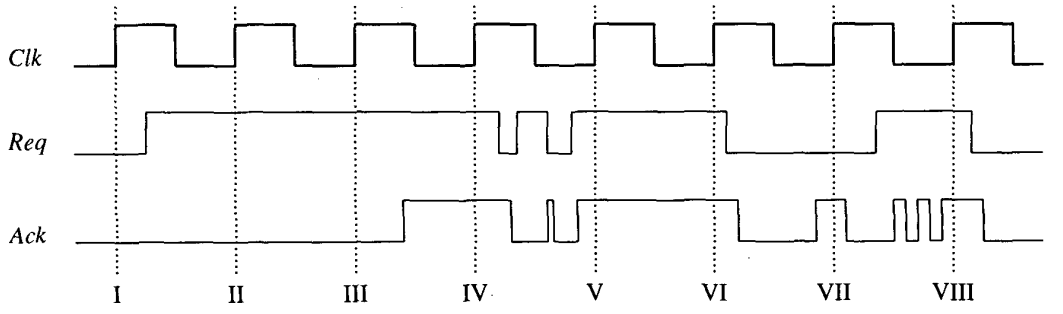


Figure 4. Some allowed and non-allowed synchronous handshake events

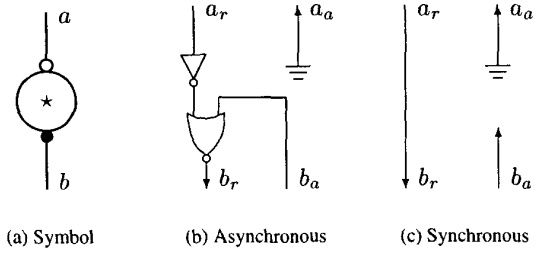


Figure 5. Repeater symbol, four-phase, and synchronous implementation

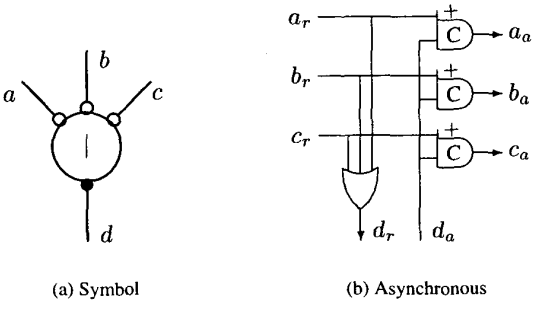


Figure 6. Three-party mixer symbol and four-phase implementation

simply ignored), and no transitions on b_r are needed between consecutive handshakes on b to keep the circuit going.

Mixer

The mixer is used to implement (mutual exclusive) sharing in Tangram. In the variant where the handshake channels are data channels rather than control channels, it is a multiplexer, of which the control circuit is based on the control mixer presented here. The symbol for a three-party control mixer is shown in Fig. 6(a). Handshakes on a, b, and c must be mutually exclusive, and in that condition, the component forwards the handshakes to channel d.

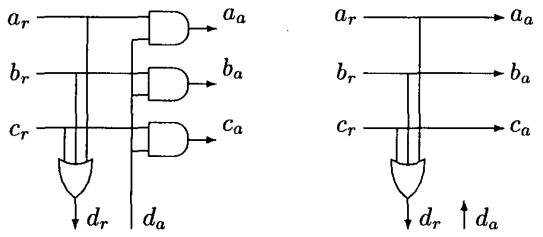
The four-phase implementation of the mixer is well-known, and shown in Fig. 6(b). It uses an asymmetric C-element for each passive port, the state-holding function of which is only required during the return-to-zero phase, when the request has already gone low, but the acknowledge has to remain high until the acknowledge of the shared channel (d) has gone low.

In the synchronous implementation, this sequential functionality is not required since we have chosen for a quick return-to-zero protocol. Consider a 'Handshake' state that e.g. involves channel a, in which a_r, a_a, d_r, and d_a are high. If before the next sample event a_r goes low, then d_r must also go low, and as a consequence of that also a_a and d_a, respectively. In an asynchronous implementation, these transitions are sequenced using the OR-gate and the C-elements. In the synchronous realization these return-to-zero phases may go in parallel. The C-elements can thus be replaced by AND gates, as shown in Fig. 7(a), which makes the implementation completely combinational, and saves sequential elements compared to the four-phase implementation.

