Burst-Mode Locally-Clocked Asynchronous Circuits

Charles Eric LaForest

December 2005
University of Waterloo
Independent Studies

Abstract

An active and a passive version of an asynchronous four-phase handshake circuit are specified as Burst-Mode machines and synthesized as hazard-free two-level logic. These circuits are used to build self-timed modules which can generate local clocks with a variable period, exchange data with other asynchronous systems, or act as input/output ports with provisions against metastability. A variation of each module with early handshake termination is also presented. Finally, some notes outline the use of these modules to create an asynchronous implementation of the Gullwing computer architecture.

This page intentionally left blank.

Contents

List of Figures

List of Algorithms

1 Introduction

The Gullwing Processor [LaF05] is a second-generation stack computer closely based on the works of Charles H. ("Chuck") Moore [Moo01][1]. The original state machine description (Algorithm 3) and planned system-level design assume a single synchronous clock for multiple processors and their shared memories to obtain efficient communication and deterministic operation. The performance and reliability of this approach are compromised by uncertainties in clock distribution (skew) and clock period (jitter), which impose additional design effort and force conservative timing margins. Additionally, the strong clock drivers required to reduce skew generate heat which slows down nearby circuits and further increases the timing margins required. The apparently obvious solution of having multiple clocks over domains small enough to reduce these problems results in poor performance since communicating data between two systems with local, unrelated clocks requires several extra cycles to resolve potential metastable states[2].

One kind of solution, asynchronous systems, discards the global clock and uses multiple interacting control elements instead [vBJN99]. The asynchronous approach presented here uses Fuhrer and Nowick's MINIMALIST Burst-Mode machine specification and synthesis tool [FN01] to build a pair of versatile building blocks which implement a four-phase handshake [PvB95]. These blocks are used to build self-timed, local, variable-period clocks which allow for tighter timing margins, asynchronous communication channels which make smaller clock domains practical, and reliable input/output ports that handle potential metastability when reading from the environment.

2 Literature Search

Beyond the sources listed in the body of this report, the following references were useful:

- [BS91][BNY99][ND91][NYD92][YDN93][HS02, LN02] cover Burst-Mode circuit theory and implementations.
- Parts of [Mye01] are a good introduction to Petri Nets and communication protocols.
- [TLE96] contains an in-depth discussion of hazards (glitches) in logic circuits.

[1] The F21 processor is also notable, but unpublished. See http://ultratechnology.com/f21cpu.html
[2] A metastable state is one which is neither of the usual high/low states.

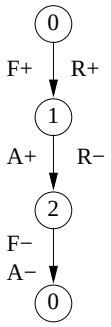

Figure 1: Active Element Specification

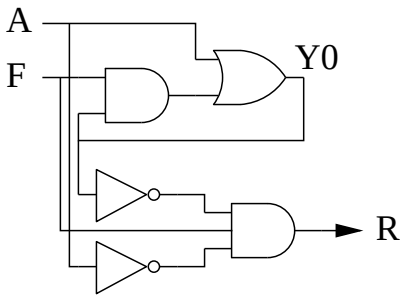

Figure 2: Active Element Schematic

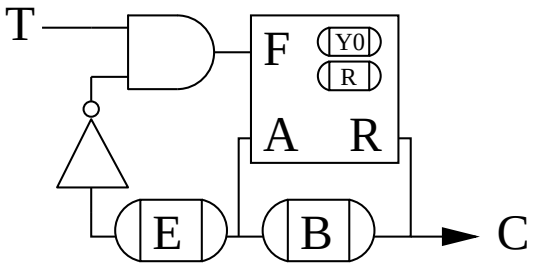

Figure 3: Active Module

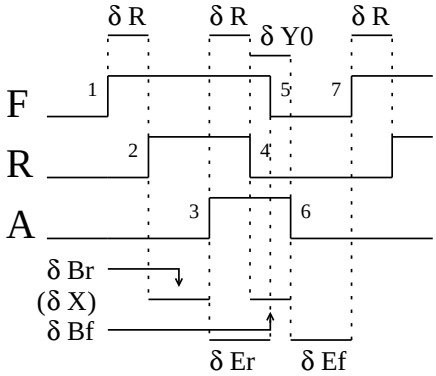

Figure 4: Active Module Waveform

3 Active Circuit

The active circuit is the main building block for all the systems that follow. It implements the active half of a four-phase handshake [PvB95].

3.1 Specification

Figure 1 describes the Burst-Mode specification of the active element, with the input signals to the left and the output signals to the right. All signal transitions are monotonic and only the ones specified on a given edge are allowed. A state is stable until all specified input transitions occur (in any order), at which point the output transitions occur (in any order) and the state changes to its next value. All inputs must remain stable until the internal state and output signal transitions have settled to their new values[3]. All edges entering a given state must result in the same configuration of input and output levels.

The initial state is 0 with all signals low. Raising the Function (F) input raises the Request (R) output until the Acknowledge (A) input rises. After R falls, the circuit returns to its initial state once F and A fall.

3.2 Synthesis

The MINIMALIST tool takes a Burst-Mode specification, performs state minimization, and returns a hazard-free[4] two-level Boolean gate implementation[5] of the corresponding Huffman Finite State Machine[6] (figure 2). In this case the three states are minimized to two and represented by the internal signal Y0. The output R is computed as a function of the inputs F and A, and of the internal state Y0:

Algorithm 1 Active Element Boolean Equations

```
Y0 = A + (F * Y0)
R = F * A' * Y0'
```

3.3 Module

While the active element performs the logic of a four-phase handshake, the timing is implemented by controlling the active element with asymmetric delay elements [MC79, Sei79] to form the active module shown in figure 3. The Request (R) signal becomes the Clock (C) output, and is fed through a Feedback delay (B) to generate the Acknowledge (A) signal. The delayed R signal is further delayed through an Enable (E) delay and used to gate the Trigger (T) input to the Function (F) signal.

3.4 Waveform

Figure 4 describes the typical operation of the active module. The Greek lowercase delta (δ) prefixes internal and external delays. For the asymmetric delay elements, the suffixes "r" and "f" denote the delays for the rising and falling edges of their input.

The δBr delay controls the duration of the high part of R and is determined by δR and the timing requirements of the logic driven by R. The δBf delay matches the internal delay δY0. It holds A high until the internal state Y0 has settled in order operate in the fundamental mode, otherwise R could glitch high.

The δEr delay must be at least as long as δR since the specification shows F falling after R. In this case, the implementation may allow this delay to be omitted. A potential problem exists if δEr is greater than δR+δBf since it would overlap with δEf and this may not be possible, depending on the implementation of the asymmetric delay element E. The δEf delay controls the duration of the low part of R and is determined by the settling time of the logic driving F[7]. This prevents glitches from propagating through and causing malfunctions.

[3] This is "fundamental mode" operation, contrasted with "input/output mode" where inputs may change concurrently with outputs and states.

[4] Free of glitches at the output for the specified input transitions.

[5] It is also possible to synthesize to generalized C-Elements (gC), but doing so here would obscure the focus of the discussion.

[6] Huffman FSM are asynchronous Mealy machines that store their state in combinational loops instead of clocked storage elements.

[7] It is assumed that the output of the logic driving F will not fall before the completion of the handshake at edge 6, since it it calculated from the output of registers.

