Norwegian University of
Science and Technology

Application of asynchronous design to microcontroller startup logic

Svein Rypdal Henninen

Master of Science in Electronics
Submission date: August 2011
Supervisor: Bjørn B. Larsen, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

A particular challenge in microcontroller system is designing the logic involved in the startup sequence, i.e. when power is initially applied to the system. Advanced microcontrollers rely on a complicated interaction between several digital and analog modules, often outside of the regular operating range of the system.

Analog modules are inherently asynchronous and using asynchronous logic to control them could have several advantages:

- No oscillator or clock system required
- Operation in near-threshold/sub-threshold region is possible, while traditional synchronous operation breaks down
- Direct integration with analog cells that support built-in handshaking capability (enable/ready signals)

The student should:

- Implement a synchronous Verilog RTL model of the startup logic to be used as a reference for further work
- Study literature and available tools (e.g. Teak/Balsa) and propose an equivalent implementation using asynchronous techniques, such as 4-phase dual-rail.
- Propose a protocol for analog cell interfaces to make the cells usable in an asynchronous design
- Synthesize both the synchronous and asynchronous implementations and simulate them with Verilog and Spice simulators

A secondary objective is to extend the startup logic with ultra-low power features, such as real-time clock, voltage monitoring, or external event detection.

Abstract

Digital circuits designed today are almost exclusively clocked. As designs grow in size it becomes harder to effectively distribute the various clock signals over the circuit. The clock is also a big contribution to the power consumption of a circuit. Some work is being done to provide alternatives to standard synchronous design. One of these alternatives is the Balsa system.

Several versions of an asynchronous module for controlling the startup process of a microcontroller was made in Balsa and compared to a standard synchronous implementation. Area estimates for the best asynchronous implementation gives a number that is a factor of over four larger than for the synchronous implementation. The asynchronous implementation has other advantages though. It has no dynamic power consumption when it is in a stable state. Additionally it can operate closer to the sub-threshold area.

The asynchronous implementations have been tested and found working in active HDL. Balsa generated verilog netlists in a 350 nm library from the balsa language description. Design Compiler from Synopsys was used to get the area estimates. The asynchronous implementations shows potential, especially with regards to reduced power consumption.

Contents

Nomenclature **xi**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**

2 Theory **3**

2.1 Introduction to Asynchronous 3

2.1.1 Potential drawbacks and gain 4

2.1.2 Protocols 6

2.1.3 Special cells 12

2.2 Power consumption 14

2.2.1 Dynamic power dissipation 14

2.2.2 Short circuit power dissipation 15

2.2.3 Leakage power dissipation 15

2.3 Metastability 16

3 Balsa **17**

3.1 Introduction to balsa 17

3.2 Balsa system 18

3.3 Balsa language 20

3.3.1 Language constructs 20

4 Startup logic specification **23**

4.1 Startup system 23

4.2 Analog Cells 24

4.2.1 Power-on-Reset (POR) generator 24

4.2.2 Voltage Monitor 24

4.2.3 Voltage Regulator 26

4.2.4 Voltage Reference 27

4.3 Startup logic Controller 28

4.4 Asynchronous specification 29

5 Design **31**

5.1 Synchronous implementation 31

5.1.1 Design Considerations 31

5.1.2 Implementation 32

5.2 Asynchronous Implementation 35
5.2.1 Description 35
5.2.2 Design Considerations 35
5.2.3 State machine implementation 36
5.2.4 Optimized implementation 39
5.3 Interface 44
5.3.1 Choice of interface 44
5.3.2 Sync interface 45
5.3.3 Passive PUSH Interface 47
5.3.4 Passive PULL Interface 48
5.3.5 Active PUSH Interface 49
6 Evaluation and results **51**
6.1 Testing 51
6.1.1 Test cases 51
6.1.2 Testing of the synchronous implementation 52
6.1.3 Testing of the asynchronous implementations 53
6.2 Synthesis 56
6.3 Discussion 58
7 Conclusion **61**

A Balsa Code **65**

A.1 State machine implementations 65

A.1.1 State machine implementation 1 65

A.1.2 State machine implementation 2 68

A.1.3 State machine implementation 3 71

A.2 Optimized implementations 74

A.2.1 Optimized implementation 1 74

A.2.2 Optimized implementation 2 76

A.2.3 Optimized implementation 3 78

B Synchronous startup logic **81**

Nomenclature

2PSR	2-phase single-rail
4PSR	4-phase single-rail
2PDR	2-phase dual-rail
4PDR	4-phase dual-rail
EDA	Electronic Design Automation
HDL	Hardware Description Language
LEDR	Level-Encoded 2-phase Dual-Rail
LSB	Least Significant Bit(s)
MSB	Most Significant Bit(s)
NCL	Null Convention Logic
RTZ	Return-To-Zero
VLSI	Very-Large-Scale-Integration
DUT	Device under test

List of Figures

2.1 Comparison between synchronous and asynchronous pipelines 4

2.2 Single-rail scheme 7

2.3 4-phase Single-rail 8

2.4 2-phase Single-rail 9

2.5 Dual-rail scheme 10

2.6 communication over 4-phase dualrail protocol 10

2.7 2-phase dual-rail transaction 11

2.8 Muller C-element 13

3.1 balsa design flow 19

4.1 Startup system with controller and analog cells 23

5.1 Two stage synchronizer 32

5.2 Simplified state Diagram of the Synchronous implementation . 33

5.3 Sync interface 45

5.4 Passive PUSH interface 47

5.5 Passive PULL interface 48

5.6 Modified passive PULL interface 49

5.7 Active PUSH interface 50

6.1 Waveform from the test of the synchronous implementation . . 53

6.2 Simulation output in execution window 54

6.3 Waveform from the test of one of the asynchronous implementations 55

List of Tables

2.1 Truth table for a Muller C-element 13

3.1 Balsa commands 21

4.1 Pin Description for the POR 24

4.2 Parameter list for POR 24

4.3 Pin Description for a voltage monitor 25

4.4 Parameter list for a voltage monitor 25

4.5 Pin Description for a voltage regulator 26

4.6 Parameter list for voltage regulator 26

4.7 Selvdd table for a voltage regulator 27

4.8 Pin Description for a voltage reference 27

4.9 Parameter list for a voltage reference 28

5.1 Signals used to change the state 34

6.1 Area estimates for startup logic 57

6.2 Area estimates for the interfaces 57

Chapter 1

Introduction

In this thesis the design and implementation of clock-less, or asynchronous, digital circuits is explored. More specifically, the circuit being made is a module controlling the startup process of a microcontroller. The unit controls several analog cell used in the startup process to provide a correct power up sequence for the microcontroller. The asynchronous startup logic is compared to a synchronous version to better see the benefits and drawbacks of the asynchronous implementation. To design the circuit the Balsa system is used, since ordinary hardware descriptive languages are geared towards synchronous design.

Chapter 2 goes through the basics of asynchronous circuits to give a better understanding of some of the concepts presented later in the thesis. This includes differences between asynchronous and synchronous systems, advantages and disadvantages, protocols and signaling schemes and some special cells. It also gives an overview of power consumption and metastability.

The balsa system and hardware descriptive language (HDL) is presented in chapter 3, to provide the base knowledge necessary to understand the balsa code presented later in this thesis.

Chapter 4 presents the specifications given for the startup system. This includes specifications for the various analog cells and behavioral specifications given for the startup logic controller.

The design process for the startup logic controller is described in chapter 5.

A synchronous design is first presented, and then the various asynchronous implementations. The interfaces used to enable communication between the asynchronous implementations and the analog cells, are presented last.

In chapter 6 the test and synthesis process is presented and the results are discussed. Comparison is made between the synchronous implementation and the asynchronous ones.

Chapter 7 presents the conclusions drawn from the work in this thesis.

The balsa code description of the startup circuit is included in the appendix A and the verilog code for the synchronous startup logic is included in appendix B. The generated verilog code has not been included as it is implemented with a Atmel technology, and therefor not testable without the technology package.

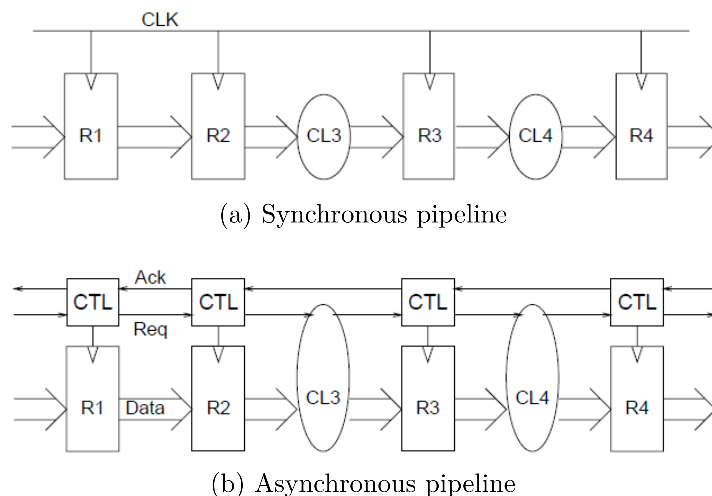
Chapter 2

Theory

2.1 Introduction to Asynchronous

Almost all digital circuits made today are almost without exception synchronous circuits. This is mainly because of the many tools which make design, simulation, verification and testing of large and complex circuits a relatively easy task. Additionally, the fact that all signals are binary and that all components share a common clock which defines a discrete time when all signals should be valid [1]. The discrete time lets designers focus their attention on functionality in stead of timing. Asynchronous circuits also require binary signals, but the components do not share a common and discrete time. Instead they use handshaking to sequence operations, communicate and synchronize events. This makes timing a bigger issue, but also gives asynchronous circuits other potential benefits.

Figure 2.1 illustrates the difference between asynchronous and synchronous circuits. In the synchronous version the clock drives the communication, and in the asynchronous version the communication is controlled by handshakes between components. The CTL blocks in 2.1b signify control logic blocks that communicate with each other through handshake signals. The control signals have to be delayed to make sure that the data passing through the combinational logic, CL, is valid before being stored in the registers. In the synchronous pipeline, the clock switches at a frequency that guarantees that data through the combinational logic is valid before it is stored in the regis-



(a) Synchronous pipeline

(b) Asynchronous pipeline

Figure 2.1: Comparison between synchronous and asynchronous pipelines

ters.

2.1.1 Potential drawbacks and gain

Asynchronous circuits have certain benefits that make them attractive, like lower power consumption and robustness to variations in power supply and temperature, but they are also associated with certain problems. These problems include area overhead due to handshaking circuitry and the lack of electronic design automation (EDA) tools. A selected set of drawbacks are discussed below:

Lack of tools:
There are few EDA tools available for a designer who wishes to design asynchronous circuits, and many of the successful circuits which have been designed so far have more or less been designed manually. The few tools that are available lack the maturity needed to make designers confident about using them and their use widespread.

Unfamiliarity:
Few electronic engineers are offered courses in asynchronous design, and are for the most part only familiar with synchronous design. Companies are as a result less willing to start asynchronous projects since it involves extra training for the designers involved.

Area overhead:
Because of the extra handshaking circuitry involved in asynchronous circuits, there may in some cases be a significant area overhead compared to a synchronous version. Every circuit component needs circuitry to control request and acknowledge signals, and this overhead increases not only the area, but can also lead to extra switching. This can result in reduced speed and increased power consumption compared.

Complex designs:
The overhead does not only decrease quality compared to the ideal case; it could also lead to more complex designs. The added complexity, in addition to the lack of mature tools can make asynchronous design even harder compared to synchronous design.

Testing and debugging:
Scannable flip-flops and automated test-pattern generation has made testing of synchronous circuits efficient and reliable. Many asynchronous design styles introduce loops which make observability, controllability and test-pattern generation challenging. Debugging failures can also be difficult because of the lack of a clock. Asynchronous circuits operate at maximum possible speed at all times, and the clock frequency can not be lowered to potentially reveal the location of errors.

