Design and FPGA-implementation of Asynchronous Circuits Using Two-phase Handshaking

Adrian Mardari, Zuzana Jelčicová and Jens Sparsø
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Email: s172997@student.dtu.dk, s173084@student.dtu.dk, jspa@dtu.dk

***Abstract*—This paper addresses the design and FPGA-prototyping of asynchronous circuits using static data-flow handshake components implemented using the two-phase bundled-data protocol. The contributions are partly tutorial and partly scientific. The paper introduces the design process, including initialization and design of coupled rings with any number of tokens. Following this, the paper presents gate-level implementations of the full set of handshake components as well as some peephole optimizations that merge the implementation of several components. The components are implemented using the click-template. The handshake register implementation is *extended* with circuitry that decouples the phase of the handshake signals on the input and output ports. Such decoupling is needed to facilitate implementation of rings with one token (or in the general case, rings with any number of tokens). Finally, the paper illustrates the design process using two circuits: one that outputs the sequence of Fibonacci numbers, and one that computes the greatest common divisor of two positive integers. All components are described in VHDL, and all code is available as open source. All components and the two circuits mentioned have been tested on a Xilinx Nexys4DDR FPGA board.**

I. Introduction

When engineering students learn digital electronics, they typically do lab exercises where they design (small) synchronous sequential circuits and implement these in FPGA technology. A similar situation does not exist for asynchronous design.

Despite decades of research, there are no widely used tools, and the situation has not improved during the last decade. CAD tools are typically developed by and used within individual university groups and companies. Many of these groups have used variants of CSP [10] to describe asynchronous circuits and systems. Some examples are [2], [16], [25]. The last of these was later commercialized by the start-up company Handshake Solutions, and at one point, their Haste language and synthesis tools were available to universities through Europractice [5]. Our experience at that time was that students ended up writing concurrent programs with very limited understanding of what hardware their programs would generate – a paradox in light of the full transparency of syntax directed compilation.

For a newcomer, and in a teaching context, we believe that less-is-more, and for that reason we aim for a simple and straightforward component based-approach. Our aim is to provide students with FPGA-implementations (i.e., synthesizable VHDL descriptions) of the handshake components presented in [23, Ch. 3]. From this, they can then build static data flow structures by simply wiring together the relevant components, they can simulate the circuits, and they can implement and operate their circuits using a conventional FPGA board. For this purpose, the click-element template [18] using only D-flip flops and combinational gates, seems to be a good fit.

The contributions of the paper are partly tutorial and partly scientific. The material and insights presented emerged from a course on asynchronous design, where students were asked to design and build small asynchronous circuits. This turned out to be surprisingly difficult. The reason is that when going beyond simple pipelines, many important details such as initialization, numbers of tokens in rings, implementation of components etc. are not well covered in the literature. The aim of our paper is to fill this void, and to enable newcomers to experiment with, and get hands-on experience with, the design and implementation of small asynchronous circuits, in order to support the learning process.

The paper makes three contributions: (1) We discuss and decide on a set of design guidelines, including how to implement two-phase rings with any number of tokens (often just a single token). (2) We present the design and FPGA-implementation of the set of handshake components from [23, Ch. 3]. The handshake registers are what we call "phase-decoupled" (based on ideas first proposed in [19]). The rest of the components are transparent to the handshaking, in contrast to what is used in most other works based on the click-template. (3) We illustrate the use of the design guidelines and the component library using two small circuits: one that emits the sequence of Fibonacci numbers and one that computes the greatest common divisor of two unsigned numbers. All code and all examples are available as open source.

The paper is structured as follows: Section II presents background and related work. Section III discusses design challenges and presents a set of design guidelines or policies. Section IV presents the design and FPGA-implementation of the set of handshake components. Section V shows component optimizations that fuse several components. Section VI presents the two example data-flow structure circuits, and finally Section VII concludes the paper.

II. Background and related work

A. Data-flow components

Asynchronous circuits are often designed using data-flow components. The data-flow abstraction decouples high level thinking from low-level implementation details including what

978-1-5386-5883-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ASYNC.2019.00010

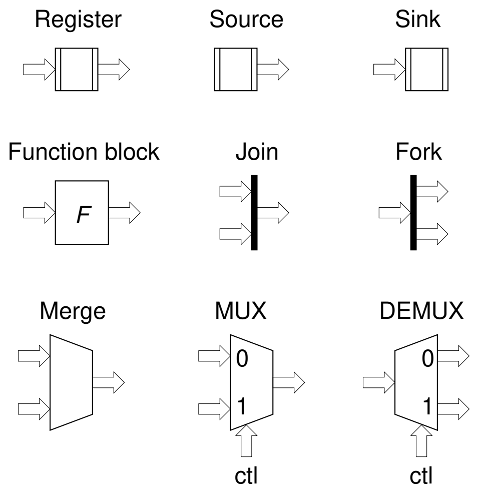

Fig. 1. The set of data-flow components from [23, Ch. 3]. The "box-arrows" represent handshake channels (a bundle of request, acknowledge and data signals). Except for the handshake register (including Source and Sink), all components are transparent to handshaking, i.e., they do not buffer data.

handshake protocol to use. The set of handshake components introduced in [23, Ch. 3] is shown in Fig. 1. Similar components are used in [3], [12], [15].

B. Two-phase bundled-data handshake components

Over the years, researchers have discussed what handshake protocol to use. Four-phase dual-rail quasi delay-insensitive handshaking has been in use since the early years [17] and is still widely used [1], [20]. However, its key feature – insensitivity to delays in gates and wires – comes at a high cost in terms of area and power.

The bundled-data design styles avoid this overhead, and Ivan Sutherlands paper “Micropipelines” [24] created a big interest in two-phase bundled-data circuits. The backbone control circuit is (still) the well-known Muller pipeline based on C-elements. Later, the Mousetrap template [21] was introduced as a faster alternative. It uses only “conventional gates” (an XOR-gate and a latch) to implement a controller that directly controls a level sensitive latch for data. However, when going beyond simple pipelines, Mousetrap still needs C-elements, for example in join and fork components.