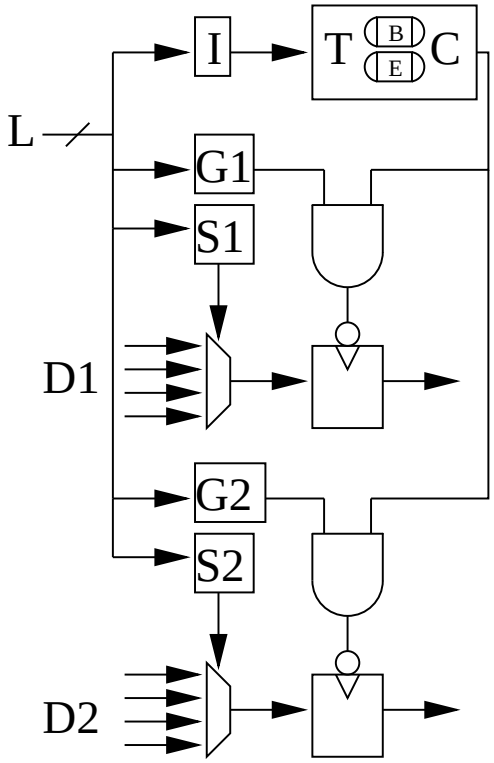

Figure 5: Local Clock Generation

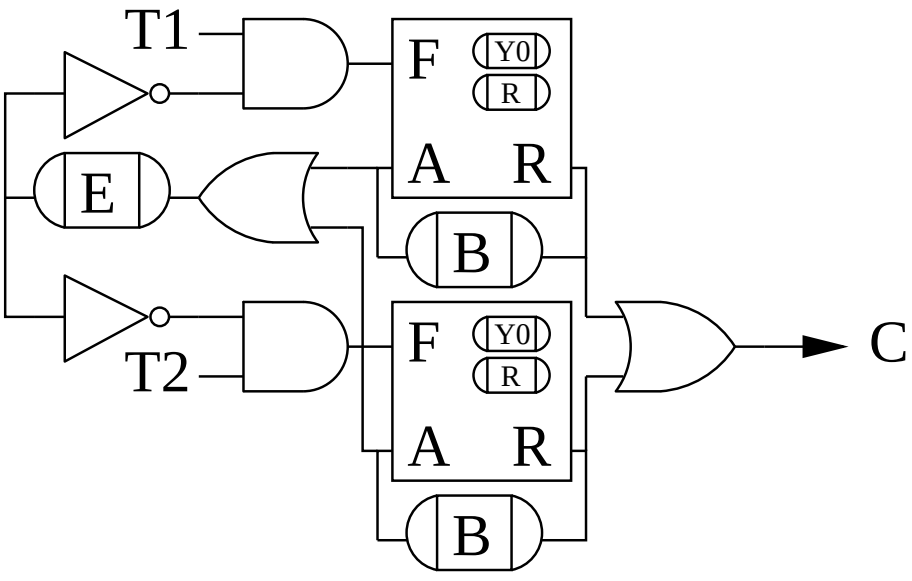

Figure 6: Concatenated Active Modules

3.5 Local Clock Generation

Figure 5 shows how an active module controls a number of registers. Control Lines (L) are decoded by I to drive the active module Trigger (T), by G1 and G2 to gate the Clock (C), and by S1 and S2 to select the source Data (D1 and D2) to the negative edge-triggered registers. The internal delays E and B generate suitable low and high phases for C. The low phase must be long enough to let the new data exit the registers and allow for the output of the decoders to settle to its final value. The high phase must be long enough for the selected Data to propagate through combinational logic and be stable at the input of the registers long enough to meet their setup time requirement.

3.5.1 Module Concatenation

The combinational delay between two registers depends on the source selected. On the other hand, the decoder delays are fairly uniform since the decoders are of similar size and complexity. A single active module has to assume the worst case for both delays. Adjusting the duration of the high phase to match the combinational delay of the selected source would improve performance significantly [PH04, p. 315]. Furthermore, similar operations have similar combinational delays and group into a small number of types (register transfers, arithmetic calculations, bit-wise Boolean operations, memory accesses, etc...), each with its own worst case delay, only one of which is the greatest, global worst case.

Figure 6 shows how to concatenate multiple active modules to generate a clock signal with a selectable high phase and a common low phase. At the end of the low phase one of the two Trigger inputs (T1 and T2) asserts and begins a handshake cycle in the corresponding active module, with a high phase determined by that module's Feedback delay (B). The Enable delay (E) is common to all the modules since the low phase must cover the worst case delay of the logic that drives each Trigger and keep them all disabled until the current cycle ends, otherwise a glitch may get through.

3.5.2 Clock Domain

Local clock generation reduces design effort by allowing the use of datapaths identical to those in synchronous systems. However, this makes it as vulnerable to clock skew and jitter as a global free-running clock. Worse yet, the irregular nature of the local clock makes it impossible to use Phase-Locked Loops (PLLs) and Delay-Locked Loops (DLLs)[8] to alleviate these problems. Thus, the domain of a local clock must be kept small enough to avoid amounts of skew and jitter that would limit performance. Coordinating these multiple small domains is the topic of section 3.6.

[8] These are circuits, crucial to large synchronous systems, which dynamically adjust the relative phase and delay of separate clocks.