We can optimize the mixer when it is known that the handshake on d will always complete within one cycle, that is, when the path from d_r to d_a is combinational. Since it is then known that d_a will go high in the same cycle as the request on the initiating passive port, we can ignore this acknowledge and omit the AND-gates, as shown in Fig. 7(b).

Sequencer

The sequencer handshake component implements the sequencing of events in Tangram. In general, it is a component with one passive (activation) port and several active



(a) General mixer (b) One-tick mixer

Figure 7. Synchronous mixer circuits

ports on which it guarantees proper sequencing of handshake events. The symbol and four-phase asynchronous implementation of the binary sequencer is shown in Fig. 8. The behavior of this (completely sequential) implementation of the sequencer is [1]:

$([a_r]; b_r\uparrow; [b_a]; x\uparrow; [x]; b_r\downarrow; [\neg b_a]; c_r\uparrow; [c_a]; a_a\uparrow;$
$[\neg a_r]; x\downarrow; [\neg x]; c_r\downarrow; [\neg c_a]; a_a\downarrow)^*$

This implementation is self-initializing: making passive input a_r low suffices to force state variable x in its desired initial state (being low).

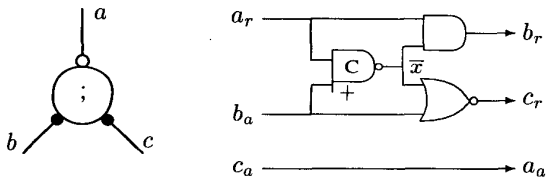


Figure 8. Two-party sequencer symbol and four-phase implementation

The synchronous implementation of the sequencer can be derived starting from the sequential specification as given above. Only this time, the updates of the state variable x take place upon rising edges of the clock, and the state is stored in a flip-flop rather than a latch. Furthermore, the synchronous protocol allows for more parallelism, particularly between the return-to-zero phase on channel b and the handshake on channel c. In addition, we strive for an implementation in which there is no clocking overhead, so that each branch can (in principle) complete within one clock period, and the total cycle time of the sequencer can be two clock periods.

This results in the following behavior, including the timing of clock events:

$(Clk^*; [a_r]; b_r\uparrow; Clk^*; [b_a]; x\uparrow; Clk;$
$([x]; b_r\downarrow; [\neg b_a])||(c_r\uparrow; Clk^*; [c_a]; a_a\uparrow; [\neg a_r]); x\downarrow; Clk;$
$[\neg x]; c_r\downarrow; [\neg c_a]; a_a\downarrow)^*$

This specification is used to define the next-state function of x and the expressions for the outputs, which leads to the circuit shown in Fig. 9.

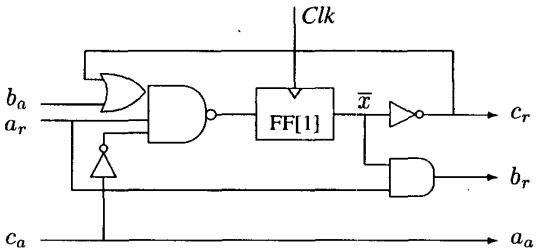


Figure 9. Synchronous two-party sequencer

When it is known that the b-branch of the sequencer completes within one clock cycle (e.g., when it connects to an assignment or a skip statement), the circuit can be simplified, as shown in Fig. 10.

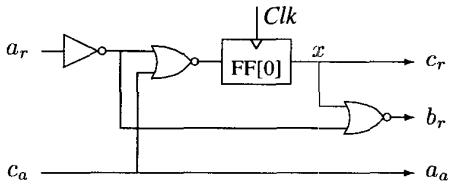


Figure 10. One-tick synchronous sequencer

The circuits shown require the (co)operation of the clock to initialize. Naturally, a hard (re)set could be added to the flip-flops, but otherwise, when input a_r is low, one clock-tick will suffice to force the flip-flop into its desired initial state ($\overline{x} = 1$ in Fig. 9, $x = 0$ in Fig. 10).