In 2010, click elements [18] were introduced as a new template for implementing two-phase bundled-data designs. The click-template allows all handshake components to be implemented using only combinational gates and D-flip-flops, that are available in all standard-cell libraries. In addition, all signal paths start and end in edge triggered D-flip-flops. This is more in line with the view of conventional (synchronous) CAD-tools. The basic idea of the click template has since been used in several other works [4], [19].

C. Control circuits for bundled data pipeline stages

The control circuit used in all two-phase and four-phase bundled data pipeline stages can be seen as variations of a C-element based Muller pipeline, possibly implemented using other components than the C-element, as is the case with Mousetrap and Click. In all cases, the behaviour of a pipeline is that data (a token) from a predecessor stage is latched by or clocked into a stage, if the data which the stage was previously holding has been taken over by the successor stage. In two-phase pipelines, a token is captured when the C-element in the stage makes a transition. In four-phase pipelines, a valid token is captured when the output of the C-element goes to “1”, and likewise, an empty token is captured when the C-element goes to “0”.

It is well known that a ring composed of Muller pipeline stages needs at least 3 stages (C-elements) to oscillate [8], [23]. The three C-elements repeatedly cycle through the following sequence of states $(010; 011; 001; 101; 100; 110)^*$. This oscillation can be seen as a standing wave that propagates by copying the crest and trough forward, with the restriction that the circuit cannot enter states 111 and 000.

D. FPGA implementation

Several papers have explored how to use conventional FPGA-technology for building prototypes of asynchronous circuits. There are several challenges involved in this: (1) implementation of C-elements and generalizations thereof, (2) handling of isochronic forks, and (3) implementation of matched delay elements that are used in bundled data circuits. The former is addressed in [9] that shows how a LUT, whose output signal is fed back to one of its inputs, can be used to realize hazard-free (generalized) C-elements. The same paper argues that isochronic forks can be handled by setting proper timing constraints for the synthesis. The implementation of delay elements is explored in quite some detail in [14], which also gives full details of the design and FPGA implementation of an asynchronous network on chip. We have adopted these techniques for implementing our two-phase bundled-data click-style circuits. For completeness, we provide pointers to additional literature on this topic [6].

III. Design methodology

A. Introduction

There is a rich literature on the implementation of asynchronous pipeline stages (handshake latches) and their use in pipelined circuits, such as arithmetic circuits and routers for networks on chips. A few representative examples are [7], [20]. When such pipelined circuits are characterized, the typical test-setup is to use an initially empty pipeline connected to independent sources and sinks. This context does not expose a number of issues related to the design and initialization of circuits containing rings. Static data-flow structures typically involve many coupled rings and short pipeline segments, possibly shared by several nested rings. Here initialization plays a key role and when pipelines are connected to form rings, the handshaking on the two handshake channels that are connected must agree on the polarity of the signal transitions (rising or falling). Below we address these issues for rings using two-phase handshaking.

B. Rings using two-phase handshaking

The static data flow structures view of a three-stage ring using four-phase handshaking is that the three latches contain a valid token, an empty token and a bubble. Such a three-stage ring containing one valid token can be used to implement iterative computations where the result from the current step depends on the result from the previous step.

For two-phase designs, the situation is different. The static data-flow structure abstraction involves only tokens and bubbles, and the forward propagation of a token is associated with a transition of the C-element in the control circuit. Consequently, a three-stage ring using C-element based control circuits will contain two tokens and one bubble. The two tokens relate to the rising and falling transition of the wave that rotates in the ring. A consequence of this is that rings in general can only contain an even number of tokens.

Following the wave analogy, a fix would be that the ring alternately propagate a rising transition and a falling transition. This could be done by inverting the request and acknowledge signals when the input and output ports of a pipeline are connected to form a ring. We experimented with this line of thinking, but as static data-flow structures typically involve many coupled rings and short pipeline segments, possibly shared by several nested rings, there are severe constraints on where inverters can be inserted. This leads to a set of six design policies as opposed to only the two we present in the following. The solution we present is inspired by [19] where Roncken et al. write: *"Did you know that a ring of original Mousetrap modules cannot possibly hold an odd number of tokens? The same is true for rings of original Micropipeline and Click modules . . . This little recognized truth appears clearly in Fig. 8 . . . "*.

The figure referred to shows a so-called canopy graph where the throughput of a 24-stage ring is plotted as a function of the number of tokens in the ring. All graphs have the same shape, but except for the new pipeline template introduced in the paper, all other graphs only have data-points corresponding to an even number of tokens. In our view, this 24-stage ring context blurs the importance of the observation. The fact that rings with a single token are not possible is a major issue because it *precludes implementation of iterative/recursive computations.*

As observed in [19], the problem is that all previously published pipeline stages (C-element-based, Mousetrap [21] and Click [18]) produce transitions on Out_req and In_ack with the *same polarity* – typically the same polarity as the incoming In_req. This can be observed in Fig. 2(a) that shows the basic Click template. Following this observation, Roncken et al. then develop a new paradigm for structural design of asynchronous circuits using "link" and "joint" components that unifies all known pipeline templates. This represents a radical change of viewpoint. Our aim is to stick to the static data-flow structures view, and still use the set of handshake components introduced in [23, Ch. 3].

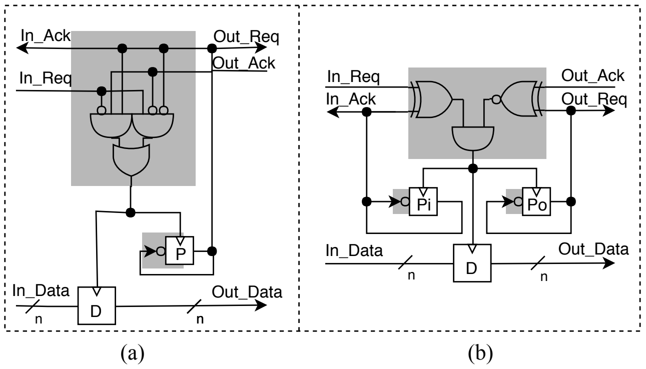

Fig. 2. (a) The Click template from [18]. The phase flip-flops are marked with P and the data registers are marked with D. (b) Our phase-decoupled Click template using separate phase flip-flops to generate In_Ack and Out_Req.