Event though there are many problems associated with asynchronous circuits there are some potential benefits. For some applications these benefits may outweigh the drawbacks, making an asynchronous implementation beneficial. The potential benefits of asynchronous design are presented below.

Lower power consumption:
The activity of the global clock in a synchronous circuit makes the circuit consume power even when it is not processing data. Some synchronous designs reduce this problem with clock gating, but the clock driver still has to constantly provide a powerful clock to always be able to reach all parts of the circuit. In asynchronous circuits, the switching activity is local and parts of the circuit not involved in the current operation are inactive.

Less electromagnetic noise:
A global clock makes sure that most transitions in the circuit are triggered by the positive edge of the clock and come in intervals determined by the clock frequency. This results in peak currents that are much higher than the average power, causing electromagnetic noise. In asynchronous systems,

the transitions are spread out more evenly over time, mostly consuming an average amount of power. This greatly reduces the noise.

No clock skew:
Clock skew refers to the clock arriving at different times to different parts of the circuit. Because of clock skew, the clock speed sometimes has to be reduced to ensure correct operation. Because of the local handshaking, this is not an issue in asynchronous design.

Robustness:
Asynchronous systems have no timing constraints, and can in theory wait for a logic value to settle before continuing with the operation, thus avoiding metastability. Asynchronous systems, in addition, are robust against physical variations like temperature, supply voltage and fabrication parameters. This is because the critical paths can use as long as they have to before completing, without causing trouble for the rest of the circuit. If temperature or supply voltage variations increase delays in the circuit, the operation will be slower, but it will not halt.

Speed:
The critical path in synchronous designs determines the clock speed of the entire circuit, and thus limiting the circuits overall performance by reducing the speed of all paths. Because there is no clock limiting potential speed, asynchronous systems will over time operate at average-case performance.

For more in depth information about the drawbacks and benefits of asynchronous design [1] and [2] is recomended.

2.1.2 Protocols

Communication between different components in an asynchronous system happens over a channel in the form of handshakes. These handshakes consists of some sort of request and acknowledge signals in addition to eventual data being transfered. The handshake has an active part which issues the request and a passive part which responds with an acknowledge. If data is being transfered the active side may either send or request data, also known as PUSHing and PULLing. A channel where data is being actively sent is called a PUSH channel, while a channel where the data is requested by the active side is called a PULL channel. It is called PUSHing because the active side

can be described as “pushing” data to the passive side. Similarly, The other kind of transaction would be a PULL, as the data could be seen as “pulled” from the passive side to the active side. Dataless channels also exist and is mostly used for synchronization between different parts of the circuit. These channels are known as sync channels.

The way components communicate with each other in an asynchronous circuit is defined in a protocol. Several different protocols for asynchronous communication have been proposed and developed. The two most common signaling schemes, single-rail and dual-rail, with their variations of 2- and 4-phase protocols are discussed here in some detail, while other protocols are briefly mentioned at the end. The different protocols have benefits compared to the others for different types of circuits.

Single-rail protocols

In single-rail protocols, each bit of data is transferred over a single line, with the request and acknowledge signals on separate lines. The single-rail protocols are often called bundled-data protocols, as the request and acknowledge signals are bundled together with the data signal(s). The single-rail scheme can be in the form of a 2-phase or 4-phase protocol, which differs in how many transition events are needed to complete a data transfer.

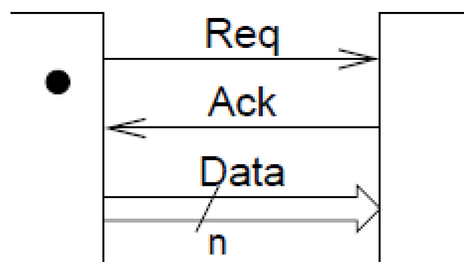


Figure 2.2: Single-rail scheme

In figure 2.2, the black dot represents the active side of the transaction. A request, along with the guarantee of valid data, is issued from the active to the passive side of the channel. An acknowledgment from the passive side means that the data is received. This transaction is a PUSH. If the data lines had been reversed, the request would have been a request for data from

the right. The acknowledgment would then mean that the data is valid and ready. This other kind of transaction would be a PULL.

Single-rail control circuits are said to be speed-independent, while the data path circuits are self-timed. These concepts are explained later. Data representation is not affected by what single-rail protocol (2-phase or 4-phase) is used since it is separated from the controlling wires, as opposed to the dual-rail protocols discussed later on.

4-phase single-rail (4PSR)

As mentioned, 4-phase means that four events, are needed to complete a data transfer. This can be illustrated as in figure 2.3. The active side of the transaction sets request high, followed by the passive side setting acknowledge high when data either has been received or is ready and valid, depending on whether the transfer is a PUSH or PULL, respectively. The active part then responds by setting request low, followed by the passive part setting acknowledge low. The active part can then start a new transaction. This is true for both push and pull transactions. 4-phase uses twice as many signaling edges as 2-phase, but the control circuits in 4-phase are often easier to implement. When driving storage elements, for instance, level-controlled latches can be driven directly from the signaling lines. The RTZ (return-to-zero) property of the protocol also means that the control lines maintain the same logical level after performing a transaction as they were before.

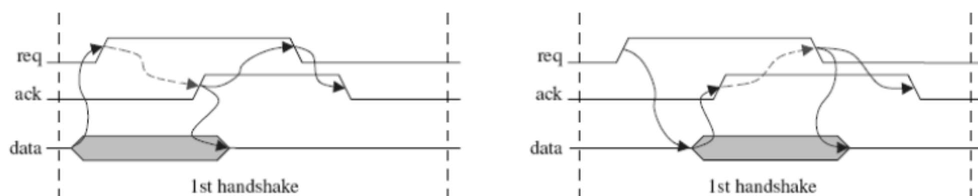


Figure 2.3: 4-phase Single-rail

2-phase single-rail (2PSR)

The 2-phase single-rail protocol interprets all transitions on the controlling wires as a signaling event. The level of the signal (binary 0 or 1) is not

important - just the transition - with a rising edge being equivalent to a falling edge. Figure 2.4 illustrates this. The active side transitions the request line, followed by the passive side transitioning the acknowledge line when the data either has been received or is ready and valid, depending on whether the transfer is a PUSH or PULL, respectively. Then, the active part is cleared to initiate a new transfer.

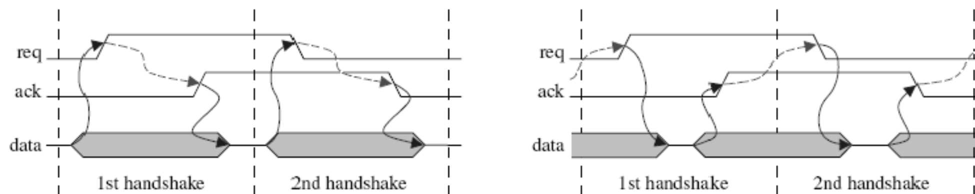


Figure 2.4: 2-phase Single-rail

Two transitions would in theory be faster than four transitions, but as mentioned, the control circuits for 2-phase protocols tend to be more complex, resulting in area overhead and possible speed degradation. The choice between 2-phase and 4-phase will depend on the function or system to be implemented, and one will not be better than the other for all applications.

Dual-rail protocols

Dual-rail signaling uses two wires for each data bit being transferred, where the request is implicit in the signal wires. It is common to represent the data bit d with the wires *dt* and *df*, where the t and f stands for true and false. The true wire indicates a logical 1 and the false wire indicates a logical 0. Only one acknowledge line is needed between components. This is illustrated in figure 2.5. The dual-rail signaling scheme uses two wires for each bit, along with one acknowledge line, resulting in 2N + 1 wires for each data channel, as opposed to the N + 2 wires needed for each data channel in the single-rail protocol. As for single-rail, dual-rail comes in the form of 2-phase and 4-phase protocols, pointing to the number of transition events needed to complete a data transfer.

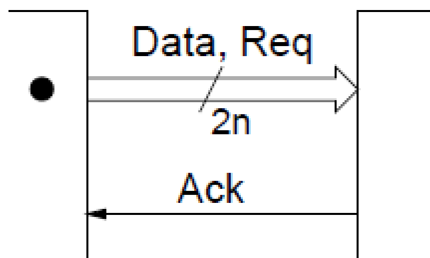
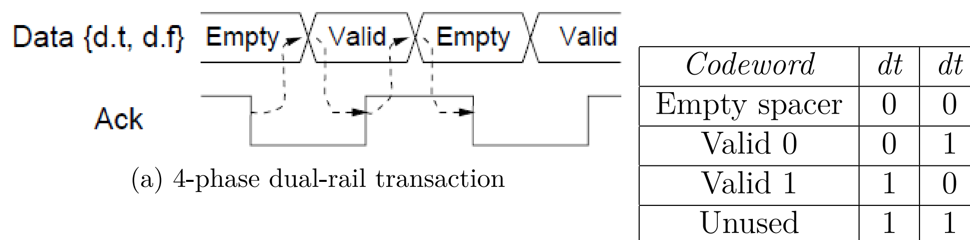


Figure 2.5: Dual-rail scheme

4-phase dual-rail (4PDR)

In the 4-phase dual-rail protocol, it is common to use one-hot encoding, where a valid 1 sets the true wire high, and a valid 0 sets the false wire high. Setting both wires to logical 0 represents an empty value, which is used as a spacer. The code words for this type of encoding is shown in table 2.6b, while a typical transaction is shown in figure 2.6a.



(a) 4-phase dual-rail transaction

Codeword	*dt*	*dt*
Empty spacer	0	0
Valid 0	0	1
Valid 1	1	0
Unused	1	1

(b) Truth table for 4-phase dualrail protocol

Figure 2.6: communication over 4-phase dualrail protocol

A data transfer is initiated with the data wires being set to a valid value. The other party acknowledges this by setting the acknowledge line high. The data wires are then returned to the empty value, which the other party responds to by setting the acknowledge line low. After this, a new transaction can be started. Because the data is inherent in the request and all the data bits need to be valid before an acknowledge, the protocol is said to be delay-insensitive [1]. This can make the protocol very attractive for some implementations. The obvious drawback with this protocol is the number of wires needed, increasing the area and switching activity.

2-phase dual-rail (2PDR)

Like the 4PDR protocol, the 2-phase dual-rail protocol also uses two wires per bit. Though instead of using the levels of the wires, it uses the transitions to encode information. The 2-phase protocol does not rely on the spacer; in stead, a new codeword is received when exactly one wire of each bit has made a transition. This is illustrated for a 2-bit channel in figure 2.7.

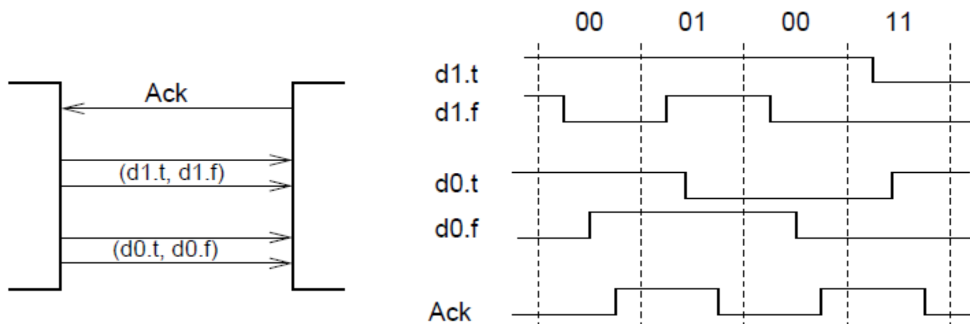


Figure 2.7: 2-phase dual-rail transaction

The true and false wires represent whether or not a logical 1 or 0 is being transmitted and both bits has to have a transaction on exactly one wire for data to be valid. In the figure, the transaction of the data word 00 is initiated with a transition on the false wires of both bits, which the receiver acknowledges by transitioning the acknowledge line. The next transaction is of the data word 01, which is represented with the false wire of the MSB and the true wire of the LSB having transitions, which again is acknowledged by a transition on the acknowledge line, and so on. The 2-phase dual-rail protocol has the advantage of decreased switching activity over the 4-phase version, but has the same amount of wires.