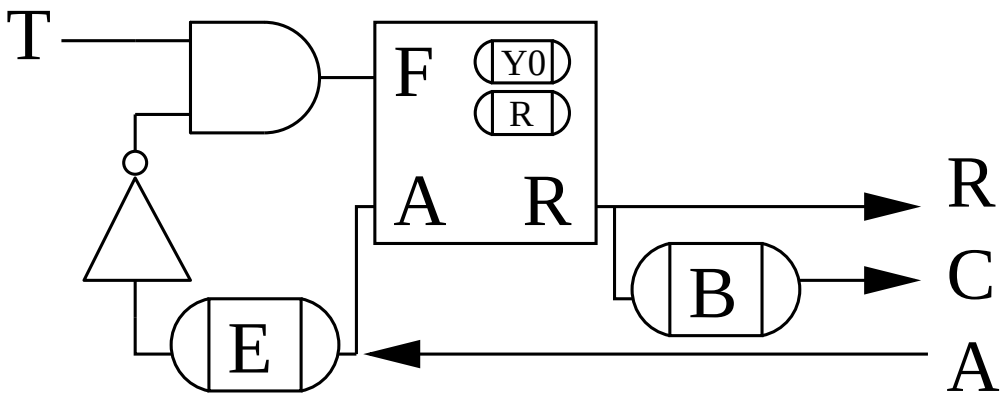

Figure 7: Active Module Synchronizer

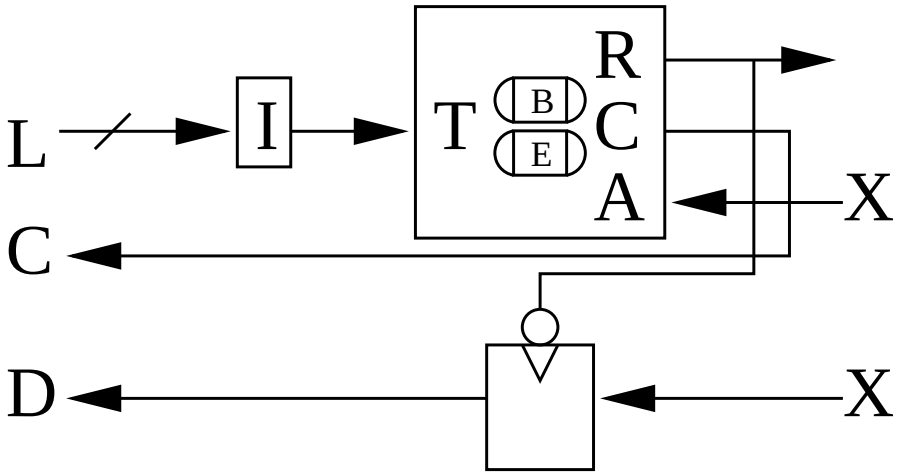

Figure 8: Active Synchronizer and Input/Output Port

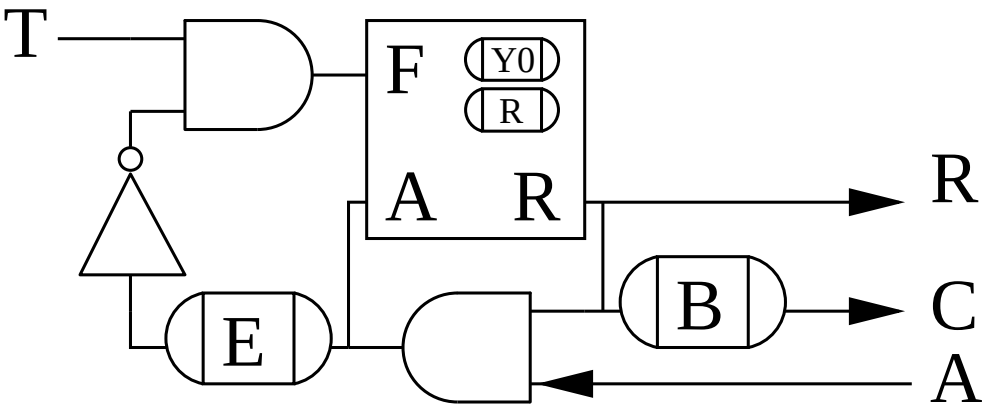

Figure 9: Active Module Synchronizer (Early Termination)

3.6 Communication

Registers controlled by different, non-concatenated active modules have local clocks which are unrelated in phase and frequency, and are thus asynchronous to each other. Transferring data between asynchronous local clock domains cannot be done directly since a receiving register may read from the output of a sending register while the latter is changing state, resulting in a metastable condition in the former. To reliably transfer data across domains, the sender must signal that there is data available and the receiver must acknowledge that it has received the data before the sender can change state again.

3.6.1 Synchronizer

Figure 7 shows how this handshake is implemented by opening the Request-Acknowledge loop of the active module from figure 3. The operation is identical to the one described in figure 4 except that the δBr and δBf delays are now reaction time delays δX from the environment. The Clock (C) is no longer a direct copy of the Request (R) signal, but a delayed one provided by the re-purposed delay element B.

Figure 8 shows how this new active synchronizer integrates in the local clock generation system from section 3.5. When triggered, the active synchronizer raises R, the register begins sampling the data from the environment X, the Clock (C) rises shortly thereafter. The entire system remains in a stable high phase state until the environment raises A, signalling that the input to the register is valid[9]. The active synchronizer then drops R to signal the receipt of the data, which is now latched into the register and visible internally, and C drops after a suitable propagation delay. The system enters a stable low phase until the environment drops A, completing the handshake. Reversing the direction and edge polarity of the register and following the same steps allows reliable writing to the environment.

3.6.2 Input/Output

If the interface with the environment cannot be made in a coordinated manner, the active synchronizer can be converted into a plain input/output port by looping back R to A (by shorting their pins together, for example) and if reading from the environment, by extending the delay of a falling edge through B to allow for the register to resolve a potential metastable state before internal registers read its output[10].

3.6.3 Early Termination

The time between the receipt of data and the end of a handshake (edges 4 to 6 in figure 4) performs only a synchronization function since the data has already been latched. Figure 9 shows a modified synchronizer which gates off A when R falls in order to abridge this part of a handshake. The low phase then completes without waiting for the environment to release its acknowledgement (A). The disadvantage of this method is that the system may initiate a new handshake while A is still high, which would prematurely end the high phase before the environment had time to respond[11].

[9] The signal A actually rises a short time *after* the data is valid to provide margin against uneven transit time (skew) and to meet the setup time of the register [PvB95].

[10] If the synchronizer will never be used as an input/output port, then the register can be omitted and the B delay reduced accordingly.

[11] It also allows two active synchronizers to communicate by cross-connecting their R and A lines. See Appendix A.5.

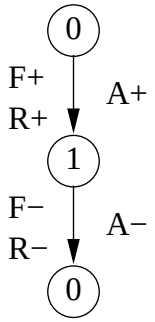

Figure 10: Passive Element Specification

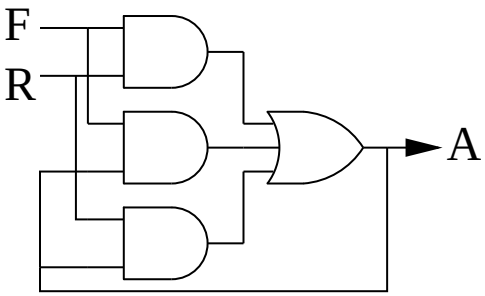

Figure 11: Passive Element Schematic

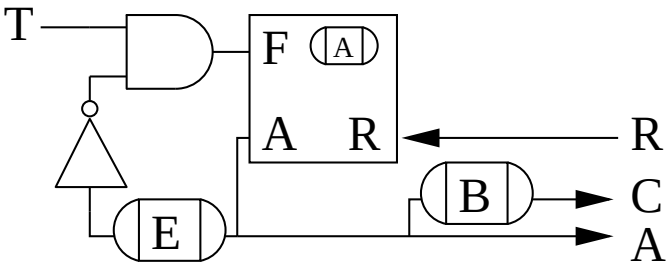

Figure 12: Passive Module Synchronizer

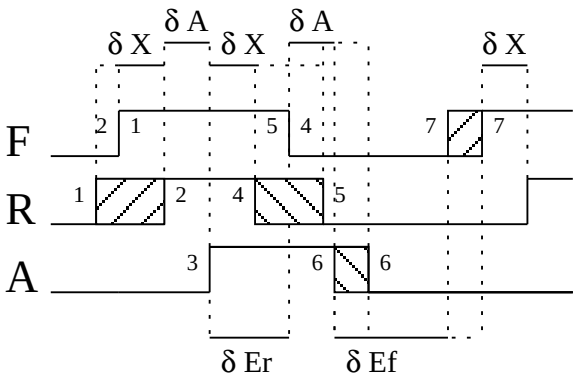

Figure 13: Passive Module Synchronizer Waveform

4 Passive Circuit

The passive element is the complement of the active element. It implements the passive half of a four-phase handshake [PvB95].

4.1 Specification

Figure 10 describes the Burst-Mode specification of the passive element, with the input signals to the left and the output signals to the right. All signal transitions are monotonic and only the ones specified on a given edge are allowed. A state is stable until all specified input transitions occur (in any order), at which point the output transitions occur (in any order) and the state changes to its next value. All inputs must remain stable until the internal state and output signal transitions have settled to their new values[12]. All edges entering a given state must result in the same configuration of input and output levels.

The initial state is 0 with all signals low. Raising the Function (F) and Request (R) inputs raises the Acknowledge (A) output, which remains high until both F and R fall[13].