C. The phase-decoupled click template

A less radical solution to the "two-phase problem" involves some form of decoupling of the phase (rising or falling) of the handshake signals in the input and output channels on some handshake registers in the circuit. In a click-based handshake register, this can be done by introducing a second state-flip flop as shown in Fig. 2(b). This figure also shows an alternative implementation of the circuit generating the click signal. The shaded rectangles indicate combinational logic that is implemented in a single LUT in a FPGA.

Conceptually, in two-phase design, there is no difference between rising and falling signal transitions. However, when it comes to implementation, the designer has to decide on the initial signal levels. We note that all handshake channels are push channels and we adopt the following policy:

P1: For all channels in the circuit, a transition on the request wire (signaling that the driving circuit has a token) is followed by a transition on the acknowledge wire *with the same polarity* (signaling that the token has been received).

Following this policy, we see that the input channel has a new token when $\mathsf{In\_req} \neq \mathsf{In\_Ack}$ and that a token on the output channel has been received by the downstream neighbour when $\mathsf{Out\_Req} = \mathsf{Out\_Ack}$. The use of XOR and XNOR gates in the click control circuit in Fig. 2(b) shows this in a more explicit way. However, the function of the circuits generating the click signals in Figs. 2(a) and 2(b) are identical.

Fig. 3 shows static data-flow structure schematics of two-phase rings with two and three stages, both initialized to hold one token in the first stage (R1). The 0's and 1's annotated to the ports of a click stage are the initial values of the state flip flops (driving the In_Ack and Out_Req signals on the input and output ports respectively). We start by labeling the channel connecting R1 and R2 with 1-0 meaning that a token is about to propagate across this channel. The remaining channels are all marked 0-0 or 1-1 because R2 and R3 driving these channels hold bubbles. If the marking on the input and output channels of a handshake register are identical, these stages can be ordinary click-stages. Figs. 3(b) and 3(c) show alternative

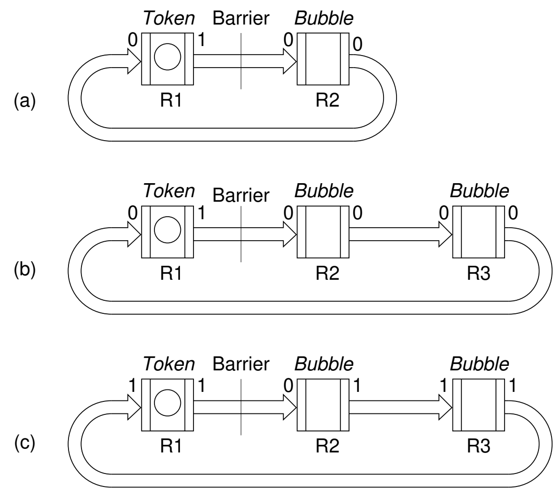

Fig. 3. Static data-flow structure schematics of a two-stage and a three-stage ring with a single token built from phase-decoupled click-stages. Initial values of all phase flip-flops are shown at the input and output ports where they directly drive the request and acknowledge signals.

initializations of the three-stage ring. In Fig. 3(b) R1 is a phase-decoupled click-stage and R2 and R3 are ordinary click-stages. In Fig. 3(c) R2 is a phase-decoupled click-stage and R1 and R3 are ordinary click-stages.

This hints that a component library may need multiple versions of the handshake register component. To simplify, we limit to the generic case, which is the phase-decoupled version, where every input and output channel has a state flip-flop for the request and acknowledge signals respectively. A possible optimization, that we have *not* implemented, is to allow the synthesis tool to perform what is called "register sharing" of the state flip-flops in a decoupled click stage.

We prefer the initial state shown in Fig. 3(b) to the initial states shown in Fig. 3(a) and 3(c) because it can be expressed in a single policy for initialization:

P2: All channels conveying tokens are initialized with Req = 1 and Ack = 0. All channels *not* conveying tokens are initialized with Req = 0 and Ack = 0. A handshake register where In_Ack ≠ Out_Req must be phase-decoupled. Handshake registers where In_Ack = Out_Req may be ordinary click-stages.

Fig. 4(a) shows the initial state of a three-stage ring with two tokens using conventional click element handshake registers. Fig. 4(b) shows the initialization that result from adhering to policy P2. Handshake registers R1 and R2 must now be phase-decoupled registers.

D. Initialization

The entire circuit is reset to a state where the state flip-flops in the click-stages are set according to the levels annotated to the channels in the static data-flow structures diagram, and where handshake registers in the data path that initially hold tokens are set to the desired initial values.

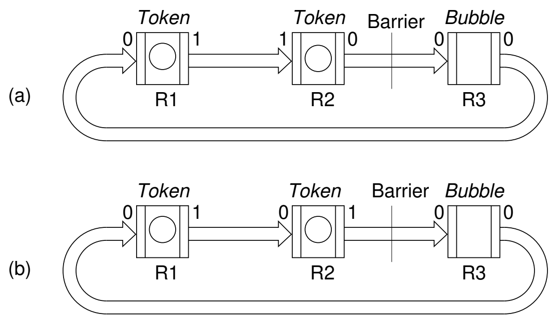

Fig. 4. Static data-flow structure schematics of a three-stage ring with two tokens. (a) Using the traditional initialization. (b) Initialized following policy P2. Handshake registers R1 and R2 must now be phase-decoupled.