Other protocols

One-hot encoding and N-of-M encoding are other types protocols that are used, and the dual-rail protocols can be seen as subsets of these. One-hot circuits represent n bits with 2^n wires. Each line can then represent n bits of information. The Null Convention Logic (NCL) uses a 4-phase protocol where each wire represents either DATA or NULL [3]. In theory, an arbitrary number of wires can be used (for instance representing 10 possible values

with 10 wires), but typically only two wires are used, in the same way as for dual-rail, representing true or false.

N-of-M encoding uses groups of M wires to encode data values, and considers data to be valid as soon as N wires are activated. 1-of-2 encoding would be the dual-rail encoding. N-of-M codes are as mentioned for the dual-rail encoding said to be delay-insensitive [4].

Another delay-insensitive protocol, is the Phased Logic [5] which uses the Level-Encoded 2-phase Dual-Rail (LEDR) encoding scheme. Without going into detail, this encoding scheme uses two wires, where one wire transmits the logical value, and the other transmits phase information.

2.1.3 Special cells

Although comprised of the same basic components, there are some basic concepts which set synchronous and asynchronous circuits apart even at the lowest level. Some cells that are rarely seen in synchronous circuits are very important to asynchronous circuits, and the missing discrete time in asynchronous circuits mean that special timing considerations must be taken. These aspects are discussed below.

Asynchronous circuits can usually be constructed from the same components as synchronous circuits. This means that they can be implemented using standard cells like AND, NAND, OR and so on from VLSI(Very-Large-Scale-Integration) libraries and there is no need to design completely new cells. However, some standard cells are usually essential to the design of asynchronous circuits. Two of them are presented in the following sub sections, namely the event synchronizing C-element and the arbitrating Mutex (mutual exclusion unit).

Muller C-element

The Muller C-element is very common in asynchronous designs, and was originally defined by David E. Muller as a way of synchronizing events [1]. Figure 2.8 shows a gate level and transistor implementation of a C-element, in addition to the symbol.

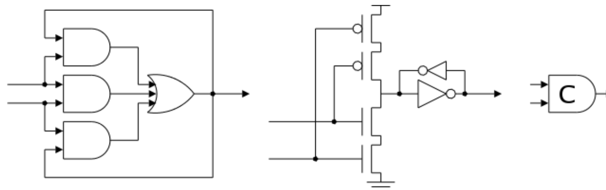


Figure 2.8: Muller C-element

There are several ways of implementing the C-element. Designing it from scratch as a new cell will give the best performance, but it can also be constructed from AND and OR gates. Regardless, it has to adhere to the truth table given in table 2.1. Here we see that the output only transitions if both inputs have the same value. It behaves like an OR-port when the output is high and like an AND-port when the output is low. The special behavior of the C-element makes it well suited for synchronizing signals.

input a	*input b*	*output y*
0	0	0
0	1	previous y value
1	0	previous y value
1	1	1

Table 2.1: Truth table for a Muller C-element

Mutex

A mutex is employed in arbitration between circuit components trying to access the same resource at approximately the same time. This is crucial in an asynchronous circuit, because sometimes it is difficult to completely determine the order in which events happen. Latches and other bi-stable devices may become metastable if two signals try to change the state of the device at the same time. A common 4-phase implementation involves cross-coupled NAND gates, and has the following behavior:

The mutex has two request inputs and two corresponding grant outputs. If both requests arrive at approximately the same time, grant 1 will be triggered. Grant 2 will not be triggered until request 1 is set low. The Mutex prevents both outputs becoming active at the same time and prevents a potentially metastable state. If the requests arrive close to one another, the Mutex may use a long time to decide [6].

2.2 Power consumption

The power consumption in CMOS technology can be split into three main components [7]; dynamic, leakage and short-circuit. This is shown in the following equation:

$$P_{total} = P_{dynamic} + P_{short-circuit} + P_{leakage} \tag{2.1}$$

The first term represents the dynamic component of power, which is determined by the switching activity in the circuit. The second term is due to the direct-path short circuit current, I_{sc}, which arises when both the NMOS and PMOS transistors are simultaneously active, conducting current directly from supply to ground. The last term is the leakage component, $P_{leakage}$, which is primarily determined by fabrication technology considerations.

2.2.1 Dynamic power dissipation

In a CMOS circuit, each node have an associated capacitance, C, consisting of a load capacitance and parasitic capacitances. During operation, C is charged when the node switches value from logical zero to logical one and each time an amount of power is dissipated. The amount of power dissipated, also known as the dynamic power component, is given in equation 2.2:

$$P_{dynamic} = \alpha C V_{dd}^2 f_{clk} \tag{2.2}$$

V_{dd} is the supply voltage, f_{clk} is the clock frequency and α is the average number of $0 \rightarrow 1$ transitions per clock cycle. Equation 2.2 assumes that a logical one refers to the voltage V_{dd}, the power dissipated each transition is then CV_{dd}^2. The number of transitions is given as α times f_{clk}.

2.2.2 Short circuit power dissipation

In static CMOS, when the input to a logic gate changes value, there will be a short period of time when both the N- and P-network are partially conducting while the input voltage V_{in} changes value. This results in short circuit power dissipation due to the current flowing from V_{dd} directly to ground. Both N- and P-networks are (partially) on when $V_{thn} < V_{in} < (V_{dd} - |V_{thp}|)$ [8]. Here the voltages V_{thn} and V_{thp} are the threshold voltages of the NMOS and PMOS transistors in the logic gate. [8] states that with careful design, this power consumption source can be kept to be less than 15 % of the dynamic power.

2.2.3 Leakage power dissipation

Leakage power is power consumption due to unintended leakage currents flowing though transistors. The leakage current can be split into three main components [8]: reverse-bias diode leakage on the transistor drains, tunneling current through gate oxide, and sub-threshold leakage through the channel of a transistor turned off. The diode leakage occurs when a transistor is turned off and the drain/source is reverse-biased by another device. The leakage current through the gate oxide happens if the oxide is sufficiently thin, though the probability of leakage current drops off exponentially with oxide thickness. The sub-threshold leakage current occurs due to carrier diffusion between the source and the drain nodes when the gate-source voltage, V_{gs}, has exceeded the weak inversion point, but is still below the threshold voltage V_{th}. According to [9] the power consumption due to leakage increases with shrinking transistor technologies and is set to exceeding total dynamic power consumption as technology drops below the $65nm$ feature size.

2.3 Metastability

Flip-flops have certain constraints with regards to the setup and hold time of their input. Violations of these constraints make the flip-flop enter a state where its output is unpredictable, this state is known as metastable state. When a flip-flop is in a metastable state, its output oscillate between a logic 0 and 1. Metastability typically occurs in these cases:

- When the input signal is an asynchronous signal.
- When the clock skew/slew is too much (rise and fall time are more than the tolerable values).
- When interfacing two domains operating at two different frequencies or at the same frequency but with different phase.
- When the combinatorial delay is such that flip-flop data input changes in the critical window (setup + hold window)

A common way to measure metastability is by the mean time between failure. A simplified way of calculating the MTBF is shown in equation 2.3 [10]. The formula uses the worst case way of representing the settling time of the flip-flop. A more accurate formula would contain an exponent on the form e^T where T depends on the settling time of the flip-flop in the case of metastability and the difference in time between the clock period and the propagation delay of the flip-flop.

$$MTBF = \frac{1}{f_{clk} f_{in} t_{pd}} \tag{2.3}$$

Where the f_{clk} is the frequency of the clock the flip-slop uses, f_{in} is the frequency of the input and t_{pd} is the propagation delay through the flip-flop. A more precise formula is presented in [11].

Chapter 3

Balsa

3.1 Introduction to balsa

This chapter introduces Balsa and some of its concepts to provide the basis necessary to understand the asynchronous design and netlist generation done in this thesis. A more thorough description of the Balsa system and language is given in the Balsa manual [12]. Balsa is freely distributed on the University of Manchester's home page [13].

Balsa is the name of both a framework for synthesizing asynchronous systems and a hardware descriptive language (HDL) used to describe such systems. It has been developed over several years by the ATP group (Advanced Processor Technologies) of the School of Computer Science, University of Manchester, and is based on the Tangram system made by Philips [12]. Balsa provides a higher abstraction level than gate level design of asynchronous circuits by introducing handshake components. More specifically Balsa provides a one-to-one mapping between the language constructs used to describe a circuit and the handshake circuit generated. While Balsa will not necessarily give the solution with the highest performance or smallest area, it makes it easier for the user to see the architecture of the generated circuit and make predictable changes to the implementation when changing the code. Balsa is most of all an attempt at building a tool capable of synthesizing large asynchronous systems while still keeping the compilation transparent and understandable.

A circuit described in Balsa is compiled into handshake components connected together by communication channels. Communication between the components takes place as handshakes defined by a protocol like the ones described in subsection 2.1.2. The handshakes may transfer data between the components or act as a control signal. If an asynchronous handshake circuit is designed in a common HDL like verilog or VHDL the choice of communication protocol between handshake modules has to be made at an early stage, as every component and module in the design have to follow the protocol. A lot of effort has to be focused on making the implementation fit the protocol during the design, rather than focusing it on functionality and optimization of the circuit. An eventual change of protocol at a late stage would necessitate a major rewrite of the circuit, though this is not the case with Balsa. With Balsa the focus of the design is on functionality and the protocol used is only determined when the a netlist is generated. This makes it possible to compare the different protocols in terms of area and performance to get the most suited implementation.

3.2 Balsa system

The balsa system consists of several tools [12], and some of the most important ones are listed below.

- **balsa-c:** the compiler for the Balsa language. The output of the compiler is an intermediate language breeze.
- **balsa-netlist:** produces a netlist appropriate to the target technology/CAD framework from a Breeze description.
- **breeze2ps:** a tool which produces a postscript file of the handshake circuit graph.
- **breeze-cost:** a tool which gives an area cost estimate of the circuit.
- **balsa-md:** a tool for generating makefiles
- **balsa-mgr:** a graphical front-end to balsa-md with project management facilities.
- **balsa-make-test:** automatically generates test harness for a Balsa description.

- **breeze-sim-control:** a graphical front-end to the simulation and visualization environment

Balsa-c interprets code written in the Balsa HDL into a handshake-component network into the intermediate breeze format. From this format cost estimations, behavioral simulations and network visualizations can be performed to improve the design. The design can also be compiled into a Verilog netlist via $balsa - netlist$, which is usable for other tools. Figure 3.1 provides a summary of the Balsa design flow.

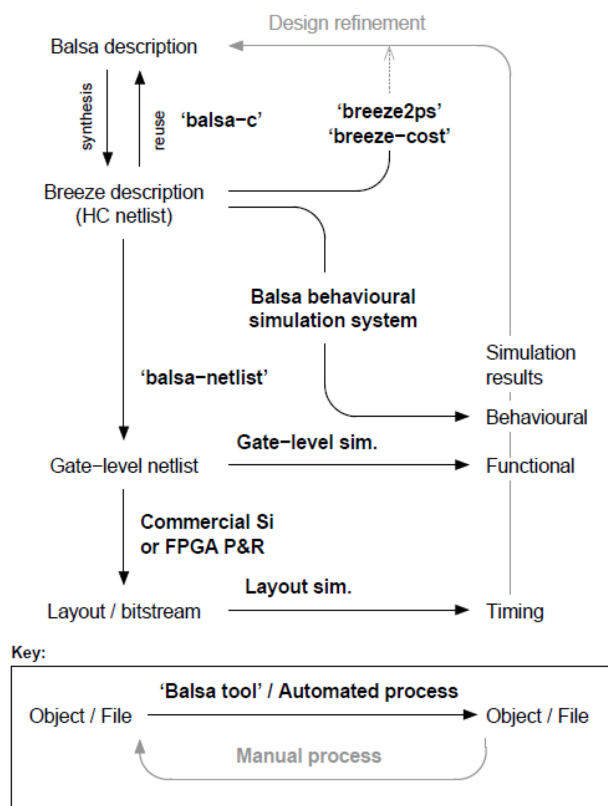


Figure 3.1: balsa design flow

The current Balsa version supports three different protocols; 4PSR, 4PDR and a 1-of-4 encoding scheme. Balsa also provides a few technologies for netlist production as extension packages. Some kind of technology has to be installed to make netlist production possible. In this design a technology provided by Atmel used for netlist generation.