Multi-party sequencers could be based on cascading binary sequencers, in which case the initialization time might be linear in the number of branches. A more dense encoding of the sequencer state could also be chosen, thus reducing the number of flip-flops and hence reducing the power consumed by the clock signal.

Variables

So-called write components are used to implement variables, and can be either flip-flop or latch type (this can be specified in Tangram, upon declaration). Fig. 11 shows the single-rail implementation of such a write component.

The latches in the latch-based implementation are transparent when their 'clock' input is high and opaque otherwise. The data is thus latched in such a variable upon the

falling edge of the write-request signal, in line with the data-valid scheme that applies in the handshake circuit [13]. The write-acknowledge signal is connected to the request signal directly, assuming that the environment provides sufficient delay between the closing of the latch and its output assuming its latched value. This assumption can be weakened by delaying the write-acknowledge signal.

The registers in the flip-flop based implementation are positive-edge triggered. To align this with the data-valid scheme, which ends at the falling edge of the write acknowledge signal, we have to invert the clock input of the flip-flop.

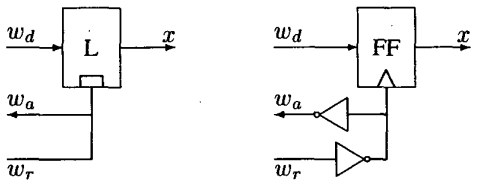


Figure 11. Four-phase single-rail implementation for latch (left) and flip-flop variable (right)

In addressing the synchronous implementation, we first focus on the implementation of this component for flip-flops. We have chosen the rising edge of the clock as the sampling edge for the request and acknowledge signals, so it appears natural to also choose this edge as the sampling edge for signals in the datapath. For the implementation of a flip-flop variable, we can then choose the configuration shown in Fig. 12, in which the data being latched at the positive clock edge is either the old value (when the write request is low), or the write data (when the write request is high). The write-request signal should be stable long enough prior to the clock edge to guarantee both the set-up time of the register and the forward time of the multiplexer.

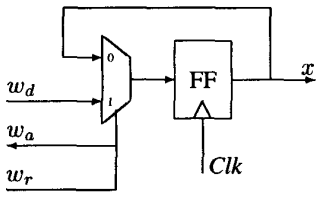


Figure 12. Synchronous flip-flop variable

The price of having a multiplexer cell per register is high. A more attractive solution is obtained through clock-gating, as addressed in Section 5.

The implementation of the latch-based variable is slightly more complicated, at least as far as the analysis is concerned. The implementation is shown in Fig. 13. One may observe the surprising structure of a latch with a multiplexer that is used in feedback mode only when the value has to remain stable. Naturally, such a structure could not be used to invert the state of the latch since that would give rise to oscillations during the period in which the latch is transparent.

The latch structure is sensitive to changes on data-input w_d and write request w_r while it is transparent, that is, while the clock is low. During this period, signal w_r must be stable, especially when the latch should maintain its state (if w_r is low at the rising edge of the clock). If the data in the latch has to be overwritten by inout data w_d (if w_r is high at the rising edge of the clock), then glitches on w_r can be tolerated.

The extra restriction on the synchronous protocol has had no impact in practice for two reasons. Firstly, that gate-level implementation of the control elements is such that glitches on control signals only occur when that signal is supposed to go high, not when it remains low. Secondly, the combinational depth of the control logic is low and typically far less than half of the maximum combinational depth in the datapath. The tuning of the clock thus typically is such that all control signals stabilize within half a clock period, that is, before the falling clock edge.

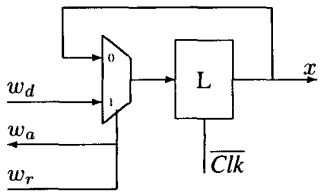


Figure 13. Synchronous latch-based variable

5 Clock gating

In the synchronous implementations presented so far, all registers (whether in the control or in the datapath) are directly connected to the clock. Naturally, this may lead to excessive triggering of these elements. In combination with the handshake signals that are available it is straightforward to suppress redundant triggers by gating off the clock signal.