In order to safely bring the circuits out of reset and into normal operation we introduce (where needed in order to keep the circuit "frozen" in its initial state), a barrier on channels that initially propagate tokens. These barriers are controlled by a global start-signal and they block the request signals on the corresponding channels. In this way, reset can be de-asserted, possibly with some skew, before the start-signal is asserted and the circuit starts operating. According to policy P2, a channel propagating a token has the request signal set to high. This means that the barrier must output a request signal that is low. An AND-gate is used to implement this forcing to zero.

IV. Implementation

We now present our implementations of the handshake components shown in Fig. 1.

A. Overview

Following [23, Ch. 3], our handshake registers (including their degenerate versions, Source and Sink) are the only components which actively implement the handshaking that makes the data-flow circuits operate. All other components (Func, Join, Fork, Merge, MUX and DEMUX) are passive/transparent from a handshaking point of view. This is in contrast to the components presented in [18], where for example the Join and DEMUX components buffer data. We prefer to see such components, which fuse for example a Join and a handshake register, or a DEMUX and a handshake register, as peephole optimizations that, if desired, may be performed later in the design process. In our view, a designer must first develop a design with the desired number of handshake registers, tokens and bubbles, and this should not be constrained by the number of joins and forks that happen to be in the circuit. Except for the handshake registers, all other components are passive/transparent to the handshaking. This means that they do not (need to) implement any phase decoupling. All of the components have been implemented and tested on a Nexys4 DDR board. The source files can be found in a Git-repository [11].

Listing 1. Phase-decoupled handshake register

```
entity decoupled_hs_reg is
  generic (
    DATA_WIDTH        : natural := DATA_WIDTH;
    VALUE             : natural  := 0;
    PHASE_INIT_IN     : std_logic := '0';
    PHASE_INIT_OUT    : std_logic := '0');
  port (rst   : in std_logic;
    -- Input channel
    in_ack    : out std_logic;
    in_req    : in std_logic;
    in_data   : in std_logic_vector
                      (DATA_WIDTH-1 downto 0);
    -- Output channel
    out_req   : out std_logic;
    out_data  : out std_logic_vector
                      (DATA_WIDTH-1 downto 0);
    out_ack   : in std_logic);
end decoupled_hs_reg;

architecture behavioral of decoupled_hs_reg is

signal phase_in, phase_out, click:std_logic;
signal data_sig:std_logic_vector
                (DATA_WIDTH-1 downto 0);
begin
  out_req  <= phase_out;
  in_ack   <= phase_in;
  out_data <= data_sig;

  click <= (in_req xor phase_in) and
           (out_ack xnor phase_out);

  clock_regs: process(click, rst)
  begin
    if rst = '1' then
      phase_in  <= PHASE_INIT_IN;
      phase_out <= PHASE_INIT_OUT;
      data_sig  <= std_logic_vector(to_unsigned
                   (VALUE, DATA_WIDTH));
    elsif rising_edge(click) then
      phase_in  <= not phase_in;
      phase_out <= not phase_out;
      data_sig  <= in_data;
    end if;
  end process;

end behavioral;
```

B. Handshake Register

The handshake register and the phase-decoupled handshake register are described in the previous section and their implementations can be seen in Fig. 2. The VHDL code for the phase-decoupled handshake register is shown in Listing 1.

When deciding the initial state of the phase flip-flops in a circuit, it is important to note that if a stage holds a token, the request and acknowledge signals in its output channel must have the opposite phases. If a stage represents a bubble, the request and acknowledge signals in its output channel must have the same phase as its downstream neighbor (c.f. policies P1 and P2).

The click pulse has a very short duration. This does not cause problems for the edge-triggered flip-flops (FF) on the FPGA we used for testing. If desired, the pulse-width can be increased by delaying the self-resetting of the control circuit.

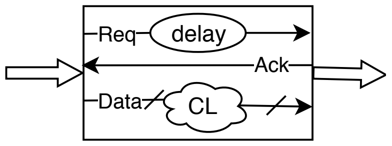

Fig. 5. Function Block

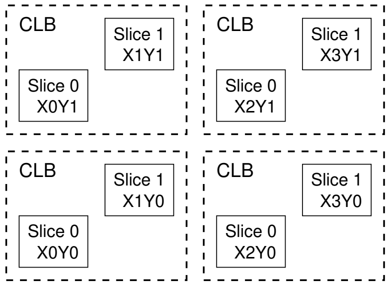

Fig. 6. A Xilinx FPGA is composed of slices (each containing a number of LUTs and DFFs). Slices are identified by Cartesian coordinates.

This can be done by adding a delay to the click signal (delaying the clocking of phase and data flip-flops) or by adding a delay after one of the phase flip-flops (delaying In_ack or Out_req).

C. Function blocks and delay elements

A function block is an ordinary combinational circuit extended with a request and an acknowledge signal, see Fig. 5. The request signal must be delayed by more than the propagation delay of the combinational circuit. For this, we use delay elements that are initially set with a very large safety margin. Later, based on post place and route simulation, the designer may manually trim down the delays to better match the propagation delay in the logic. Automation of this process is future work. This simple and straightforward implementation of a function block does not offer any joining of inputs or forking of outputs.

The delay elements are implemented following the guidelines outlined in [14]: a chain of LUTs whose relative physical placement on the FPGA is constrained/controlled. Listing 2 on the next page shows the VHDL code for the delay element. The LUT component used in this implementation has a single input and implements a buffer (a so-called LUT1 initialized with truth table "10"). In order to obtain reproducible delay values, the placement of the LUTs that implement delay elements is crucial. The *rloc* attribute allows the designer to specify the relative location of the slices in which the LUTs are placed. The relative placement is specified using the Cartesian coordinates *(X#Y#)* of the slices, as illustrated in Fig. 6. The VHDL code for the delay element is shown in Listing 2. As seen in the code the chain of LUTs are placed in a single column in slices X0Y0, X0Y1, X0Y2, ... in the following order, as specified by the Y-index of the slice: (0, 1, 0, 1, 0, 1, 0, 1, 2, 3, ...). By always placing the next LUT in a different slice, we get a higher delay due to the delay of the wires.