3.3 Balsa language

This is a short introduction to the balsa language to provide the background necessary to understand the code presented in section 5.2. For a more thorough and in depth description of the balsa HDL, the balsa manual is recommended [12].

As mentioned in section 3.1, the balsa HDL is closely related to the handshake components making up the circuit implementation.

3.3.1 Language constructs

Table 3.1 present the different commands available in the balsa HDL.

Sync is a dataless handshake. The code initiates a handshake and halts the circuit until handshake is completed.

The two operators **<-** and **->** are used to assign data to and from channels. **<-** is used to assign a value to an output channel, while **->** is used to assign a value from an input channel to either an output channel or a variable. The **:=** operator is used to assign the result of an expression to a variable.

The **;** operator is used to denote sequentiality between statements. If two statements are separated by **;**, the first statement is performed before the other. On the other hand the || operator denotes concurrency. Two statements separated by || are executed at the same time.

continue is effectively a null command. It has no effect, but may be required for syntactic correctness in some instances. The **halt** command causes a process thread to deadlock.

The **loop** command can cause an infinite repetition of a block of code, or it can be finite if a **while** construct is added. The **for** loop construct is used for iteratively laying out repetitive structures, not for behavioral description as many are used to.

Balsa has **if** and **case** constructs to achieve conditional execution. While **case** statements only have one guard expression, while **if** statements can

Command	*Description*
sync	dataless handshake for control
<-	handshake data transfer from an expression to an output port
->	handshake data transfer to a variable from an input port
:=	assigns a value to a variable
;	sequential operator
||	parallel operator
continue	null command
halt	causes deadlock
loop ... end	loop forever
loop ... while ... then ... also ... end	conditional loop with optional initial command
for ... in ... then ... end	iteration used for repeating structures
if ... then ... else ... end	conditional execution, may have multiple guarded commands
case ... of ... end	conditional execution based on constant expressions
select ... end	non-arbitrated choice operator
arbitrate ... end	arbitrated choice operator
-> then ... end	handshake enclosure

Table 3.1: Balsa commands

have several.

The select **statement** chooses between the two input channels by waiting for data on either channel to arrive. The code enclosed between the select statement and the **end** is executed. If there is a possibility that the inputs arrive at the same time, the **arbitrate** statement must be used instead to ensure correct operation.

Normally handshakes are points of synchronization for assignments between channels or assignments between channels and variables. A transfer is requested and when all parties to the transaction are ready, the transfer completes. After completion of the handshake, the data provider is free to remove the data. If the data on a channel is required more than once, it must be stored in a variable. Balsa has two language constructs that allow the handshake on a channel to be held open whilst a sequence of actions completes. The handshake is said to enclose the other commands. Handshake enclosure can be achieved by use of the **select** or **arbitrate** command or by assigning channels into a command using the syntax: *<channels>* ***-> then*** *<commands>* ***end***

Common operators (such as and, or, +, -, =, > and <) that you find in almost every language are also found in the Balsa language.

Chapter 4

Startup logic specification

4.1 Startup system

In this project a system for ensuring the correct startup sequence for a microcontroller is designed. The system is shown in figure 4.1.

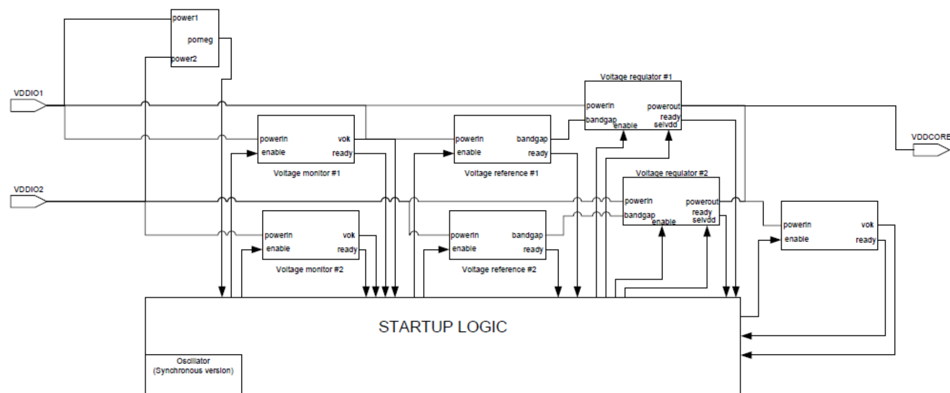


Figure 4.1: Startup system with controller and analog cells

The startup system consists of a controller labeled startup logic and several analog cells which it is meant to interface with. The specifications for the analog cells was given as follows.

4.2 Analog Cells

4.2.1 Power-on-Reset (POR) generator

A generator is a microcontroller peripheral that generates an active low reset signal, *porneg*, when power is applied to the device. This is to ensure that the device is powered up in a known state. In this design the POR cell monitors the voltage from two power supplies. When both voltages are below the given threshold, its output is a logic zero. When one of the power inputs rises above the threshold, its output stays zero for a predefined time, $Tpor$, before it then goes to a logic one.

Pin name	*Direction*	*Size*	*Comment*
power1	input	AVR32 POWER BITS	Power supply 1 monitored by the POR
power2	input	AVR32 POWER BITS	Power supply 2 monitored by the POR
porneg	output	1	Power-On Reset output - 0 : reset, 1 : not reset

Table 4.1: Pin Description for the POR

Parameter Name	*Default Value*	*Used for*
Threshold	1000 mV	POR threshold
Tpor	100 ns	minimum duration of reset

Table 4.2: Parameter list for POR

4.2.2 Voltage Monitor

A voltage monitor is a microcontroller peripheral used to monitor a power supply voltage, and signal whether the voltage is above or below a given threshold voltage, $VThreshold$. The *vok* signal indicates a voltage above the threshold with a logic one, and a zero if the voltage is below the threshold. When the cell is enabled it requires a certain amount of time, $TReady$, to properly turn itself on, and the signal *vok* signal is not valid before that. The voltage monitor uses the *ready* to indicate whether it has been turned on, a logic one indicated that it is on. If the supply voltage is changed there is a given delay, $TMeasure$, before the *vok* signal is updated.

Pin name	*Direction*	*Size*	*Comment*
power	input	AVR32 POWER BITS	Power supply monitored by the voltage monitor
enable	input	1	Enable signal : 1 : enabled, 0 : disabled
ready	output	1	Signals that the voltage monitor is ready (after enable 0->1)
vok	output	1	output - 0 : power<threshold, 1 : power>threshold

Table 4.3: Pin Description for a voltage monitor

Parameter Name	*Default Value*	*Used for*
VThreshold	1000 mV	Voltage monitor threshold
TReady	1000 ns	enable->ready time
TMeasure	100 ns	time needed to update vmon after a change in power

Table 4.4: Parameter list for a voltage monitor

4.2.3 Voltage Regulator

A voltage regulator is a device that is used to automatically maintain a constant voltage level. This voltage regulator can supply several different output voltages determined by the logic value of the *selvdd* input given in table 4.7. The *bandgap* input is a reference voltage used to generate the correct output voltage. When enabled, the voltage reference needs some time to properly turn itself on. Correct operation is indicated by the *ready* signal outputting a logic one. If the *selvdd* input is changed, the ready signal goes low until the output voltage has been updated.

Pin name	*Direction*	*Size*	*Comment*
powerin	input	AVR32 POWER BITS	input from power supply
powerout	output	AVR32 POWER BITS	output power
enable	input	1	Enable signal : 1 : enabled, 0 : disabled
bandgap	input	1	Bandgap reference voltage
selvdd	input	AVR32 VREG SELVDD BITS	see next table
ready	output	1	Signals that the voltage monitor is ready (after enable 0->1)

Table 4.5: Pin Description for a voltage regulator

Parameter Name	*Default Value*	*Used for*
TStart	20000ns	Time needed by the regulator to reach target voltage after enable 0->1
TChange	1000 ns	time needed to update powerout after a change in selvdd

Table 4.6: Parameter list for voltage regulator

Selvdd	*bandgap*	*powerout(mV)*
XXX	0	0
000	1	600
001	1	800
010	1	1000
011	1	1200
100	1	1400
101	1	1600
110	1	1800
111	1	2000

Table 4.7: Selvdd table for a voltage regulator

4.2.4 Voltage Reference

A voltage reference is a device that outputs a constant voltage regardless of the load on the device, power supply variations, temperature changes, and the passage of time. This specific voltage reference is modeled after a bandgap reference voltage and is used to provide a stable voltage to the voltage regulator. When enabled, the voltage reference needs some time to properly turn itself on. Correct operation is indicated by the *ready* signal outputting a logic one. The *bandgap* voltage is a voltage, but is modeled in this project as a logic signal for the sake of simplicity. A logic one indicates the correct bandgap voltage.

Pin name	*Direction*	*Size*	*Comment*
powerin	input	AVR32 POWER BITS	input from power supply
enable	input	1	Enable signal : 1 : enabled, 0 : disabled
bandgap	output	1	Bandgap reference voltage
ready	output	1	Signals that the voltage monitor is ready (after enable 0->1)

Table 4.8: Pin Description for a voltage reference

Parameter Name	*Default Value*	*Used for*
TReady	2000 ns	enable->ready time

Table 4.9: Parameter list for a voltage reference

4.3 Startup logic Controller

The following specification was given for the startup system

Two power supply inputs $VDDIO1$ and $VDDIO2$ may rise from 0v to 3.3V (for $VDDIO1$) or 5V ($VDDIO2$). When $VDDIO1$ or $VDDIO2$ rise above 1.0V the POR cell releases its reset, *porneg*. As long as the reset is active the startup logic is kept in a $RESET$ state and every analog peripheral should be disabled. When the reset is released the startup logic will then do the following operations:

- Turn-on Voltage monitor #1 and check if $VDDIO1$>1.6V ($VDDIO1$ is "good").
- Turn-on Voltage monitor #2 and check if $VDDIO2$>3.3V ($VDDIO2$ is "good").
- if $VDDIO1$ is good and $VDDIO2$ is "bad": Turn-on voltage reference #1, wait for it to be ready and then turn-on voltage regulator #1 (*selvdd* = 600mv).
- if $VDDIO1$ is bad and $VDDIO2$ is good: Turn-on voltage reference #2, wait for it to be ready and then turn-on voltage regulator #2 (*selvdd* = 600mV).
- if $VDDIO1$ and $VDDIO2$ are good: Turn-on voltage reference #2, wait for it to be ready and then turn-on voltage regulator #2 (*selvdd* = 600mV).
- if $VDDIO1$ and $VDDIO2$ are bad, wait for one of them to be good.
- When one of the voltage regulators has been turned on (with *selvdd* set to 600mV initially), the startup logic must increase the output voltage until the third voltage monitor says "voltage OK" (Threshold = 1.6V).

4.4 Asynchronous specification

The asynchronous implementation needs to follow the specification given in section 4.3, other than that only one specification was given; that the 4-phase dual-rail protocol should be used for communication.