The most straightforward application of clock-gating is in the implementation of variables in the datapath. The write request signal in such a variable encodes whether or not an update of the registers is required, and can thus directly be used for clock-gating, as shown in Fig. 14. The savings are also apparent: the multiplexer per bit has been replaced by a single AND gate.

In the flip-flop variant, the write-request signal must be high long enough to generate a clock pulse of sufficient width at the clock input of the registers. For the latch-based variant, the requirements for the write-request signal are even more strict. During the phase in which the clock

is low, the latch-enable signal directly follows the write-request. This implies that transients on this signal can only be tolerated when the latch must be written at the end of that period, just as for the ungated variant of Fig. 13 discussed earlier.

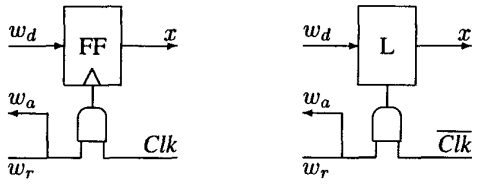


Figure 14. Clock-gated flip-flop and latch-based variables

Clock gating in the datapath is a clear win: many multiplexers can be saved this way (thus establishing an area reduction with respect to the pure synchronous variant), and power is reduced, since each variable essentially constitutes a clock domain of multiple registers that is switched off when inactive.

The advantage of introducing clock-gating in the control is less apparent. By the definition of the synchronous protocol, we can in each component with only one passive port (such as the sequencer, parallel, do, if, and case components) gate off the clock using a two-input AND gate. One input is then connected to the request signal of that passive port, the other input to the global clock, and the output to the local clock that is thus established. This investment only pays off if the local clock domain is sufficiently large, as e.g. in a multi-party sequencer or parallel component.

The gating in the control comes at a price, however, since it makes initialization of registers in the control through initial clock ticks while keeping the request low impossible. Hard sets or resets are then required, e.g. for the sequencer circuits in Fig. 9 and Fig. 10. This adds a global signal to the circuit, possibly complicating layout.

The skew introduced between the local clocks and the global clock in the clock-gating described above is limited to a single AND gate, which makes it well manageable with respect to timing closure.

6 Design flow

The design flow from Tangram to standard-cell netlists for the synchronous backend is based on the single-rail backend, as described in detail in [13]. The compilation from Tangram to handshake circuits is independent of the backend applied later.

The step from handshake circuits to synchronous gate-level implementations is started with a traversal of the handshake circuit to determine for as many handshake channels as possible how many clock ticks will be required for each handshake on that channel to complete. For many channels, this can be determined (such as around assignments and sequences thereof), for some it cannot (such as for communications and while-loop constructs).

In the gate-level expansion step, the implementation of the control components is tuned to the number of required clock ticks per channel as determined earlier. The sequencer and mixer, for instance, are simplified with respect to the general case if a handshake on its active port is known to complete within one cycle, as shown in Section 4. Another example is the parallel component, which can be implemented using an AND gate if all its active ports require exactly the same number of clock cycles to complete, but requires registers otherwise.

In this paper we have only implemented and evaluated the variant of the synchronous protocol in which the control is truly synchronous and clock-gating is applied to the registers in the datapath.

Although the same standard-cell library is used for the technology mapping of both the single-rail and the synchronous backend, there is a slight difference in the way this mapping is performed. For the synchronous version, *all* combinational logic is optimized by a logic optimizer, whereas in the asynchronous version, only the combinational logic in the datapath is optimized, thus preventing the introduction of glitches in the handshake control.

In the single-rail backend, the timing management consists of characterizing the worst-case path through each block of datapath logic, and tuning the delay path that matches it with the appropriate margins taken into account.

In the synchronous backend, in contrast, one figure must be determined that is an upper bound for *all* worst-case timing paths, whether in the logic, or in the control. This figure then sets an upper bound for the maximum frequency at which the circuit can be operated.