Listing 2. Delay element

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library unisim;
use unisim.vcomponents.lut1;

entity delay_element is
  generic(
    size  : natural range 1 to 30 := 10);
  port (
    d     : in std_logic; -- Data in
    z     : out std_logic);
end delay_element;

architecture lut of delay_element is
  component lut1
    generic (
      init  : bit_vector  := "10");
    port (
      I0    : in   std_ulogic;
     O      : out  std_ulogic
    );
  end component;
  -- Internal signals.
  signal s_connect: std_logic_vector(size downto 0);
  -- signal constraints
  attribute DONT_TOUCH : string;
  attribute DONT_TOUCH of s_connect : signal
                                      is "true";
  attribute rloc : string;

begin
  s_connect(0) <= d;
  -- Create a riple-chain of luts
  lut_chain : for index in 0 to (size-1) generate
    signal o : std_logic;
    type y_placement is array
              (integer range 0 to 29) of integer;
    -- y coordinates for relative location
    constant y_val: y_placement := (0,1,0,1,0,1,0,1,
       2,3,2,3,2,3,2,3,4,5,4,5,4,5,4,5,6,7,6,7,6,7);

    attribute rloc of delay_lut : label is
              "X0Y" & integer 'image(y_val(index));

    begin
      delay_lut: lut1
        generic map(
          init => "10") --truth table
        port map(
          I0 => s_connect(index),
         O  => o
        );

    s_connect(index +1) <= o after 1 ns;
  end generate lut_chain;
  -- Connect the output of delay element
  z  <= s_connect(size);