Chapter 5

Design

5.1 Synchronous implementation

5.1.1 Design Considerations

The synchronous startup logic interacts and controls several analog cells. As the signals from the analog peripherals are asynchronous there is a definite chance that the will be metastability problems on the inputs.

By using formula 2.3 in section 2.3 the MTBF can be calculated. The available clock in the implementation technology was given as 20MHz and the given propagation delay of the flip-flops was between 0.5 ns and 1 ns depending on load, temperature and other conditions. Looking at the ready input from one of the voltage regulators where the update time TChange is 1000 ns, this gives a MTBF between 0.5μs and 1μs, which is a clearly an unacceptable number, and this is without considering the other any other ports as well.

A simple way of improving the MTBF is by using a slower clock, though that will impact performance. Which could be somewhat feasible considering that the startup logic spends a lot of time waiting for inputs from the analog cells, however since the MTBF is very bad the improvement isn't likely to be good enough.

The most common way of improving the MTBF is by adding successive flip-flops to improve synchronization on the input [11]. This gives any metastability problems more time to settle with each successive flip-flop, with the downside of added latency on the input and added area due to the extra flip-flops. A two-stage synchronizer is shown below in figure 5.1, though a three-stage synchronizer was used in the design.

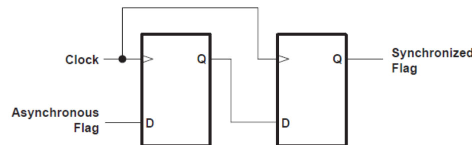


Figure 5.1: Two stage synchronizer

5.1.2 Implementation

The startup logic controller was implemented as a straight forward Mealy state machine, meaning that the next stage is dependent on both the current state and by the values of its inputs. The specified behavior of the system is mostly sequential with the exception when the startup logic increments the output voltage of one of the system by incrementing the *selvdd* signal to one of the voltage regulators. A simplified state diagram is shown in figure 5.2.

The system starts in the $RESET$ state waiting for the *reset* signal from the POR cell. None of the analog cells are enabled in this state. When the *reset* is released it moves on to the next state MON_ON.

In the MON_ON stage the two voltage monitors monitoring the power supply inputs are enabled. When both of the monitors have signaled that they are ready, the startup logic moves on to their next state MON_SELECT.

In the MON_SELECT state the next state is determined by the *vok* signals from the two voltage monitors. If the *vok* signal from monitor 2 is high, the next state is $REF2_ON$. If the *vok* signal from monitor two is low and the *vok* signal from monitor 1 is high, the next state is $REF1_ON$. If both signals are low the state machine stays in the MON_SELECT state until one of the signals goes high, and then sets the corresponding next state.

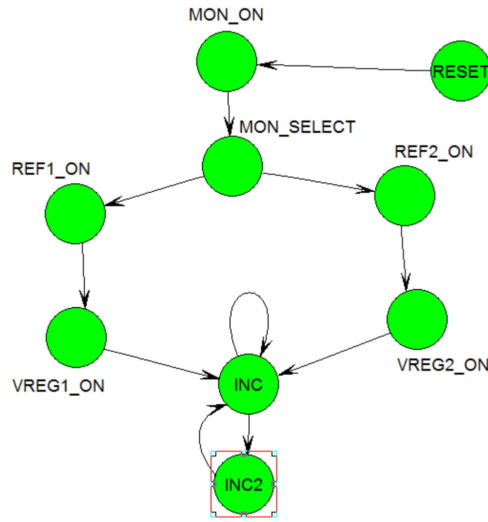


Figure 5.2: Simplified state Diagram of the Synchronous implementation

Both $REF1_ON$ and $REF2_ON$ are equal in behavior, though they control different voltage references. In both states the matching voltage reference is enabled and the state machine loops in that state until the *ready* signal from the voltage reference is set high. Then the next state is set to $VREG1_ON$ or $VREG2_ON$, depending on which state the state machine is currently in.

In the $VREG$ states voltage monitor 3 and the appropriate voltage regulator is enabled and the initial *selvdd* value set. The state machine loops in the state until voltage monitor 3 signals that it is ready and then the next state is set to INC.

The state INC and $INC2$ controls the incrementation of the output voltage $VDDCORE$ by incrementing the *selvdd* signal to the enabled voltage regulator, until the *vok* signal from voltage monitor 3 goes high. *Selvdd* is incremented whenever the voltage regulator signals that it is ready, as long as the *vok* signal from voltage monitor 3 is low and *selvdd* haven't been incremented too much. By too much it is meant that the counter used to assign *selvdd* its value hasn't reached the binary value 111. Incrementing further would make the counter over wrap and get the binary value 000. The state machine loops from the INC state to $INC2$ and back again. All the incrementation behavior is done in the INC state, the only purpose of the $INC2$ state is to give voltage monitor 3 time to update its *vok* signal.

Without the $INC2$ state the startup logic increments *selvdd* event though the output voltage from the voltage regulator was over the defined threshold. This is because there is a delay between the time when $VDDCORE$ reaches its threshold value and the time when voltage monitor 3 detects this. When the *vok* signal is set high the state machine loops in the INC state, though nothing is done until the state machine is reset by the reset signal from POR going low.

Table 5.1 shows which signals and which value they need to have for the state machine to move from the current state to the next state.

Current state	*Inputs values changing state*	*Next state*
RESET	POR reset = 1	MON_ON
MON_ON	mon1_ready, mon2_ready = 1	MON_SELECT
MON_SELECT	mon2_vok = 1	REF2_ON
MON_SELECT	mon2_vok = 0, mon1_ready = 1	REF1_ON
REF2_ON	ref2_ready = 1	VREG2_ON
REF1_ON	ref1_ready = 1	VREG1_ON
VREG2_ON	mon3_ready	INC
VREG1_ON	mon3_ready	INC
INC	mon3_vok = 0, mon3_ready, (vreg1_ready || vreg2_ready) = 1	INC2
INC2		INC

Table 5.1: Signals used to change the state

5.2 Asynchronous Implementation

5.2.1 Description

For the asynchronous implementation two different kinds of solutions was made, one modeled as a state machine like the synchronous implementation, and the other written to make an optimized implementation without regard for the readability of the code. The analog cells which the startup logic interacts with are the same, with the exception that interfaces has been added to enable communication in the specified 4-phase dual rail protocol.

Three different versions was made for both the state machine and optimized implementations. The difference between them is the interface used to connect to voltage monitor 1 and 2, meaning that the majority of the circuit is still the same between the different versions. The corresponding state machine and optimized implementations uses the same interfaces, meaning that state machine implementation 1 uses the same interfaces as the first optimized implementation, and so on.

5.2.2 Design Considerations

Balsa doesn't support passive pull ports. All data ports are either passive PUSH, active PUSH or active PULL ports. The recommended replacement is an active PUSH port enclosed by a sync select. The active PULL port on the receiving end of the channel functions as usual. An alternative solution is an active PULL port and an active PUSH port as earlier, but without the using the sync channel enclosure. In this case balsa generates a so called passivator between the two ports. A passivator is a "unit" with a passive PUSH port connected to the active PUSH port, and a passive PULL port connected to the active PULL port. The passivator functions as memory, storing the data from the PUSH channel until the PULL channel requests it, or waiting until the PUSH before acknowledging (sending the data) the PULL. To make this work properly there need to be a matching number of PULLs and PUSHes. If not, the procedures(threads) may end up waiting for a PUSH that never comes, or try PULLing data that won't be sent. This will make the threads controlling the port lock, possibly halting the entire module/procedure.

5.2.3 State machine implementation

State machine implementation 1 and overall behavior

The setstate procedure consists of balsa case-statements similar to verilog case-statements. The states themselves are also like the verilog cases used in the synchronous startup logic, modeling the same behavior for all the states, except the INC states and $RESET$ state. In addition there is a final state, $DONE$, where the stop variable is set to halt the main loop, stopping the circuit. To illustrate, the code of the first two states is given below.

Listing 5.1: First two states

```
1   case state_r of
2     MON_ON then
3      mon1_enable <- 1 ||
4      mon2_enable <- 1 ||
5      select mon1_ready then
6        continue
7      end ||
8      select mon2_ready then
9        continue
10     end ;
11     nextstate := MON_SELECT
12
13   | MON_SELECT then
14     vok1 -> vok1_t ||
15     vok2 -> vok2_t ;
16     if vok2_t = 1 then
17       nextstate := REF2_ON
18     | vok1_t = 1 then
19       nextstate := REF1_ON
20     else
21       nextstate := MON_SELECT
22     end
```

As with the synchronous implementation the analog cells are enabled and the ready and *vok* signals from the cells are then used to determine the next state. The big difference here is the MON_ON state where the state machine doesn't loop in the state, checking the input signals every iteration, instead the circuit waits for both the ready signals before moving on to the next state. The MON_SELECT state loops like in the synchronous implementation, Pulling the *vok* signal from the interface to the analog cells every iteration.

Like the MON_ON stage, the $REF1_ON$ and $REF2_ON$ stages enable their corresponding voltage reference and wait for a ready response. When the ready signal arrives the next state is set to $VREG1_ON$ or $VREG2_ON$ depending on current state. In the $VREG$ states voltage monitor 3 is turned on, as well as the voltage regulator associated with the state and the initial binary *selvdd* value of 000 is sent to the voltage regulator. When the ready signals from both the voltage monitor and the enabled voltage regulator arrives, the state machine PULLs the vok value from voltage monitor 3. After that then next state is set to INC and the current state is saved in a variable called *prevstate*. The *prevstate* variable is used in the INC state.

The INC state models controls the output, $VDDCORE$, by incrementing the *selvdd* value to the voltage regulator until the voltage is over a given threshold. In the INC state first test the vok3_t variable which is used to store the *vok* value PULLed from the voltage monitor monitoring $VDDCORE$. If the *vok* value stored in the variable is a logic 1 the state machine sets the *nextstate* variable to $DONE$, but if it is 0 the *selvdd* value sent to the voltage regulator is incremented. This is done by incrementing a count variable and PUSHing this value over the *selvdd* channel. The state then waits for a *ready* signal from the voltage regulator specified by the *prevstate* variable. Last the *vok* value is pulled and this value is tested in the next iteration of the main loop. The state machine stays in this state until the pulled *vok* value is 1.

While the the $INC2$ state is used to give $vok3$ time to properly update, a sync channel has been added in the vok-loop for the asynchronous implementations. The sync channel initiates a handshake with an external delay cell. When a suitable time has passed the delay cell acknowledges the handshake and the vok-loop continues.

The additional DONE state is as mentioned used to stop the circuit. It does this by setting the stop variable controlling the main loop. The main loop is shown in the code below.

Listing 5.2: Main loop

```
1  begin
2    state_r := MON_ON;
3    loop while stop = 0 then
4      setstate();
5      state_r := nextstate
6    end
7  end
```

The final difference to the synchronous version is the lack of a *RESET* state. The balsa-generated circuit has an *activate_r* input used to start the circuit, which is connected to the power-on-reset signal. When this input is high the circuit is active, while it is automatically resets when it goes low.

State machine implementation 2

An alternative to the MON_SELECT state was also made to prevent the startup logic from looping while it waits for one or more of the the pulled *vok* signals to be high. This state waits for a response to be sent instead of pulling it directly. As there is a chance of both signals arriving at the same time the startup logic has to arbitrate between them. If a variable called *vokmutex* is zero a value is written to it depending on which signal is chosen by the arbitration. If the variable already has been written to nothing happens and the state continues with the next part of its behavior. The next part is a check on the *vokmutex* variable to see which signal was arbitrated and using this to set the next state. The code for the state is shown in the code below.