4.2 Synthesis

The MINIMALIST tool takes a Burst-Mode specification, performs state minimization, and returns a hazard-free[14] two-level Boolean gate implementation[15] of the corresponding Huffman Finite State Machine[16] (figure 11). In this case the two states are minimized to one, which is held by having the output A fed back as an input. The output A is computed as a function of the inputs F and R, and of A itself:

Algorithm 2 Passive Element Boolean Equation

```
A = (F * R) + (F * A) + (R * A)
```

4.3 Module

While the passive element performs the logic of a four-phase handshake, the timing is implemented by controlling the passive element with asymmetric delay elements [MC79, Sei79] to form the passive module shown in figure 12. The Acknowledge (A) signal becomes the Clock (C) output, once fed through a delay (B), and the A signal is delayed through an Enable (E) delay and used to gate the Trigger (T) input to the Function (F) signal.

4.4 Waveform

Figure 13 describes the typical operation of the active module. The Greek lowercase delta (δ) prefixes internal and external delays. For the asymmetric delay elements, the suffixes "r" and "f" denote the delays for the rising and falling edges of their input. The δEr delay controls the minimum duration of the high phase of A and is determined by the timing requirements of the logic being driven by A. The δEf delay controls the duration of the low phase of A and is determined by the settling time of the logic driving F[17] in order to prevent glitches from propagating through and causing malfunctions.

[12] This is "fundamental mode" operation, contrasted with "input/output mode" where inputs may change concurrently with outputs and states.

[13] This is the same behaviour as a Muller C-Element, one of the fundamental asynchronous control circuits. This implementation has not yet been compared to existing ones.

[14] Free of glitches at the output for the specified input transitions.

[15] It is also possible to synthesize to generalized C-Elements (gC), but doing so here would obscure the focus of the discussion.

[16] Huffman FSM are asynchronous Mealy machines that store their state in combinational loops instead of clocked storage elements.

[17] It is assumed that the output of the logic driving F will not fall before the middle of the handshake at edge 3. This guaranteed by the delays on A, which drives the local clock controlling the registers from which F is computed.

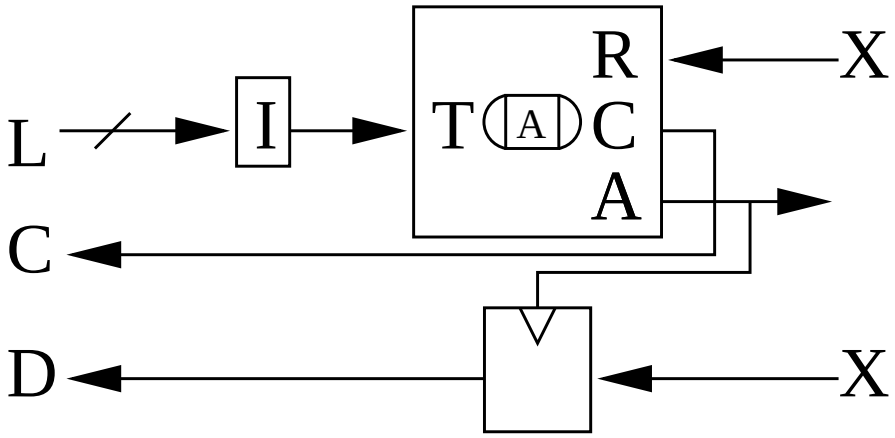

Figure 14: Passive Synchronizer and Input/Output Port

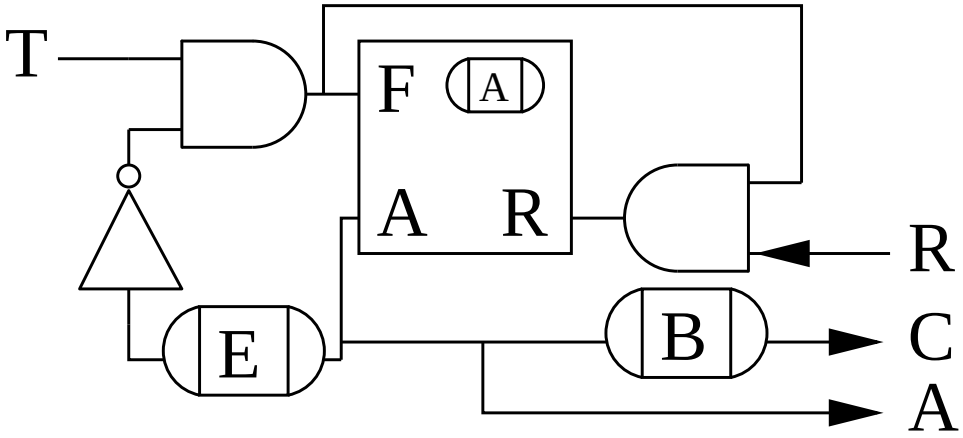

Figure 15: Passive Module Synchronizer (Early Termination)

4.5 Passive Communication

Registers controlled by different, non-concatenated active modules have local clocks which are unrelated in phase and frequency, and are thus asynchronous to each other. Transferring data between asynchronous local clock domains cannot be done directly since a receiving register may read from the output of a sending register while the latter is changing state, resulting in a metastable condition in the former. To reliably transfer data across domains, the sender must signal that there is data available and the receiver must acknowledge that it has received the data before the sender can change state again.

4.5.1 Synchronizer

Figure 14 shows how the passive synchronizer integrates in the local clock generation system from section 3.5. When triggered, the passive synchronizer waits for R to rise, after which it raises A to latch the received data[18] and make it visible to the internal registers. The Clock (C) rises shortly thereafter. The entire system remains in a stable high phase state until the environment drops R. The passive synchronizer then drops A to signal the end of the handshake, and C drops after a suitable propagation delay. Reversing the direction and the edge polarity of the register and following the same steps allows reliable writing to the environment.

4.5.2 Input/Output

If the interface with the environment cannot be made in a coordinated manner, the passive synchronizer can be converted into a plain input/output port by looping back A to R via an inverter[19], and if reading from the environment by extending the delay of a falling edge through B to allow for the register to resolve a potential metastable state before internal registers read its output[20].

4.5.3 Early Termination

The time between the receipt of data and the end of a handshake (edges 4 to 6 in figure 13) performs only a synchronization function since the data has already been latched. Figure 15 shows a modified synchronizer which gates off R when F falls in order to abridge this part of a handshake. The high phase then completes without waiting for the environment to release its request (R). The disadvantage of this method is that the system may initiate a new handshake while R is still high, which would prematurely begin the high phase before the environment had time to respond[21].

[18] The signal R actually rises a short time *after* the data is valid to provide margin against uneven transit time (skew) and to meet the setup time of the register [PvB95].

[19] This suggests that a passive version of the active module in figure 3 is possible, but this has not been explored yet.

[20] If the synchronizer will never be used as an input/output port, then the register can be omitted and the B delay reduced accordingly.

[21] It may also allow two passive synchronizers to communicate by cross-connecting their A and R lines via inverters, but this has not been explored yet.