end lut;
```

D. Join and Fork

Simple and straightforward implementations of the join and fork components are shown in Fig. 7. They are textbook implementations [23, Sect. 5.2] using a click-circuit to implement the functionality of C-element.

Following design policies P1 and P2, the simple join in Fig. 7 can always be used. The phase flip-flop is initialized according to the state of the input and output channels. Because the component is transparent to handshaking, these

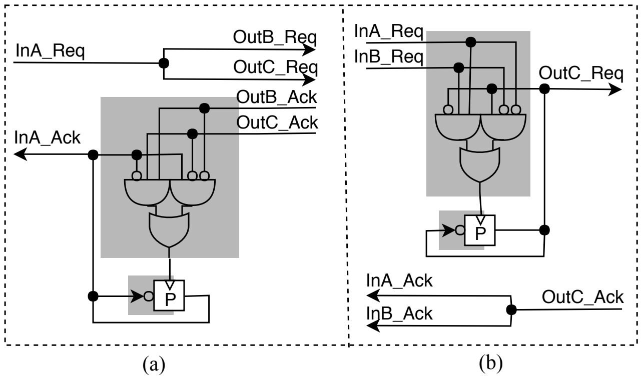

Fig. 7. (a) Fork. (b) Join.

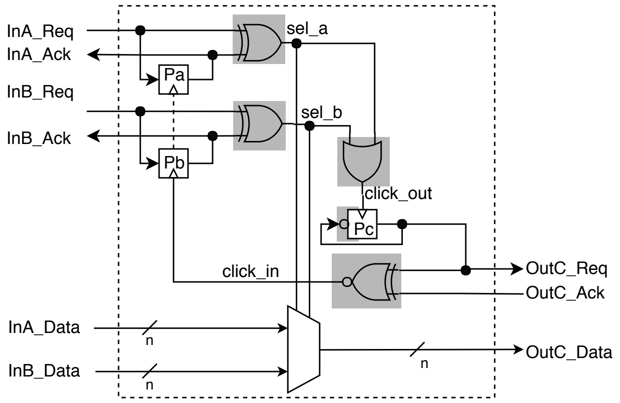

Fig. 8. Merge

are guaranteed to be in phase. Again, the shaded rectangles indicate combinational logic that is implemented in a single LUT in a FPGA.

E. Merge

The implementation of the Merge is shown in Fig. 8. It assumes mutually exclusive inputs and therefore uses separate phase flip-flops (denoted Pa and Pb) in the input ports. As the input and output phase flip-flops are clocked by separate signals, it also needs a separate phase flip-flop (denoted Pc) in the output port.

The circuit functions as follows: A transition on either InA_Req or InB_Req asserts either Sel_A or Sel_B and the multiplexor propagates the proper input data to the output (Out_Data). This also creates a rising edge on the signal click_out, which causes a transition on Out_Req. Finally, this creates a (silent) falling transition on signal click_in. When the right hand environment later acknowledges by transitioning signal Out_Ack, this causes a rising edge on signal click_in. This clocks both Pa and Pb and causes a transition on InA_Ack if the operation of the merge started by a transition on InA_Req or a transition on InB_Ack if the operation of the merge started by a transition on InB_Req.

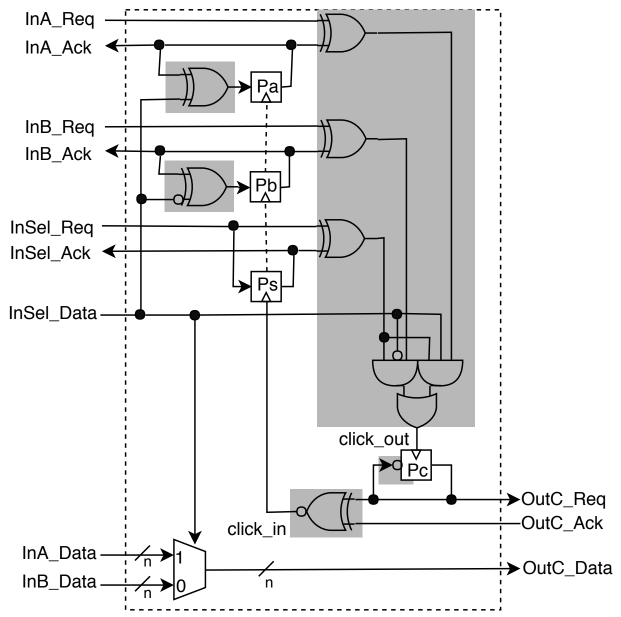

Fig. 9. MUX

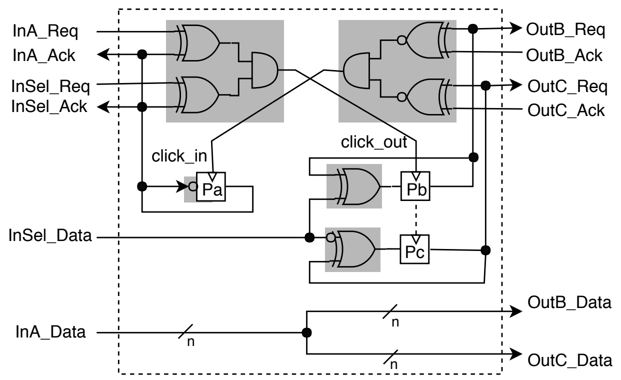

Fig. 10. DEMUX

This way of using a phase-flip-flop to produce an acknowledge based on the corresponding request is a small variation that we prefer instead of the clock-gating used in in the buffered Merge in [18] and the plain Merge described in [13]. A gated clock produced by an AND-gate requires the gating signal to be stable in a time window overlapping the period where the clock signal is high. Our solution avoids this timing requirement.

F. MUX and DEMUX

The MUX and DEMUX components are used to implement conditional flow control. The MUX has two input channels (InA, InB), a selection (input) channel for choosing between InA and InB and an output channel (OutC). Fig. 9 shows the implementation of the MUX. The phase flip-flops Pa, Pb and Ps are all clocked on every transition of the incoming acknowledge by the same signal derived from the function OutC_Req = OutC_Ack. The phase flip-flop Pc drives the request signal of the output channel (signal OutC_Req) and is toggled whenever there is a token on the selector channel and the selected input. Similar to the Merge, the MUX has phase-decoupled channels due to the nature of its function.

Fig. 10 shows the implementation of the DEMUX (inspired by [13]). It has two input channels (InA and InSel) and two output channels (OutB and OutC). The component joins the two inputs and produces an output on the selected channel. Similar to the MUX, the DEMUX has multiple internal phase flip-flops. The phase flip-flops Pb and Pc are clocked when both request signals on the input channels transition. Phase flip-flop Pa (participating in the input channel handshakes) is clocked whenever an acknowledgement is received (as indicated by the following expression: OutB_Ack = OutB_Req) $\wedge$ (OutC_Ack = OutC_Req). Again, we prefer this style of clocking to the gated clocking used in the components described in [13], [18].

V. Peephole optimizations

It is possible to reduce the hardware cost of a circuit by performing peephole optimizations, where certain combinations of handshake components are replaced by a single fused circuit. All of these optimizations involve merging handshake registers and one or more of the passive components. The original click-paper [18] showed how easy it is to extend the click-template with join-functionality on the input and fork functionality on the output. The same is the case for our phase-decoupled handshake register. Below we describe a range of such fused components.

A. Join+Register+Fork, Join+Register and Register+Fork

The schematic symbols for a handshake register fused with a Join and/or a Fork are shown in Fig. 11, and Fig. 12 shows

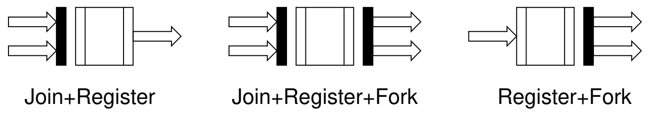

Fig. 11. Schematic symbols for a handshake register fused with a Join and/or a Fork.

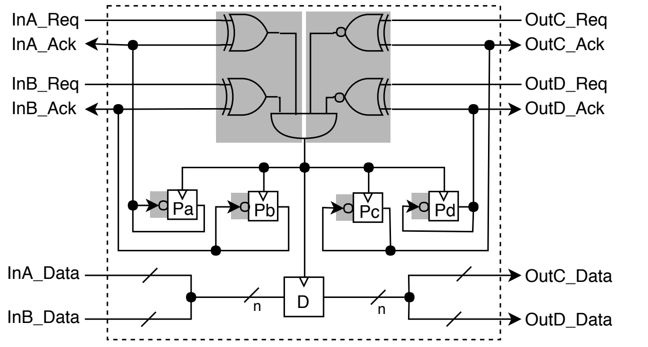

Fig. 12. Implementation of a phase-decoupled fused Join+Register+Fork component with two input channels (A and B) and two output channels (C and D).

the implementation of a fused Join+Register+Fork circuit with two input channels and two output channels. For simplicity the figure shows a design with separate phase flip-flops for each input and output channel. As all phase flip-flops are clocked by the same signal, at most two phase flip-flops are needed – phase flip-flops initialized to the same value can share a single flip-flop. It is easy to see how input and output channels can be dropped from or added to the circuit by dropping or adding XOR or XNOR gates.

B. *Register+Merge, Register+MUX and Register+DEMUX*

We have developed fused versions of a handshake register fused with a Merge or a MUX or a DEMUX. All implementations are in the Git-repository, but the implementations of these fused components are only marginally smaller and faster than compositions of the basic components. Because of this, and due to space limitations, we do not include the descriptions here. This is also in line with our overall goal of simplifying the design process.