Listing 5.3: MON_SELECT state from state machine implementation 2

```
1   | MON_SELECT then
2     arbitrate vok2 then
3       if vokmutex = 0 then
4         vokmutex := 2
5       end
6     | vok1 then
7       if vokmutex = 0 then
8         vokmutex := 1
9       end
10    end;
11    if vokmutex = 1 then
12      nextstate := REF1_ON
13    | vokmutex = 2 then
14      nextstate := REF2_ON
15    end
```

State machine implementation 3

According to [12] the arbitration circuit generated by balsa leads to a significant increase in area. To prevent this increase in area another implementation

was made. Instead of doing the arbitration in the startup logic this implementation will passively wait for a data signal from the interface with the arbitrated result. The interface is described in 5.3.5. In this implementation there is only one channel for the *vok* signal from the voltage monitors. The state waits for a handshake on the *vok* channel and the data sent is used to determine the next state. The code for the state is shown in listing 5.4

Listing 5.4: MON_SELECT state from state machine implementation 3

```
1  | MON_SELECT then
2    vok -> then
3      case vok of
4        1 then
5          nextstate := REF2_ON
6      | 0 then
7          nextstate := REF1_ON
8      end
9    end
```

5.2.4 Optimized implementation

Optimized implementation 1 and overall behavior

This implementation takes advantage of the fact that sequential circuit behavior can easily be modeled in balsa. A lot of the code in the state machine is enabling an analog cell, and then waiting for one and more signals from the cell. This is easy to model in balsa and it isn't really necessary to use a state machine to control this behavior. The parts of the code where data values are pulled from the analog cells are a bit more complicated to model. Here a while loop is required to substitute the state machine looping in a particular state. The code in listing 5.5 models the same behavior as the MON_ON state and part of the MON_SELECT state in subsection 5.2.3.

Listing 5.5: code for beginning of optimized implementation 1

```
1   mon1_enable <- 1 ||
2   mon2_enable <- 1 ||
3   select mon1_ready then
4     continue
5   end ||
6   select mon2_ready then
7     continue
8   end ;
9   loop while (vok1_t or vok2_t) = 0 then
10    vok1 -> vok1_t ||
11    vok2 -> vok2_t
12  end ;
```

The part of the code before the while loop is exactly like the *MON_ON* state, but instead of setting the *nextstate* variable for use in the next iteration of the state machine a single sequential operator is used to indicate that the while loop is executed afterwards. The loop PULLs the *vok* signal from voltage monitor 1 and 2 in each iteration and stores them in the variables *vok1_t* and *vok2_t*. When one or both of the variables are 1 the loop exits and the startup logic moves on to the code shown in listing 5.6.

Listing 5.6: code for selecting a voltage reference

```
1   if vok2_t = 1 then
2     ref2_enable <- 1 ||
3     select ref2_ready then
4       continue
5     end ;
6     vreg2_enable <- 1
7   | vok1_t = 1 then
8     ref1_enable <- 1 ||
9     select ref1_ready then
10      continue
11    end ;
12    vreg1_enable <- 1
13  end ;
```

This part of the code handles the selection of which of the voltage references and voltage regulators are to be turned on. Depending on the *vok1_t* and *vok2_t* variables either voltage reference 1 is turned on or voltage reference 2 is turned on. When the voltage reference that is turned on signals that it is ready the corresponding voltage regulator is turned on. After this voltage monitor 3 is turned turned on, while the other two voltage monitors are turned off. When voltage monitor 3 signals that it is ready the startup logic

begins the incrementation part of its behavior. This is shown in listing 5.7.

Listing 5.7: incrementation loop for optimized implementation

```
1  loop while vok3_t = 0 then
2    inc();
3    selvdd <- count ;
4    if vok2_t = 1 then
5      select vreg2_ready then
6        inc()
7      end
8    | vok1_t = 1 then
9      select vreg1_ready then
10       inc()
11     end
12   end ;
13   --sync delay ;
14   vok3 -> vok3_t ;
15 end
```

Like in the state machine implementation the while-loop controlling the incrementation loops as long as the value pulled from voltage monitor 3 isn't a logic 1. First a counter is incremented and the value is sent over the *selvdd* channel to the interface. The inc-procedure used to increment a counter is shown in listing 5.8. Then an if-sentence checks whether to wait for a ready signal from voltage regulator 1 or 2. When the ready signal arrives the vok3 signal is PULLed from voltage monitor 3.

Listing 5.8: procedure for incrementation

```
1 shared inc is
2 begin
3   if count < 7 then
4     count := (count +1 as 3 bits)
5   end
6 end
```

The inc procedure is used to increment a counter used to assign the next *selvdd* value. The procedure has a if-guard to ensure that the counter doesn't increment when the current count value is a binary 111, so that the counter doesn't over wrap and get a new binary value of 000. When the incrementation is done the next *selvdd* value is sent to the voltage regulator. When the loop finishes the startup logic is done and sets its activate acknowledge output high. As long as the activate request input doesn't go low the startup logic stays static.

Optimized implementation 2

The optimized implementation 2 uses the same interfaces and solution as the second state machine implementation, for the same reason. Arbitration with a *vokmutex* variable is used, but instead of setting the next state, it enables the selected voltage reference. When the voltage reference signals that it is ready the corresponding voltage regulator is turned on. This code substitutes the code in listing 5.6 and the vok-loop in listing 5.5 from optimized implementation 1, but the rest of the code is the same. The changed code for this implementation is showed in listing 5.9.

Listing 5.9: Code changed in optimized implementation 2

```
1   arbitrate vok2 then
2     if vokmutex = 0 then
3       vokmutex := 2
4     end
5   |     vok1 then
6     if vokmutex = 0 then
7       vokmutex := 1
8     end
9   end;
10  if vokmutex = 1 then
11    ref1_enable <- 1 ||
12    select ref1_ready then
13      continue
14    end ;
15    vreg1_enable <- 1
16  | vokmutex = 2 then
17    ref2_enable <- 1 ||
18    select ref2_ready then
19      continue
20    end ;
21    vreg2_enable <- 1
22  end ;
```

Optimized implementation 3

This implementation is the optimized equivalent of state machine implementation 3. Compared to the code in optimized implementation 1 it only substitutes the vok-loop in listing 5.5, leaving the rest of the code unchanged. The changed code for this implementation is showed in listing 5.10. In state implementation 3 the nextstate variable is set within the case-statement, but

in this implementation the one of the temporary variables vok1_t or vok2_t is set to one. The temporary *vok* variables are used to select which voltage reference is turned on in listing 5.6, like in optimized implementation 1.

Listing 5.10: Code changed in optimized implementation 3

```
1  vok -> then
2    case vok of
3      1 then
4        vok2_t := 1
5    | 0 then
6        vok1_t := 1
7    end
8  end ;
```

5.3 Interface

5.3.1 Choice of interface

To enable communication between the analog cells and the startup logic there has to be an interface that controls the communication. Several types of interfaces was considered to enable communication between the asynchronous startup logic and the analog cells. The analog cells responds as specified earlier in section 4.2, and all the proposed interfaces follows the specified 4-phase dual rail transfer protocol. The sync interface could be used with and other protocol as well.

To determine which kinds of interfaces was needed the signals to and from the analog cells were first considered. The signals could either be considered a sync signal or a data signal, and they further be either a passive or active signal. The ready signals from the various analog cells are all treated the same by the startup logic. After enabling an analog cell, the startup logic waits for the ready signal from the cell before continuing. This behavior makes it natural to implement the interface to the ready signals as sync channels where the ready signal initiates the handshake. This interface is presented in subsection 5.3.2.

When enabling the various analog cells the startup logic needs to set the the enable input of a cell high, as long as the cell should be enabled, and low when it should be off. This makes it natural to implement the interface as a data channel to ensure full control over the enabling of the cells. As the startup logic controls the sending of the data, the interface must be made as the passive end of the channel. The *selvdd* signal is in principle the same, but it has a data width of 3 bits instead of 1 bit for the enable signals. The interface presented in subsection 5.3.3 is used for both the *selvdd* and enable signals, with only few modifications between them.

The last signal to consider between the startup logic and the analog cells is the *vok* signal from the voltage monitors. In the various asynchronous implementations the only difference with regards to the interfaces are with the vok signals. When the startup logic loops, incrementing *selvdd*, it also checks the *vok* signal from voltage monitor 3, this process is the same for all the implementations. Here the startup logic PULLs the *vok* signal every iteration of the loop. In this case the *vok* signal has to be a data signal, and the

interface has to provide it whenever the startup logic requests it. This makes the interface the passive side of a data channel. The interface is presented in subsection 5.3.4. In the first optimized and state machine implementations the vok signal is PULLed in the same way as in the incrementation part of the code, and the same interface is used.

The second optimized and state machine implementations waits for the *vok* signals from voltage monitor 1 and 2. In this case the *vok* signals have been implemented as sync channels and therefore uses the same interface as the ready signals.

In the third version of the optimized and state machine implementations the startup logic waits for a single data input representing which of the voltage monitors have been selected. Here the startup logic is the passive side of a PUSH channel, and the interface is the active side. The interface is presented in subsection 5.3.5.

5.3.2 Sync interface

The sync interface is designed as the active side of a sync channel. It's designed to send out a sync request when its input from an analog cell goes high, and lower the sync request when a sync acknowledge arrives from the startup logic. This interface has been realized using a c-element, an AND gate, an inverter and a simple pulse generator. Figure 5.3 shows the interface.

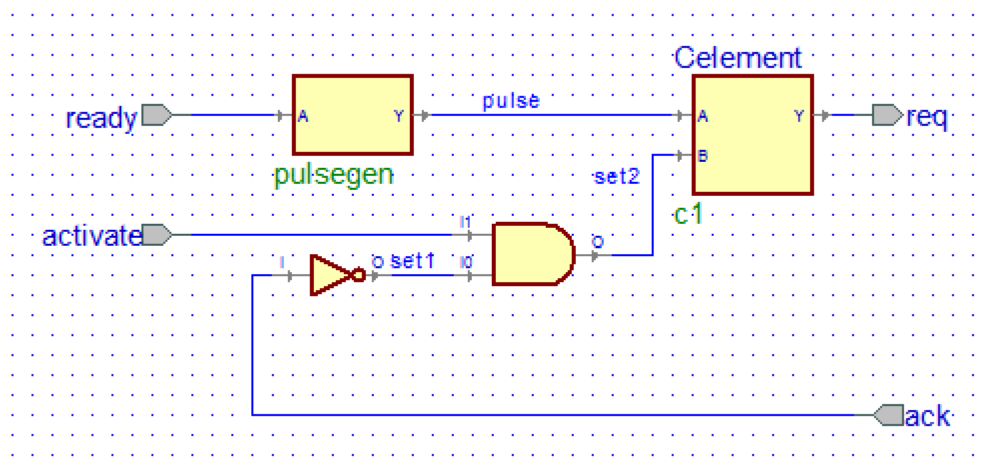


Figure 5.3: Sync interface

The output of the c-element is used as the sync request signal to the startup logic. As explained in subsection 2.1.3 the output of the c-element is set to 0 when both its inputs are 0, and 1 when both its inputs are 1. With any other combination of inputs the c-element will keep its previous output value. Initially the interface outputs a logic zero to the startup logic, and the sync acknowledge input it gets back is a logic zero as well. The acknowledge signal is inverted and connected to one of the inputs c-element. The other input of the c-element is connected to the pulse generator. The input of the pulse generator is connected to the input from the analog cell. When the signal to the pulse generator goes high, the pulse generator send a pulse to the c-element. This briefly makes both the inputs to the c-element high, setting its output high. When the startup logic sets the acknowledge signal high the inverted input of the c-element is set low, making both the inputs low and setting the output of the c-element low again.

The pulse generator consists of an AND gate where both its inputs comes from the same source, though one is inverted. Not considering delays, the output of the AND gate would always be zero. Considering the delay through the inverter on one of the inputs there will be a short period, every time the signal to the pulse generator changes, where both the inputs to the AND gate are equal. This means that when the input signal goes high both the inputs to the AND will be high making its output high as well. It is important that the delay through the inverter is sufficiently long to make the pulse good enough to change the value of the c-element.