Algorithm 3 Gullwing Finite State Machine

Inputs				Outputs f(ISR0,TOS,LAST,State)	
ISR0	TOS	LAST	State	Next	Control
PC@		X	0	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
JMP		X	0	1	MEM -> MAR, -> PC
JMP		X	1	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
JMP0	=0	X	0	1	NEXT -> TOP, DS(POP), MEM -> MAR, -> PC
JMP0	!=0	X	0	0	NEXT -> TOP, DS(POP), PC -> ALU(+1) -> PC, -> MAR, ISR>>
JMP0		X	1	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
JMP+	MSB(0)	X	0	1	NEXT -> TOP, DS(POP), MEM -> MAR, -> PC
JMP+	MSB(1)	X	0	0	NEXT -> TOP, DS(POP), PC -> ALU(+1) -> PC, -> MAR, ISR>>
JMP+		X	1	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
CALL		X	0	1	PC -> ALU(+1) -> R, RS(PUSH), MEM -> MAR, -> PC
CALL		X	1	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
RET		X	0	1	RS(POP), R -> MAR, -> PC
RET		X	1	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
LIT		X	0	0	MEM -> TOP, TOP -> NEXT, DS(PUSH), PC -> ALU(+1)->PC,->MAR, ISR>>
@A+		X	0	1	A -> MAR, -> ALU(+1) -> A
@A+		X	1	0	MEM -> TOP, TOP -> NEXT, DS(PUSH), PC -> MAR, ISR>>
@R+		X	0	1	R -> MAR, -> ALU(+1) -> R
@R+		X	1	0	MEM -> TOP, TOP -> NEXT, DS(PUSH), PC -> MAR, ISR>>
@A		X	0	1	A -> MAR
@A		X	1	0	MEM -> TOP, TOP -> NEXT, DS(PUSH), PC -> MAR, ISR>>
!A+		X	0	1	A -> MAR, -> ALU(+1) -> A
!A+		X	1	0	DS(POP), NEXT -> TOP, TOP -> MEM, PC -> MAR, ISR>>
!R+		X	0	1	R -> MAR, -> ALU(+1) -> R
!R+		X	1	0	DS(POP), NEXT -> TOP, TOP -> MEM, PC -> MAR, ISR>>
!A		X	0	1	A -> MAR
!A		X	1	0	DS(POP), NEXT -> TOP, TOP -> MEM, PC -> MAR, ISR>>
COM		0	0	0	TOP -> ALU(NOT) -> TOP, ISR>>
COM		1	0	0	TOP -> ALU(NOT) -> TOP, PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
AND		0	0	0	TOP, NEXT -> ALU(AND) -> TOP, DS(POP), ISR>>
AND		1	0	0	TOP,NEXT->ALU(AND)->TOP,DS(POP),PC->ALU(+1)->PC,->MAR,MEM->ISR
XOR		0	0	0	TOP, NEXT -> ALU(XOR) -> TOP, DS(POP), ISR>>
XOR		1	0	0	TOP,NEXT->ALU(XOR)->TOP,DS(POP),PC->ALU(+1)->PC,->MAR,MEM->ISR
+		0	0	0	(TOP, NEXT) -> ALU(+) -> TOP, DS(POP), ISR>>
+		1	0	0	(TOP, NEXT) -> ALU(+)->TOP,DS(POP),PC->ALU(+1)-> PC,->MAR,MEM->ISR
2*		0	0	0	TOP -> ALU(2*) -> TOP, ISR>>
2*		1	0	0	TOP -> ALU(2*)->TOP,PC->ALU(+1)->PC,->MAR,MEM->ISR
2/		0	0	0	TOP -> ALU(2/) -> TOP, ISR>>
2/		1	0	0	TOP -> ALU(2/)->TOP,PC->ALU(+1)->PC,->MAR,MEM->ISR
+*	LSB(0)	0	0	0	ISR>>
+*	LSB(0)	1	0	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
+*	LSB(1)	0	0	0	(NEXT, TOP) -> ALU(+) -> TOP, ISR>>
+*	LSB(1)	1	0	0	(NEXT, TOP) -> ALU(+) -> TOP, PC -> ALU(+1) -> PC, -> MAR,MEM->ISR
A>		0	0	0	A -> TOP, TOP -> NEXT, DS(PUSH), ISR>>
A>		1	0	0	A -> TOP, TOP -> NEXT, DS(PUSH), PC ->ALU(+1)->PC,->MAR,MEM->ISR
>A		0	0	0	TOP -> A, NEXT -> TOP, DS(POP), ISR>>
>A		1	0	0	TOP -> A, NEXT -> TOP, DS(POP), PC -> ALU(+1)->PC,->MAR,MEM->ISR
DUP		0	0	0	TOP -> NEXT, DS(PUSH), ISR>>
DUP		1	0	0	TOP -> NEXT, DS(PUSH), PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
DROP		0	0	0	NEXT -> TOP, DS(POP), ISR>>
DROP		1	0	0	NEXT -> TOP, DS(POP), PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
OVER		0	0	0	NEXT -> TOP, TOP -> NEXT, DS(PUSH), ISR>>
OVER		1	0	0	NEXT -> TOP, TOP -> NEXT, DS(PUSH),PC->ALU(+1)->PC,->MAR,MEM->ISR
R>		0	0	0	RS(POP), R -> TOP, TOP -> NEXT, DS(PUSH), ISR>>
R>		1	0	0	RS(POP),R->TOP,TOP->NEXT,DS(PUSH),PC->ALU(+1)->PC,->MAR,MEM->ISR
>R		0	0	0	DS(POP), NEXT -> TOP, TOP -> R, RS(PUSH), ISR>>
>R		1	0	0	DS(POP), NEXT -> TOP, TOP -> R, RS(PUSH),PC->ALU(+1)->PC,->MAR,MEM->ISR
NOP		0	0	0	ISR>>
NOP		1	0	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
UNDEF0		0	0	0	ISR>>
UNDEF0		1	0	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
UNDEF1		0	0	0	ISR>>
UNDEF1		1	0	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
UNDEF2		0	0	0	ISR>>
UNDEF2		1	0	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR
UNDEF3		0	0	0	ISR>>
UNDEF3		1	0	0	PC -> ALU(+1) -> PC, -> MAR, MEM -> ISR

5 Gullwing Implementation

5.1 Finite State Machine Table

Algorithm 3 [LaF05] describes the Mealy finite state machine (FSM) which controls the Gullwing processor. The Instruction Shift Register (ISR) shifts in a PC-Fetch (PC@) instruction as instructions are shifted out, denoted by the shorthand `ISR>>`. The signal `ISR0` refers to the location in the ISR containing the current instruction. The signal `LAST` is set when the next instruction to be executed is `PC@`. An `X` denotes a don't care condition where the value of `LAST` is irrelevant. The entries under Top-Of-Stack (`TOS`) are combinational functions of the contents of the top of the data stack respectively describing the all-zero, positive/negative, and even/odd conditions. At each step, `State` takes on the value of `Next`.

5.2 Implementation

The implementation of the Gullwing architecture is easily described using the locally clocked method presented in section 3.5. The control lines (L) are the signals listed under `Inputs` in algorithm 3. The local clock (C) is controlled by the instruction decoder (I) which selects a period of appropriate duration for the kind of instruction to execute. The register-to-register transfers are controlled by each register's clock gating (G) and source select (S) decoders.

The state of the FSM is kept in a similarly controlled register which toggles itself as needed (figure 16).

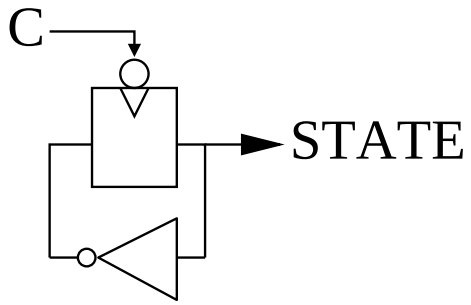

Figure 16: State Register

The ISR is also controlled in the same manner, but is a compound system which either loads instructions from memory (MEM) or shifts them out (ISR0) and generates the LAST signal (figure 17).