C. *Join+Func+Register+Fork*

We considered fusing components that implement a complete pipeline stage, i.e., a fused Join+CL+Register+Fork circuit. This could be done by fusing the Join and the Fork into the handshake register. In the general case, where nothing is known about the surrounding circuitry, this would require a matched delay element for each input channel. As the matched delay elements are expensive to implement in FPGA technology, we decided not to pursue this idea.

VI. Design examples

In this section, we illustrate the use of the components and the design methodology by showing and explaining the design and implementation of two small circuits that contain multiple coupled rings and pipeline segments. Both circuits have been implemented and tested on the Nexys4DDR FPGA-board and the code is available in the Git-repository.

A. *Fibonacci*

The Fibonacci circuit has no inputs; it simply computes and outputs the sequence of Fibonacci numbers (0, 1, 1, 2, 3, 5, ...). Our implementation using phase-decoupled two-phase handshake components is shown in Fig. 13. The figure also shows the initial state of the circuit and the use of fused components. A matched delay is only required in the function block (CL0), since the LUTs generating the click signals in the other components normally provide sufficient delay margins.

The circuit is initialized with a token in each of the two Register+Fork components. The design consists of two nested rings: an inner ring containing RF0 $\rightarrow$ J0 $\rightarrow$ CL0 $\rightarrow$ R0 and an outer ring containing the same components *and* handshake register RF1. The inner ring has two handshake registers and one token. The outer ring has three handshake registers and two tokens. By following policy P2, we can ensure correct initialization of both rings. Notice that the schematic shows no annotations on the input channels of join J0; our passive

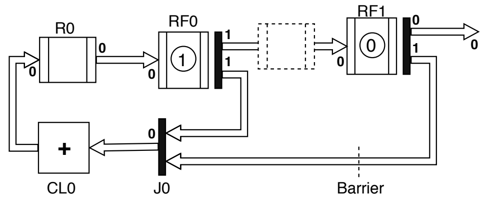

Fig. 13. Schematic of the Fibonacci circuit. The handshake register shown using dashed lines could be added for a more straightforward implementation, but without it, the circuit offers better illustration of the spread-token operation.

join simply forwards the acknowledge signal from the output channel to the two input channels (without any buffering). This acknowledge signal is produced by handshake register R0.

When the go signal is asserted and the barrier opens, the circuit starts: J0 joins the tokens from RF0 and RF1 and the resulting (single) token spreads across the J0, CL0 (the adder) and into R0. At the same time, the environment consumes (a forked copy of) the token in RF1. This spread-token operation is mentioned/assumed in [23], and studied in detail in [22]. We have chosen this design, instead of the more straight-forward implementation (with an additional handshake register shown using dashed lines in Fig. 13), to better illustrate the spread token semantics. The Git repository contains an illustration of the spread token operation of the Fibonacci circuit (as well as for the GCD circuit presented in the next section).

B. *Greatest common divisor*

The greatest common divisor (GCD) circuit shown in Fig. 14 was designed after [23, Sec. 3.7] with small modifications. As we use two-phase handshaking, we need fewer handshake registers. In addition, we use a Merge, ME0, instead of a MUX at the end of the if-then-else construct. The circuit is initialized according to policies P1 and P2 with a token (with value '1') in handshake register R0 and with bubbles in the remaining handshake registers. The circuit has no barrier since, after reset, it waits for a token on the input channel.

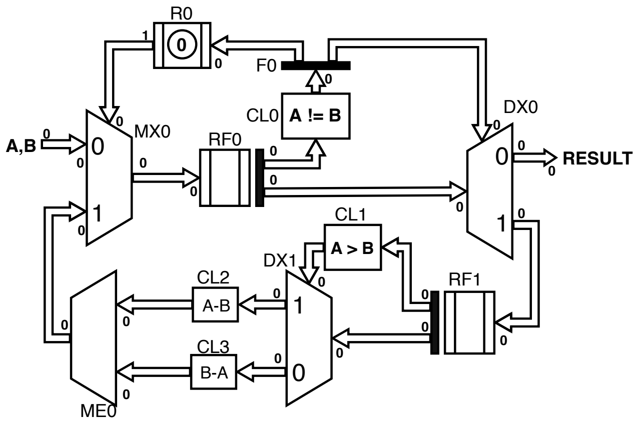

Fig. 14. Schematic of the GCD circuit.

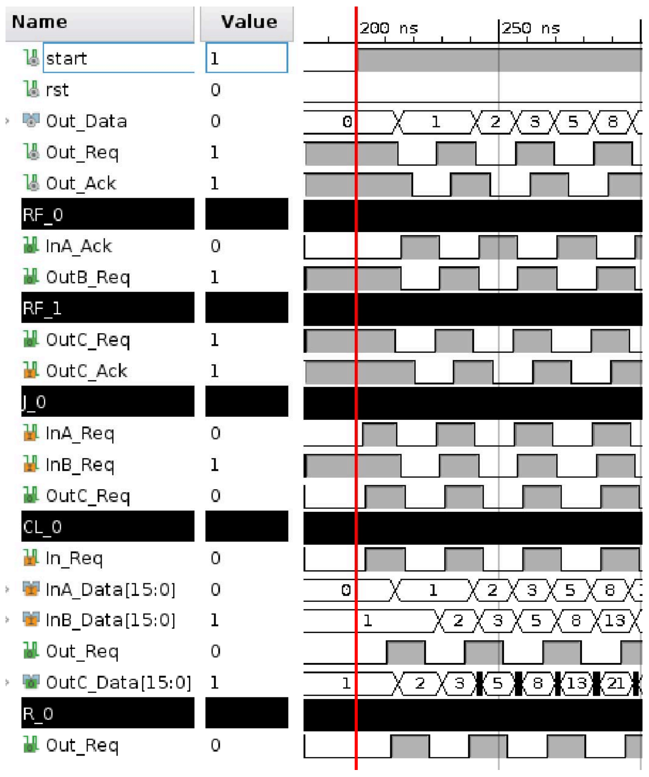

Fig. 15. Post-synthesis timing simulation of the Fibonacci circuit.

Notice that R0 has different phases on the input and output channels. This is because the ring MX0 → RF0 → F0 → R0 has a single token. The other rings in this circuit are: MX0 → RF0 → DX0 → RF1 → DX1 → ME0 and MX0 → RF0 → F0 → DX0 → RF1 → DX1 → ME0. In all of the rings the tokens eventually get spread across several components as seen in the step-wise illustration provided in the Git repository.

C. FPGA Implementation

Both the Fibonacci circuit and the GCD circuit have been implemented on a Digilent Nexys4DDR FPGA-board (with a Xilinx Artix 7 chip) and the circuits have been operated manually. Input channels are implemented using a debounced pushbutton for the request signal, a set of switches for the data, and an LED for the acknowledge signal. Output channels are implemented using LEDs for the request signal and the data signals, and a debounced pushbutton for the acknowledge signal. The corresponding XDC-files (constraint files specifying the pinout) are included in the design sources in the GitHub repository. In the component source files, the *"DONT_TOUCH"* attribute is set for combinational signals and registers, to force the place and route tool to keep the signals. Therefore, minimal project setup is necessary for using the designs.

A post synthesis simulation of the Fibonacci circuit is shown in Fig. 15. The first five signals show the environment signals. Below these, some select internal signals are also plotted and grouped by component. A file showing a similar simulation of the GCD circuit is included in the GitHub repository.