With respect to the worst-case timing path through the logic, one should observe that the deepest combinational expression determines the timing for all paths. For instance, in Tangram program

```
x:=x-y ; y:=0,
```

the first assignment will determine the timing of the synchronous realization, whereas in the single-rail realization, the second assignment can complete a lot quicker than the first one involving the subtraction. If the worst-case path through the datapath logic is excessive, one may reduce that, e.g. by introducing an extra clock-tick for assignments involving this longest path, thus effectively halving that path.

The control circuit in the synchronous backend has been designed such that it will not constitute the worst-case path

in a typical design. Care has to be taken not to put too many logic gates in series. For instance, in the design of a $(N+1)$-party sequencer based on the binary sequencer presented in Fig. 9, a complete left-branching tree will see N AND-gated in series, a problem that the right-branching variant has not.

7 Case studies

We compare the synchronous and single-rail implementations of Tangram designs: a DCC error decoder and the CPU of an 80c51 microcontroller. In the comparisons, the Tangram programs and handshake circuits that serve as a starting point are the same for synchronous and single-rail implementation. The difference is only in the implementation of the control part of the handshake components. The synchronous variant that is used is the one where clock-gating is applied only to the datapath registers. The numbers for performance and power are obtained from simulations of netlist.

DCC error detector

The error detector for the DCC system is discussed in detail in [3]. A detailed evaluation of its single-rail design can be found in [13].

Table 1 presents the important design characteristics of the synchronous and single-rail version, compiled from the same Tangram program (and handshake circuit). The total gate area is not much different for the two variants: the synchronous version turns out to be some 2% smaller than the single-rail version.

The number of state-holding elements in the control has reduced drastically. Of the 467 C-elements, only 227 occur as a flip-flop in the synchronous controller. The C-elements that have disappeared are those that were part of sharing constructs (mixers, multiplexers, and demultiplexers). The reason that this roughly halving of the number of state-holding elements did not bring any area improvement is that a typical C-element (2-input, asymmetric) is mapped onto a cell occupying less than two gate equivalents, whereas the flip-flop in the library occupies four grids.

The power figures are a convincing win for the single-rail circuit. The synchronous design has maximum clock-gating in the datapath, but the price for this is high: the 227 flip-flops in the control, that together constitute the state machine for clock gating, provide a high clock load. In that sense the clock-gating is really extreme: for the gating of 402 latches, 227 flip-flops are invested. In that light, the 467 C-elements in the control is an even higher investment for low-power. The advantage of C-elements over flip-flops, however, is that their operation is autonomous, and that power is only consumed at state-transitions, not at the pace that is required to drive the whole circuit, as in the synchronous version.

Characteristic	[unit]	s.r.	sync
Area	geq	4564	4482
Registers in datapath	#	402	402
C-elements in control	#	467	
Flip-flops in control	#		227
Energy	nJ	155	537
Cycle time	μsec	22.7	19.5

Table 1. Comparison of single-rail and synchronous DCC decoder

Actually, the power figures are optimistic for the synchronous version. In this application, the run-time of the algorithm depends on the number of errors in the code word that is begin processed. In the synchronous version this means that the number of clock-ticks required to complete the computation is data dependent. It is assumed that while the new sample is not yet available, the clock to the whole circuit is gated off, which would be easy to implement in this Tangram design.

Interestingly, the power consumption of the synchronous handshake version is comparable to that of the (clock-gated) synchronous implementation of the same function that originally set the target for our single-rail work [3, 13].

As for the performance of the circuit, in terms of the cycle time required for a typical test sequence, the synchronous circuit appears to be slightly faster. This is likely due to the fact that the synchronous implementation uses a quick return-to-zero scheme. The DCC decoder does not contain deep combinational circuitry.

80c51 microcontroller

The second example is taken from the 80c51 microcontroller reported in [6], and later used in e.g. the smart-card controller reported in [8]. We have compiled a variant of its CPU that was optimized for lower power consumption. Table 2 presents the area and power characteristics of the single-rail and of the synchronous compilation of this of the CPU. The exact synchronous and single-rail performance has not been determined yet, although analysis indicates that the performance will not differ by more than 10%.