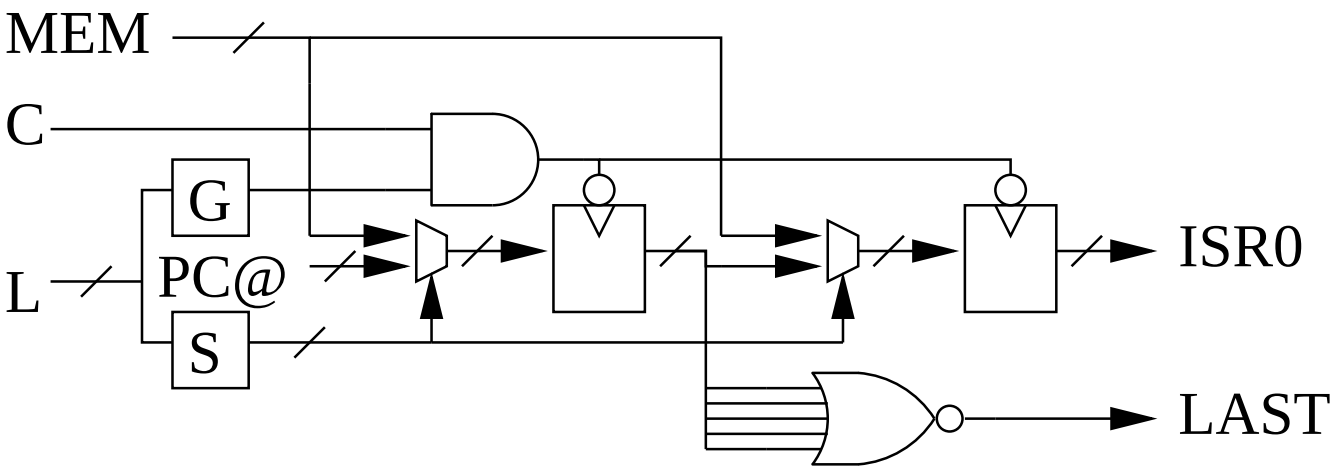

Figure 17: Instruction Shift Register

A VHDL Code

Here are VHDL simulations of the element syntheses and module designs. The delays are for demonstration only.

A.1 Delay Element

This code describes a universal transport delay element.

```
-- VHDL simulation of asymmetric delay element
library IEEE;
use ieee.std_logic_1164.all;
entity delay is
        generic (
        re_delay : time := 0 ns; -- rising edge
        fe_delay : time := 0 ns  -- falling edge
        );
        port (
        input  : in std_logic := '0';
        output  : out std_logic := '0'
        );
end delay;
architecture main of delay is
begin
        process begin
                wait until rising_edge(input);
                output <= transport input after re_delay;
                wait until falling_edge(input);
                output <= transport input after fe_delay;
        end process;
end main;
```

A.2 Active

A.2.1 Element

```
-- VHDL simulation of Burst-Mode active 4-phase handshake element
library IEEE;
use ieee.std_logic_1164.all;
entity active is
        generic (
        state_delay : time := 0 ns;
        output_delay : time := 0 ns
        );
        port (
        F  : in std_logic := '0';
        A  : in std_logic := '0';
        R  : out std_logic := '0'
        );
end active;
architecture main of active is
        -- Internal state
        signal Y0 : std_logic := '0';
begin
        -- AND + OR
        Y0 <= transport A OR (F AND Y0) after state_delay;
        -- NOT + AND
        R  <= transport F AND (NOT A) AND (NOT Y0) after output_delay;
end main;
```

A.2.2 Module

```
-- VHDL testbench for active.vhd
library IEEE;
use ieee.std_logic_1164.all;
entity active_tb is
end active_tb;
architecture main of active_tb is
        signal fun_in : std_logic := '0';
        signal fun_en : std_logic := '0';
        signal enable_in : std_logic := '0';
        signal enable_out : std_logic := '0';
        signal feedback_in : std_logic := '0';
        signal feedback_out : std_logic := '0';
        signal fun : std_logic := '0';
        signal req : std_logic := '0';
        signal ack : std_logic := '0';
begin
        fsm : entity work.active
        generic map (
                state_delay => 2 ns,
                output_delay => 2 ns
        )
        port map (
                F => fun,
                A => ack,
                R => req
        );
        enable : entity work.delay
        generic map (
                re_delay => 1 ns,
                fe_delay => 1 ns
        )
        port map (
                input => enable_in,
                output => enable_out
        );
        feedback : entity work.delay
        generic map (
                 re_delay => 1 ns,
                 fe_delay => 1 ns
        )
        port map (
                input => feedback_in,
                output => feedback_out
        );
        feedback_in <= req;
        ack <= feedback_out;
        enable_in <= feedback_out;
        fun_en <= enable_out;
        fun <= transport fun_in AND (NOT fun_en) after 1 ns;

        process begin
                wait for 10 ns;
                fun_in <= '1';
        end process;
end main;
```

A.3 Passive

A.3.1 Element

```
-- VHDL simulation of Burst-Mode passive handshake (Muller C?) element
library IEEE;
use ieee.std_logic_1164.all;
entity passive is
        generic (
        delay : time := 0 ns
        );
        port (
        F  : in std_logic := '0';
        R  : in std_logic := '0';
        A  : out std_logic := '0'
        );
end passive;
architecture main of passive is
        signal A_i : std_logic := '0';
begin
        A_i <= transport (F AND R) OR (F AND A_i) OR (R AND A_i) after delay;
        A <= A_i;
end main;
```

A.3.2 Module

```
-- VHDL testbench for passive.vhd
library IEEE;
use ieee.std_logic_1164.all;
entity passive_tb is
end passive_tb;
architecture main of passive_tb is
        signal fun_in : std_logic := '0';
        signal fun_en : std_logic := '0';
        signal enable_in : std_logic := '0';
        signal enable_out : std_logic := '0';
        signal feedback_in : std_logic := '0';
        signal feedback_out : std_logic := '0';
        signal fun : std_logic := '0';
        signal req : std_logic := '0';
        signal ack : std_logic := '0';
begin
        fsm : entity work.passive
        generic map (
                delay => 8 ns
        )
        port map (
                F => fun,
                R => req,
                A => ack
        );
        enable : entity work.delay
        generic map (
                re_delay => 9 ns,
                fe_delay => 20 ns
        )
        port map (
                input => enable_in,
                output => enable_out
        );
        feedback : entity work.delay
        generic map (
                re_delay => 5 ns,
                fe_delay => 2 ns
        )
        port map (
                input => feedback_in,
                output => feedback_out
        );
        -- Nota Bene
        feedback_in <= NOT ack;
        req <= feedback_out;
        enable_in <= ack;
        fun_en <= enable_out;
        fun <= fun_in AND (NOT fun_en);

        process begin
                wait for 30 ns;
                fun_in <= '1';
        end process;
end main;
```