Both circuits work correctly in simulation *and* on the actual FPGA board. As this paper focuses on the design process and on FPGA-prototyping, we use delay elements with very conservative (high) values. For this reason, it does not make sense to report performance measures. A more detailed discussion of performance and performance optimization is beyond the scope of this paper.

VII. Conclusion

This paper presented a simple, structural approach to the design and FPGA implementation of asynchronous circuits using data-flow handshake components. The aim of the paper is to enable students, and others who are in the process of learning asynchronous design, to design and implement small asynchronous circuits using FPGA technology.

The components use two-phase bundled-data handshaking and are implemented using a novel phase-decoupled extension of the click-element template. This phase-decoupling allow implementation of nested rings with any number of tokens including the most typical situation – rings with a single token. In this way, two-phase bundled-data implementations of iterative/recursive functions are now possible.

The paper presents the implementation (described in VHDL) of all components in the library and it illustrates the design method using two example circuits: Fibonacci and greatest common divisor. All code, including the design examples, is available as open source.

Source code

The paper is accompanied by an on-line repository [11] containing: (a) Schematics and VHDL source code for all the handshake components. (b) Schematics and source code for the two design examples including VHDL test-benches for simulation. (c) A sequence of snapshots of the schematics illustrating the token-flow operation of the circuits.

References

[1] Filipp Akopyan, Jun Sawada, Andrew Cassidy, et al. True North: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. *IEEE Tran. Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, 2015.
[2] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. Int'l. Conf. Computer-Aided Design*, pages 262–265, November 1989.
[3] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras. xmas: Quick formal modeling of communication fabrics to enable verification. *IEEE Design Test of Computers*, 29(3):80–88, 2012.
[4] M. Davies, N. Srinivasa, T. Lin, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.
[5] Europractice. URL: http://www.europractice.com.
[6] P. D. Ferguson, A. Efthymiou, T. Arslan, and D. Hume. Optimising self-timed FPGA circuits. In *Proc. Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 563–570, 2010.
[7] Alberto Ghiribaldi, Davide Bertozzi, and Steven M. Nowick. A transition-signaling bundled data NoC switch architecture for cost-effective GALS multicore systems. *Proceedings - Design, Automation, and Test in Europe Conference and Exhibition*, pages 332–337, 2013.
[8] Mark R. Greenstreet, Jørgen Staunstrup, and Ted E. Williams. Self-timed iteration. In Carlo H. Séquin, editor, *Proceedings of VLSI '87*, pages 269–282. IFIP, August 1987.

[9] Quoc Thai Ho, Jean-Baptiste Rigaud, Laurent Fesquet, Marc Renaudin, and Robin Rolland. Implementing asynchronous circuits on LUT based FPGAs. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 36–46. Springer, 2002.

[10] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[11] https://github.com/zuzkajelcicova/Async-Click-Library.

[12] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 127–136, 2018.

[13] I Kotleas, D.R. Humphreys, R.B. Sørensen, E. Kasapaki, F. Brandner, and J. Sparsø. A Loosely Synchronizing Asynchronous Router for TDM-Scheduled NOCs. In *Proc. IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 151–158, 2014.

[14] Jon Neerup Lassen. FPGA prototyping of asynchronous networks-on chip. Master's thesis, Dept. of Information Technology, Technical University of Denmark, 2008. Report IMM-M.Sc.-2008-26) available at http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=7126.

[15] Rajit Manohar. Reconfigurable asynchronous logic. In *Proc. Custom Integrated Circuits Conference (CICC)*, pages 13–20. IEEE, 2006.

[16] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[17] David E. Muller. Asynchronous logics and application to information processing. In H. Aiken and W. F. Main, editors, *Proc. Symp. on Application of Switching Theory in Space Technology*, pages 289–297. Stanford University Press, 1963.

[18] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An implementation style for data-driven compilation. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 3–14, 2010.

[19] M. Roncken, S. M. Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland. Naturalized communication and testing. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 77–84, 2015.

[20] Basit Riaz Sheikh and Rajit Manohar. An asynchronous floating-point multiplier. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 89–96, 2012.

[21] M. Singh and SM Nowick. MOUSETRAP: High-speed transition-signaling asynchronous pipelines. *IEEE Transactions on VLSI Systems*, 15(6):684–698, 2007.

[22] Danil Sokolov, Ivan Poliakov, and Alex Yakovlev. Analysis of static data flow structures. *Fundamenta Informaticae*, 88(4):581–610, 2008.

[23] J. Sparsø and S. Furber, editors. *Principles of asynchronous circuit design – A systems perspective*. Kluwer Academic Publishers, 2001.

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

[25] C. H. van Berkel, C. Niessen, M. Rem, and R. J. J. Saeijs. VLSI programming and silicon compilation: A novel approach from Philips research. In *Proceedings of the 1988 IEEE International Conference on Computer Design*, pages 150–166. IEEE, 1988.