The last part of the interface is an AND gate that is used to reset or initialize the c-element to a known value, 0. One of the inputs to AND gate is connected to a reset signal, while the other is connected to the inverted acknowledge input. The output is connected to the input of the c-element. The AND gate works as an enable for the inverted acknowledge input. Whenever the reset signal is high the inverted acknowledge input is transported to the input of the c-element, while if the reset signal is low the signal to c-element is low as well regardless of the value of inverted acknowledge signal. As the reset the signal from the POR is used. This is to make sure that the sync request signal is low whenever the startup logic is reset by POR.

5.3.3 Passive PUSH Interface

The passive push interface is designed as passive side of a PUSH channel. When it receives a valid 4-phase dual rail data signal it will respond by setting its acknowledge output high. When the both the wires of a data input bit is low the acknowledge signal is low. The output from the interface to an analog cell is made to keep a constant value depending on the last data input from the startup logic. For each bit in the of the data being sent there is a latch and an acknowledge generator. Figure 5.4 shows the interface.

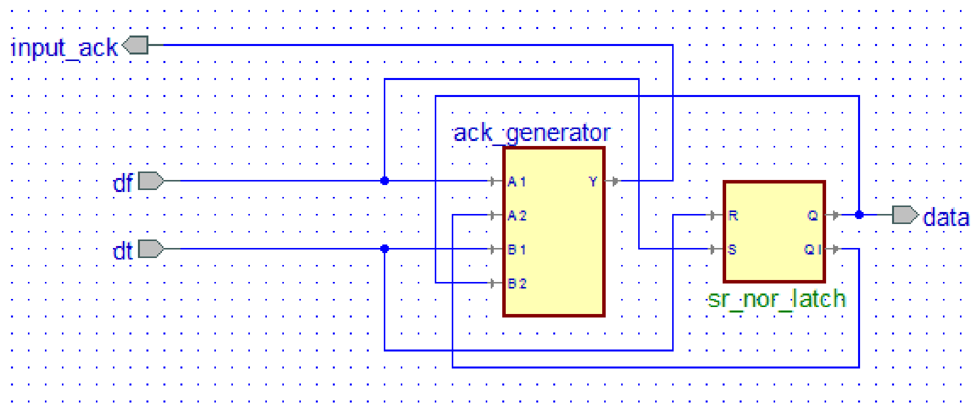


Figure 5.4: Passive PUSH interface

The acknowledge generator consists of two AND gates with their outputs connected to an XOR gate. The output of the OR gate is the acknowledge signal sent to the startup logic. The acknowledge generator should only send a high output when both the input and the output data is valid. By valid it is meant that one of the data wires of an input bit, df or dt, has to be high, and the corresponding output on the latch has to be high as well. A data wire is connected to one of the AND gates together with the corresponding latch output, while the other data wire and latch output is connected to the other AND gate.

The S input of the SR latch is connected to the dt wire of a data input bit, while the df wire is connected to the R input. When S is high and R is low the Q output of the latch is high, while the Qi input is low. A one bit data input of 1 (dt, df = 1, 0) gives Q a value of 1, and a data input of 0 (dt, df = 0, 1) gives Q a value of 0. Because of this Q is used as the output to the analog cell. When both the data wires are low the output of the latch will stay unchanged.

If more than one bit of data is to be sent from the startup logic to the analog cell there is as mentioned one latch and one acknowledge generator per bit, but the outputs from all the acknowledge generator has to be ANDed as well. This means that all the acknowledge generators have to have a high output before an acknowledge to the startup logic is sent. This also means that all the inputs and outputs have to be valid before an acknowledge is sent.

5.3.4 Passive PULL Interface

This interface is designed as the passive end of a PULL channel. It is designed to send out a 4-phase dual rail data signal whenever the request signal from the startup logic is high. When the request signal is low, both the data wires are supposed to be zero. There are two proposed implementations, though they are pretty similar.

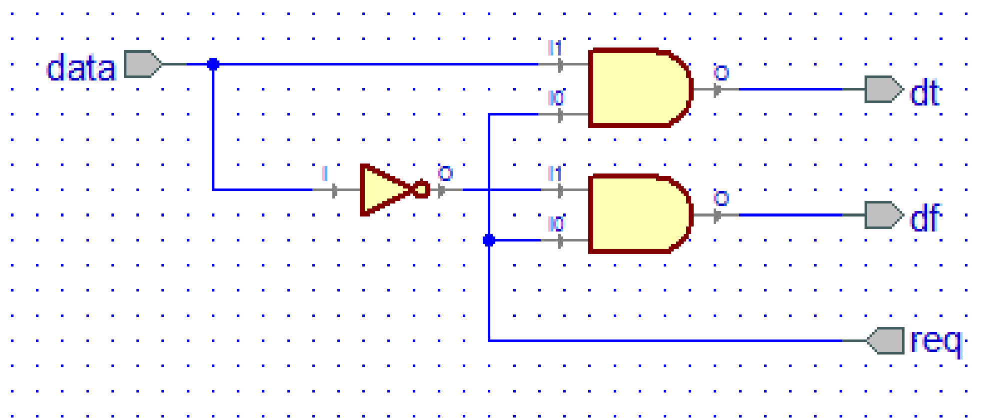


Figure 5.5: Passive PULL interface

The simpler of the two implementations consists of two AND gate and an inverter. The data signal to the interface is split and connected one of the inputs of each of the two AND gates. One of the split signals are inverted. The request signal from the startup logic is connected to the other input of both of the two AND gates, and functions as a enable. When the request is high the values on the other inputs are propagated as the data to the startup logic. If the request is low both data lines are low. The inverted data input is used as the df data signal, while the unchanged data input is used as the dt data signal. The first proposed interface is shown in figure 5.5.

There is one problem with the simple implementation. If the data input to the

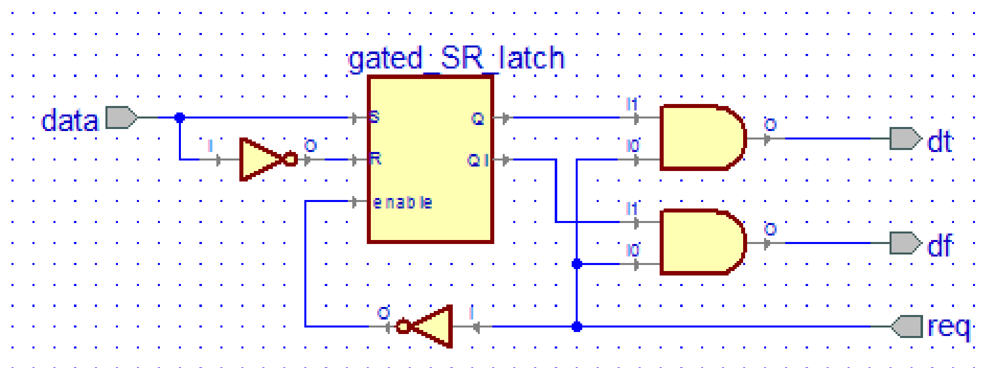


Figure 5.6: Modified passive PULL interface

interface changes while the request is high the data signal sent to the startup logic will change and this will not work with the protocol used. The second implementation fixes this. A gated SR latch is added between the AND gates and the data input. The data signal is connected to the S port of the latch, while the inverted data input is connected to the R port of the latch. The Q output from the latch is connected to the dt AND gate, while the Qi output is connected to the df AND gate. An inverted request signal is connected to the enable gate of the latch to make sure that its outputs doesn't change while the request signal is high. The modifies interface is shown in figure 5.6

5.3.5 Active PUSH Interface

The active push interface is designed as the active side of a PUSH channel. It is designed with two inputs from analog cells and a one bit 4-phase dual rail channel. When one of the inputs from the analog cells goes high the interface will initiate a PUSH with data indicating which of the input went high. The interface is initiated with both wires of the output bit being low and when the acknowledge signal arrives on the channel during a PUSH, the wires are both set low again.

Since this interface has two inputs from independent analog cells there is a certain chance of ending up in a metastable state if both inputs goes high at approximately the same time. The interface has been made with a mutex unit, as described earlier in section 2.1.3 of the theory, to get around this potential problem. The inputs from the analog cells have been connected to request inputs of the mutex, via an AND gate. The grant outputs of the

mutex are each connected to a modified version of the circuit used in the sync interface. The only difference between the modified version and the original sync interface is that the resetting of the two c-elements has been combined. The interface is shown in figure 5.7

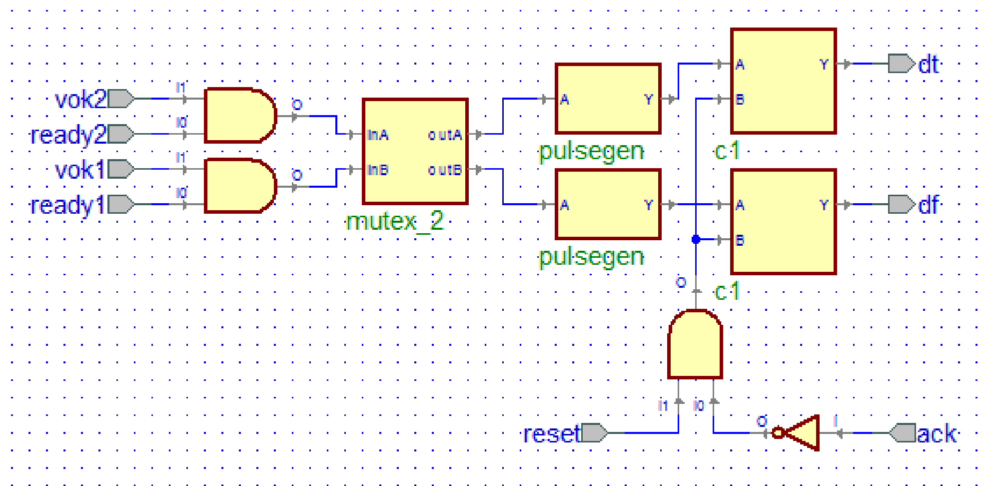


Figure 5.7: Active PUSH interface

The interface has been made to send information only once, when one of the inputs is set high or when both of the inputs are set high at the same time. The interface has no direct connection between the startup logic and the analog cell to acknowledge a selected request, meaning that there is no way for the startup logic to make the analog cell lower the signal connected to the request input. This means that the input not selected by the mutex will not have the chance to initiate a PUSH request.

Chapter 6

Evaluation and results

6.1 Testing

In this section the testing of both the synchronous and asynchronous implementation is presented. The startup logic is tested as an entire system, together with the analog cells, to verify correct behavior. This assumes correct behavior for the analog cells and this has been verified as well, but it is not presented in this thesis because the work involved is fairly trivial. Subsection 6.1.1 presents the test cases specified, while subsection 6.1.2 and 6.1.3 presents the testing of the synchronous ans asynchronous implementations respectively. All the implementations were found to be working.

6.1.1 Test cases

The test cases specified for the startup logic is quite simple. As the only signals in and out of the system is the two power inputs and the single power output, there is a very limited number of scenarios to test for and very few possible results. The different test scenarios are given below:

- Power supply 1 rises above the threshold specified for voltage monitor 1, and stays above the threshold, while power supply 2 does not rise above the threshold specified for voltage monitor 2

- Power supply 2 rises above the threshold specified for voltage monitor 1, and stays above the threshold, while power supply 2 does not rise above the threshold specified for voltage monitor 1
- Both power supplies rises above the thresholds of their corresponding voltage monitor and stays above the threshold

For all of the test cases above the power output $VDDCORE$ shall be the same, while a correct $VDDCORE$ voltage is not guarantied otherwise. If the selected input power supply drops below corresponding threshold voltage $VDDCORE$ might not be able to supply the correct voltage. This depends on which power supply is selected and how far below the corresponding threshold the voltage drops. If power supply 1 is selected and drops below the corresponding threshold $VDDCORE$ will supply a too low voltage, limited by the input voltage. This is because threshold specified for voltage monitor 1 and the voltage monitor used to control $VDDCORE$ is the same. For power supply 2 this is not necessarily the case. Because the specified voltage threshold for voltage monitor 2 is higher than threshold for the monitor used to control $VDDCORE$, the startup logic might still be able to supply a high enough voltage.