A.4 Active-Passive Communication

A.4.1 Standard

```
-- VHDL testbench for active-passive communication
library IEEE;
use ieee.std_logic_1164.all;
entity ap_tb is
end ap_tb;
architecture main of ap_tb is
        -- Function input and enabled input
        signal a_fun_in : std_logic := '0';
        signal a_fun_en : std_logic := '0';
        signal p_fun_in : std_logic := '0';
        signal p_fun_en : std_logic := '0';
        -- Enable delays
        signal a_enable_in : std_logic := '0';
        signal a_enable_out : std_logic := '0';
        signal p_enable_in : std_logic := '0';
        signal p_enable_out : std_logic := '0';
        -- Communication lines transport delays
        signal req_req_in : std_logic := '0';
        signal req_req_out : std_logic := '0';
        signal ack_ack_in : std_logic := '0';
        signal ack_ack_out : std_logic := '0';
        -- Delayed generated clocks
        signal a_clk : std_logic := '0';
        signal p_clk : std_logic := '0';
        signal a_clk_in : std_logic := '0';
        signal a_clk_out : std_logic := '0';
        signal p_clk_in : std_logic := '0';
        signal p_clk_out : std_logic := '0';
        -- FSM signals
        signal a_fun : std_logic := '0';
        signal a_req : std_logic := '0';
        signal a_ack : std_logic := '0';
        signal p_fun : std_logic := '0';
        signal p_req : std_logic := '0';
        signal p_ack : std_logic := '0';
begin
        a_fsm : entity work.active
        generic map (
                state_delay => 8 ns,
                output_delay => 6 ns
        )
        port map (
                F => a_fun,
                A => a_ack,
                R => a_req
        );
        p_fsm : entity work.passive
        generic map (
                delay => 6 ns
        )
        port map (
                F => p_fun,
                R => p_req,
                A => p_ack
```

```
);
a_enable : entity work.delay
generic map (
        re_delay => 10 ns,
        fe_delay => 10 ns
)
port map (
        input => a_enable_in,
        output => a_enable_out
);
p_enable : entity work.delay
generic map (
        re_delay => 10 ns,
        fe_delay => 10 ns
)
port map (
        input => p_enable_in,
        output => p_enable_out
);
a_clk_delay : entity work.delay
generic map (
        re_delay => 2 ns,
        fe_delay => 2 ns
)
port map (
        input => a_clk_in,
        output => a_clk_out
);
p_clk_delay : entity work.delay
generic map (
        re_delay => 2 ns,
        fe_delay => 2 ns
)
port map (
        input => p_clk_in,
        output => p_clk_out
);
req_req : entity work.delay
generic map (
        re_delay => 1 ns,
        fe_delay => 1 ns
)
port map (
        input => req_req_in,
        output => req_req_out
);
ack_ack : entity work.delay
generic map (
        re_delay => 1 ns,
        fe_delay => 1 ns
)
port map (
        input => ack_ack_in,
        output => ack_ack_out
);
req_req_in <= a_req;
p_req <= req_req_out;
```

```
        ack_ack_in <= p_ack;
        a_ack <= ack_ack_out;
        a_enable_in <= a_ack;
        a_fun_en <= a_enable_out;
        p_enable_in <= p_ack;
        p_fun_en <= p_enable_out;
        a_clk_in <= a_req;
        a_clk <= a_clk_out;
        p_clk_in <= p_ack;
        p_clk <= p_clk_out;
        a_fun <= a_fun_in AND (NOT a_fun_en);
        p_fun <= p_fun_in AND (NOT p_fun_en);

        process begin
                wait for 10 ns;
                a_fun_in <= '1';
                wait for 20 ns;
                p_fun_in <= '1';
        end process;
end main;
```

A.4.2 Early Termination

```
-- VHDL testbench for active-passive fast communication
library IEEE;
use ieee.std_logic_1164.all;
entity ap_fast_tb is
end ap_fast_tb;
architecture main of ap_fast_tb is
        -- Function input and enabled input
        signal a_fun_in : std_logic := '0';
        signal a_fun_en : std_logic := '0';
        signal p_fun_in : std_logic := '0';
        signal p_fun_en : std_logic := '0';
        -- Enable delays
        signal a_enable_in : std_logic := '0';
        signal a_enable_out : std_logic := '0';
        signal p_enable_in : std_logic := '0';
        signal p_enable_out : std_logic := '0';
        -- Communication lines transport delays
        signal req_req_in : std_logic := '0';
        signal req_req_out : std_logic := '0';
        signal ack_ack_in : std_logic := '0';
        signal ack_ack_out : std_logic := '0';
        -- Delayed generated clocks
        signal a_clk : std_logic := '0';
        signal p_clk : std_logic := '0';
        signal a_clk_in : std_logic := '0';
        signal a_clk_out : std_logic := '0';
        signal p_clk_in : std_logic := '0';
        signal p_clk_out : std_logic := '0';
        -- FSM signals
        signal a_fun : std_logic := '0';
        signal a_req : std_logic := '0';
        signal a_ack : std_logic := '0';
        signal p_fun : std_logic := '0';
        signal p_req : std_logic := '0';
```

```
        signal p_ack : std_logic := '0';
begin
        a_fsm : entity work.active
        generic map (
                state_delay => 5 ns,
                output_delay => 5 ns
        )
        port map (
                F => a_fun,
                A => a_ack,
                R => a_req
        );
        p_fsm : entity work.passive
        generic map (
                delay => 5 ns
        )
        port map (
                F => p_fun,
                R => p_req,
                A => p_ack
        );
        a_enable : entity work.delay
        generic map (
                re_delay => 5 ns,
                fe_delay => 5 ns
        )
        port map (
                input => a_enable_in,
                output => a_enable_out
        );
        p_enable : entity work.delay
        generic map (
                re_delay => 5 ns,
                fe_delay => 5 ns
        )
        port map (
                input => p_enable_in,
                output => p_enable_out
        );
        a_clk_delay : entity work.delay
        generic map (
                re_delay => 2 ns,
                fe_delay => 2 ns
        )
        port map (
                input => a_clk_in,
                output => a_clk_out
        );
        p_clk_delay : entity work.delay
        generic map (
                re_delay => 2 ns,
                fe_delay => 2 ns
        )
        port map (
                input => p_clk_in,
                output => p_clk_out
        );
```

```
        req_req : entity work.delay
        generic map (
                re_delay => 1 ns,
                fe_delay => 1 ns
        )
        port map (
                input => req_req_in,
                output => req_req_out
        );
        ack_ack : entity work.delay
        generic map (
                re_delay => 1 ns,
                fe_delay => 1 ns
        )
        port map (
                input => ack_ack_in,
                output => ack_ack_out
        );
        -- Nota Bene
        req_req_in <= a_req;
        p_req <= req_req_out AND p_fun;
        ack_ack_in <= p_ack;
        a_ack <= ack_ack_out AND a_req;
        a_enable_in <= a_ack;
        a_fun_en <= a_enable_out;
        p_enable_in <= p_ack;
        p_fun_en <= p_enable_out;
        a_clk_in <= a_req;
        a_clk <= a_clk_out;
        p_clk_in <= p_ack;
        p_clk <= p_clk_out;
        a_fun <= a_fun_in AND (NOT a_fun_en);
        p_fun <= p_fun_in AND (NOT p_fun_en);

        process begin
                wait for 10 ns;
                a_fun_in <= '1';
                wait for 10 ns;
                p_fun_in <= '1';
                wait for 10 ns;
                a_fun_in <= '0';
                wait for 10 ns;
                p_fun_in <= '0';
        end process;
end main;
```