Most notable in Table 2 is the reduction in area of the synchronous version compared to the single-rail version. The main reason for this is the high number of sharing structures (mixers and multiplexers) in the design. This is responsible for the reduction of the number of C-elements when going to flip-flops (only 197 flip-flops for the original 572 C-elements), and for the optimization of the combina-

Characteristic	[unit]	s.r.	sync
Area	kgeq	5.1	4.1
Area for logic	kgeq	3.0	2.4
Registers in datapath	#	240	240
C-elements in control	#	572	
Flip-flops in control	#		197
Energy per instruction	nJ	0.39	1.7
Performance	MIPS	5.6	5.7

Table 2. Comparison of single-rail and synchronous 80C51 CPU

tional logic (the AND-gates in the mixer allow easy integration with other cells).

The performance figures for the two versions are similar, assuming it is not limited by the timing of the ROM or RAM being used. Otherwise, the asynchronous version would pay the price only when accessing these, whereas in the synchronous version the clock must be matched to these devices.

The synchronous handshake version has area and power characteristics that are comparable to that of the VHDL-synthesised synchronous circuit that was used as reference design for the asynchronous 80c51 [6, 8]. An interesting difference is that the handshake version requires only 6.6 clock ticks per instruction on average, whereas the reference design requires 10.4 clock ticks (assuming an running average of 1.7 bytes per instruction), thus illustrating the elimination of redundant actions as addressed in [6].

Discussion

The two case studies presented above are in line with our general findings: the synchronous implementations of handshake circuits tend to be smaller, show comparable performance, and consume three to four times the power of the single-rail realizations of the same handshake circuit.

Synchronous realizations typically require less circuit area since delay matching is not needed and the combinational logic in the control can be optimized. The latter is tricky for asynchronous control, where even simple bubble shuffling can introduce glitches in the handshake control.

The performance of the single-rail and synchronous circuits are similar. We have used the same margins (50%) in tuning the delay-matching paths in the single-rail datapaths and in tuning the synchronous clock. The reason that the asynchronous versions are not faster than their synchronous equivalent is threefold.

First of all, the slowest datapath in these designs is still relatively fast (the circuits have low combinational depth), such that the penalty of waiting for this in every cycle is not very high. Secondly, in the delay matching in the single-rail datapaths, we do not take the timing of the control into account. The results hint that the asynchronous circuits may gain performance here (say 10 to 20%). Thirdly, the synchronous protocol allows for parallelism (e.g. in sequencing handshakes) between the return-to-zero phase of one handshake, and the active part of the next handshake, whereas the single-rail protocol is fully sequential.

Synchronous implementation of handshake circuits has taught us that the amount of clock gating that this introduces is extremely high, due to the sparse encoding of the handshake control circuit. In an asynchronous implementation, sparse encoding helps in reducing power since it minimizes the number of transitions between state changes, whereas in a synchronous implementation, from some point onwards it actually increases power consumption because all state-holding elements in the control need to be clocked. Apparently, by taking the handshake circuit as a starting point, the state encoding in the controller is too sparse to be optimal with respect to synchronous power consumption.

8 Related work

Synchronous forms of handshaking have been studied in the field of mapping algorithms onto FPGAs. Page and Luk proposed a scheme in the context of compilation starting from Occam [12], aimed the hardware-software codesign starting from a common language. A form of handshaking based on pulse-mode emulation (as proposed by Keller in 1974 [7]) has been proposed by O'Leary and Brown [10, 11]. In both approaches, pulses on request and acknowledge wires (around clock edges) encode handshake events.

In terms of sample-based protocols introduced in this paper, the main difference is that transitions from the 'Pending' to the 'Idle' state (in Fig. 3) are allowed and transitions from 'Pending' to 'Pending' are not. This implies that storage elements are often required to trace back the path to the request source when the acknowledge comes along, most notably in the context of sharing. In general, circuits compiled using the protocol introduced in this paper will have significantly fewer sequential elements in the control and are thus smaller in area and more power-friendly.