A.5 Active-Active Communication

```
-- VHDL testbench for active-active fast communication
library IEEE;
use ieee.std_logic_1164.all;
entity aa_fast_tb is
end aa_fast_tb;
architecture main of aa_fast_tb is
        -- Function input and enabled input
        signal a1_fun_in : std_logic := '0';
        signal a1_fun_en : std_logic := '0';
        signal a2_fun_in : std_logic := '0';
        signal a2_fun_en : std_logic := '0';
        -- Enable delays
        signal a1_enable_in : std_logic := '0';
        signal a1_enable_out : std_logic := '0';
        signal a2_enable_in : std_logic := '0';
        signal a2_enable_out : std_logic := '0';
        -- Communication lines transport delays
        signal req_ack1_in : std_logic := '0';
        signal req_ack1_out : std_logic := '0';
        signal req_ack2_in : std_logic := '0';
        signal req_ack2_out : std_logic := '0';
        -- Delayed generated clocks
        signal a1_clk : std_logic := '0';
        signal a2_clk : std_logic := '0';
        signal a1_clk_in : std_logic := '0';
        signal a1_clk_out : std_logic := '0';
        signal a2_clk_in : std_logic := '0';
        signal a2_clk_out : std_logic := '0';
        -- FSM signals
        signal a1_fun : std_logic := '0';
        signal a1_req : std_logic := '0';
        signal a1_ack : std_logic := '0';
        signal a2_fun : std_logic := '0';
        signal a2_req : std_logic := '0';
        signal a2_ack : std_logic := '0';
begin
        a1_fsm : entity work.active
        generic map (
                state_delay => 5 ns,
                output_delay => 5 ns
        )
        port map (
                F => a1_fun,
                A => a1_ack,
                R => a1_req
        );
        a2_fsm : entity work.active
        generic map (
                state_delay => 5 ns,
                output_delay => 5 ns
        )
        port map (
                F => a2_fun,
                A => a2_ack,
                R => a2_req
```

```
);
a1_enable : entity work.delay
generic map (
        re_delay => 5 ns,
        fe_delay => 5 ns
)
port map (
        input => a1_enable_in,
        output => a1_enable_out
);
a2_enable : entity work.delay
generic map (
        re_delay => 5 ns,
        fe_delay => 5 ns
)
port map (
        input => a2_enable_in,
        output => a2_enable_out
);
a1_clk_delay : entity work.delay
generic map (
        re_delay => 2 ns,
        fe_delay => 2 ns
)
port map (
        input => a1_clk_in,
        output => a1_clk_out
);
a2_clk_delay : entity work.delay
generic map (
        re_delay => 2 ns,
        fe_delay => 2 ns
)
port map (
        input => a2_clk_in,
        output => a2_clk_out
);
req_ack1 : entity work.delay
generic map (
        re_delay => 1 ns,
        fe_delay => 1 ns
)
port map (
        input => req_ack1_in,
        output => req_ack1_out
);
req_ack2 : entity work.delay
generic map (
        re_delay => 1 ns,
        fe_delay => 1 ns
)
port map (
        input => req_ack2_in,
        output => req_ack2_out
);
-- Nota Bene
req_ack1_in <= a1_req;
```

```
        a2_ack <= req_ack1_out AND a2_req;
        req_ack2_in <= a2_req;
        a1_ack <= req_ack2_out AND a1_req;
        a1_enable_in <= a1_ack;
        a1_fun_en <= a1_enable_out;
        a2_enable_in <= a2_ack;
        a2_fun_en <= a2_enable_out;
        a1_clk_in <= a1_req;
        a1_clk <= a1_clk_out;
        a2_clk_in <= a2_ack;
        a2_clk <= a2_clk_out;
        a1_fun <= a1_fun_in AND (NOT a1_fun_en);
        a2_fun <= a2_fun_in AND (NOT a2_fun_en);

        process begin
                wait for 10 ns;
                a1_fun_in <= '1';
                wait for 10 ns;
                a2_fun_in <= '1';
                wait for 10 ns;
                a1_fun_in <= '0';
                wait for 10 ns;
                a2_fun_in <= '0';
        end process;
end main;
```

References

[BNY99] Erik Brunvand, Steven Nowick, and Kenneth Yun, *Practical advances in asynchronous design and in asynchronous/synchronous interfaces*, DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation (New York, NY, USA), ACM Press, 1999, An overview of the field. Covers control, datapaths, and processors, pp. 104–109.

[BS91] Janusz A. Brzozowski and Carl-Johan H. Seger, *Asynchronous circuits*, Springer-Verlag, New York, NY, USA, 1991, Section 15.4 briefly covers Burst-Mode circuits.

[FN01] Robert M. Fuhrer and Steven Nowick, *Sequential optimization of asynchronous and synchronous finite-state machines: Algorithms and tools*, Kluwer Academic Publishers, Norwell, MA, USA, 2001, The primary reference for Burst-Mode synthesis using the MINIMALIST CAD tool.

[HS02] Soha Hassoun and Tsutomu Sasao (eds.), *Logic synthesis and verification*, Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[LaF05] Eric LaForest, *Unit 2-3 (IS 102B) report: The Gullwing stack computer architecture*, IS Unit report, University of Waterloo, Independent Studies Program, April 2005, Otherwise unpublished.

[LN02] Luciano Lavagno and Steven M. Nowick, *Asynchronous control circuits*, ch. 10, pp. 255–284, in Hassoun and Sasao [HS02], 2002, Introduces Burst-Mode circuits.

[MC79] Carver Mead and Lynn Conway, *Introduction to VLSI systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979, The original and authoritative book on VLSI circuitry.

[Moo01] Charles H. Moore, *c18 colorForth compiler*, Proceedings of the 17th EuroForth Conference (Schloss Dagstuhl, Saarland, Germany), University of Teesside, November 2001, One of the few published papers by Chuck Moore. Describes the c18 instruction set in detail.

[Mye01] Chris J. Myers, *Asynchronous circuit design*, Wiley-Interscience, 2001, Read as an introduction to Petri Nets and communication protocols.

[ND91] Steven M. Nowick and David L. Dill, *Automatic synthesis of locally-clocked asynchronous state machines*, ICCAD '91: Proceedings of the 1991 IEEE International Conference on Computer-Aided Design (Washington, DC, USA), IEEE Computer Society, Nov 1991, Covers the rules for Burst-Mode specifications. The state machine implementation method is obsolete though., pp. 318–321.

[NYD92] Steven M. Nowick, Kenneth Y. Yun, and David L. Dill, *Practical asynchronous controller design*, ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors (Washington, DC, USA), IEEE Computer Society, 1992, Shows practical DRAM and SCSI controller designs using Burst-Mode circuits. The state machine implementation method is obsolete though., pp. 341–345.

[PH04] David A. Patterson and John Hennessy, *Computer organization and design*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004, The primary undergraduate textbook on the topic.

[PvB95] A. Peeters and K. van Berkel, *Single-rail handshake circuits*, ASYNC '95: Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies (Washington, DC, USA), IEEE Computer Society, 1995, Contains an excellent discussion of four-phase handshakes.

[Sei79] Charles L. Seitz, *System timing*, ch. 7, in [MC79], 1979, Fundamental reference. Covers asymmetric delays, controlled clocks, and asynchronous communication.

[TLE96] Nozar Tabrizi, Michael J. Liebelt, and Kamran Eshraghian, *Dynamic hazards and speed independent delay model*, ASYNC '96: Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems (Washington, DC, USA), IEEE Computer Society, 1996, Contains an excellent introduction to the various kinds of hazards found in logic circuits., p. 94.

[vBJN99] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven Nowick, *Applications of asynchronous circuits*, Proceedings of the IEEE (Washington, DC, USA), vol. 87, IEEE Computer Society, Feb 1999, An overview of the useful properties of asynchronous circuits., pp. 223–233.

[YDN93] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick, *Practical generalizations of asynchronous state machines*, Proceedings of the [4th] European Conference on Design Automation, 1993, with the European Event in ASIC Design (Washington, DC, USA), IEEE Computer Society, Feb 1993, Introduces Extended Burst-Mode synthesis., pp. 525–530.