The clock-gated circuits such as compiled in a synchronous handshake approach (whether based on the work described in this paper, or on the variants described in [12, 10, 11]) provide an interesting class of synchronous circuits. The clock domains in the datapath are very small, thus facilitating clock distribution and layout. The largest clock domain will typically be that of the complete handshake control, which still operates as one synchronous state machine.

In synchronous circuit design, clock-gating is typically applied either at a coarse-grain block level or at the cell-

level by trying to reconstruct hold conditions from the netlist [18]. The clock-gating presented here, in which a behavioral description is implemented directly using a distributed clock-gated controller results in extremely gated circuits, most likely too extreme to be optimal, at least for power.

9 Conclusion

We have presented a synchronous backend for the Tangram silicon compiler. A strong point of this alternative to single-rail implementation is that is starts from the same handshake circuit representation, so that the compilation from Tangram remains the same. The circuits can be used straightforwardly for prototyping on FPGAs, due to their synchronous nature. The mapping of asynchronous implementations on FPGAs, in contrast, is always very delicate and depends on the particular FPGA architecture at hand.

The synchronous backend is also interesting as a fallback scenario. Testing for production faults using scan test, for instance, can be done using commercially available DfT/ATPG tools by replacing the flip-flops by their scan variants. Furthermore, the availability of a synchronous alternative reduces the risk of investing in the Tangram design-flow.

In this paper, we have assumed the circuits to be synchronous with respect to a single clock. With the synchronous protocol it is also be possible to connect two handshake circuits operating on different clocks, although this naturally comes at the price of occasional metastability problems. This interfacing boils down to designing a two-clock passivator that implements the synchronization between two (active) handshake ports. The design of this component falls outside the scope of this paper.

With the addition of this hybrid class of circuits to the synchronous-asynchronous landscape, the number of design alternatives has again increased. Making a fair comparison between asynchronous and synchronous implementations of designs appears an ever-increasingly hard job.

Acknowledgement

We thank Cees Niessen for challenging us to explore synchronous implementation of handshake circuits. Ken Stevens is gratefully acknowledged for his many helpful comments and suggestions.

References

[1] K. v. Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[2] K. v. Berkel and A. Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–133. IEEE Computer Society Press, Mar. 1996.

[3] K. v. Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. A fully-asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid-State Circuits*, 29(12):1429–1439, Dec. 1994.

[4] K. v. Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation*, pages 384–389, 1991.

[5] W. A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336, Atlantic City, NJ, 1967. Academic Press.

[6] H. v. Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters, and G. Stegmann. An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, 1998.

[7] R. M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, Jan. 1974.

[8] J. Kessels, T. Kramer, G. den Besten, A. Peeters, and V. Timm. Applying asynchronous circuits in contactless smart cards. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 36–44. IEEE Computer Society Press, Apr. 2000.

[9] J. Kessels and A. Peeters. The Tangram framework: Asynchronous circuits for low power. In *Proc. of Asia and South Pacific Design Automation Conference*, Feb. 2001.

[10] J. O'Leary. *A model and proof technique for verifying hardware compilers for communicating processes*. PhD thesis, Cornell University, 1995.

[11] J. O'Leary and G. Brown. Synchronous emulation of asynchronous circuits. *IEEE Transactions on Computer-Aided Design*, 16(2):205–209, Feb. 1997.

[12] I. Page and W. Luk. Compiling Occam into FPGAs. In W. Moore and W. Luk, eds, *FPGAs*, pages 271–283, 1991.

[13] A. M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996. http://www.win.tue.nl/~wsinap/pdf/Peeters96.pdf.

[14] L. A. Plana. *Contributions to the Design of Asynchronous Macromodular Systems*. PhD thesis, Department of Computer Science, Columbia University, 1998.

[15] L. A. Plana and S. H. Unger. Pulse-mode macromodular systems. In *Proc. International Conf. Computer Design (ICCD)*, pages 348–353, Oct. 1998.

[16] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[17] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[18] Q. Wu, M. Pedram, and X. Wu. Clock-gating and its application to low power design of sequential circuits. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(103):415–420, Mar. 2000.