6.1.2 Testing of the synchronous implementation

The synchronous implementation was tested according to the test cases given in subsection 6.1.1. A waveform from the test is given in figure 6.1.

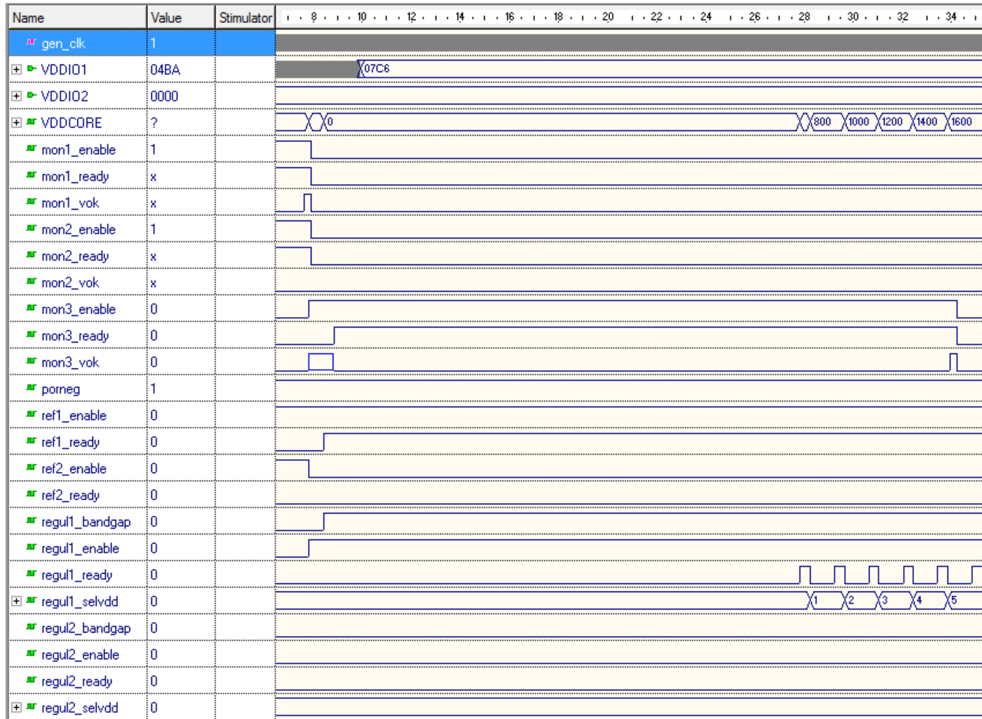


Figure 6.1: Waveform from the test of the synchronous implementation

6.1.3 Testing of the asynchronous implementations

Testing in Balsa

The first implementation written in balsa was also tested in balsa. To test the startup logic in balsa a test environment mimicking the entire system, including the analog cells, was made. The interface between the startup logic and the analog cells was incorporated in the analog cells themselves for the most part. Testing in balsa system is done mostly through the breeze2ps and balsa-make-test tools. Breeze2ps provides a graphic presentation of the generated handshake circuit, while balsa-make-test controls the test harness. As an example the graphical representation of the balsa version of a voltage monitor is shown below.

During simulation handshakes are shown on the graphical representation as color in the channels between the handshake components. In addition to this there is a text dump of the top level handshakes from the simulation in the

execution window. If there are deadlocks or other problems during the test it can be smart to add to add extra outputs for trouble shooting. In figure 6.2 a text dump from the testing of the startup logic is shown.

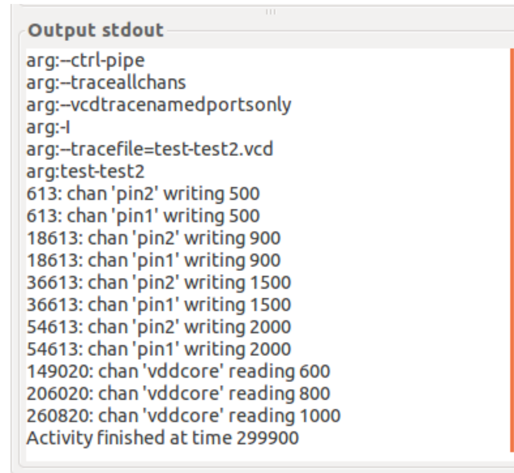


Figure 6.2: Simulation output in execution window

As mentioned earlier in this subsection, the test environment was made with separate analog cells. This was to get the correct dependency between the analog cells. For example, a voltage regulator cell should not output a voltage on its powerout without a bandgap signal from the corresponding voltage reference. This was more time consuming and complex than initially anticipated. Because the signals to the analog cells weren't guaranteed to not occur at the same time, arbitration between the different input signals had to be done within each analog cell. As balsa doesn't support arbitration of more than two signals without complicating the design by making arbitration trees (several stages of arbiters, where the arbitrated outputs are sent to the next stage until one signal is selected), the design of the analog cells for the test environment became more complicated.

Testing of verilog netlists

When testing the verilog netlists generated by balsa the same test environment used for testing the synchronous implementation was used, with the addition of the interfaces between the startup logic and the analog cells. Active HDL from Aldec was used for testing.

The interfaces was tested by themselves before the entire system was tested together. The process of testing the interfaces was fairly trivial, but when the total system was tested together some problems occurred. Neither the startup logic nor the interfaces were able to properly instantiate themselves. The outputs from the startup logic were dependent on their corresponding inputs to have known values (not binary X) to themselves have known outputs. To fix this the reset signal from the POR analog cell was used to instantiate the outputs from the interfaces to the startup logic.

Some of the netlists generated had an extra initialization input used to control part sequencing in the circuit behavior. These netlist were tested like any other implementation, without any problems other than having to add an extra input to the test bench. The figure below shows a waveform from the testing of one of the final implementations.

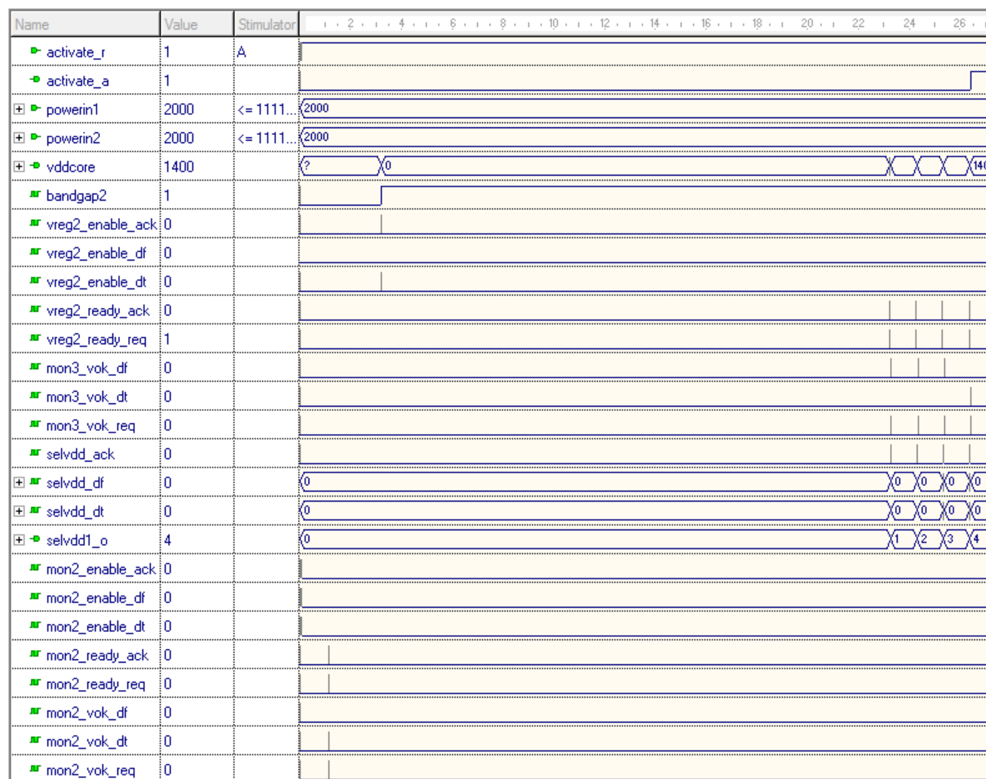


Figure 6.3: Waveform from the test of one of the asynchronous implementations

6.2 Synthesis

Design Compiler from Synopsys was used for the synthesis of the synchronous implementation. When synthesizing using Design compiler the design is read and a generic netlist consisting of generic components is generated. The area estimate given by Design compiler gives a size relative to a two input NAND-gate. The technology used for the synthesis was a 350nm library, and the size given for a NAND-gate in the library is 38.7.

As described in chapter 3 balsa provides its own area estimate and netlist generation. A technology package was provided by Atmel to be able to generate the netlist in the same 350nm library as the synchronous implementation. To get an area estimate to compare to the synchronous Design compiler was used. The netlist generated by balsa was loaded in Design compiler and synthesis was run without optimization enabled to keep the netlist unchanged.

Some of the netlists generated had an extra initialization input used to control part of the sequencing in the circuit behavior. In those cases Design compiler threw warnings because of supposed timing loops and possibly changed the circuit to prevent errors. The estimated area was unexpectedly large in those cases, either because of possible changes by Design compiler or because of genuine problems in the circuit. In those cases the design was rewritten.

The area estimate given by balsa bases its numbers on the handshake components used in the generated netlist. Theoretical numbers are given for the size of the different handshake components based on their relative size compared to each other. The area cost is only a guideline and has no relation to either the technology or protocol used in the netlist generation, and as such cannot be compared directly to area estimate for the synchronous implementation.

The area estimates from balsa and the synthesis is presented in table 6.1. If an extra sync channel is added to interface with an external delay cell, extra area of 627 have to be added to the area estimates for the asynchronous implementations.

Implementation	*Area estimate*	*Balsa estimate*
Synchronous state machine	9393	–
Asynchronous state machine 1	55764	4239
Asynchronous state machine 2	63970	4593
Asynchronous state machine 3	51790	3823
Asynchronous optimized 1	41055	2800
Asynchronous optimized 2	50164	3223
Asynchronous optimized 3	39520	2480

Table 6.1: Area estimates for startup logic

The interfaces were also synthesized. The area estimates for the interfaces are presented in table 6.2.

Interface	*Area estimate*
Sync	387
1-bit passive PUSH	252
3-bit passive PUSH	832
simple passive PULL	135
modified passive PULL	348
active PUSH	967

Table 6.2: Area estimates for the interfaces

6.3 Discussion

Six different asynchronous implementations have been made, though they are closely related. The first implementation made was state machine implementation 1. It was closely modeled after the synchronous state machine. The other state machine implementations were made later after the first had been thoroughly tested, both in balsa and verilog. The optimized implementations were made last. Optimized implementation 3 ended being the best of all the implementations. Both the startup logic and the interfaces used were smaller than the other implementations.

The first state machine implementation was the only implementation thoroughly tested in balsa. A test environment was made to be as close to the verilog environment as possible. Balsa provides a graphic representation of the device under test (DUT), where the handshakes are represented as color coded transitions during simulation. This was really useful during the initial design phase when errors were causing deadlocks in the behavior, but the time spent making the test environment was very high. It is possible that it would have been better to just generate a verilog netlist and do the testing on the generated code. The later implementations were mostly tested in active HDL, due to the balsa testing being time consuming.

By comparing the results of the synthesis it is clear that the synchronous implementation is a lot smaller than all the asynchronous ones. With an area estimate of 9393 it is smaller by a factor of over 4 than the smallest asynchronous implementation, and a factor of almost 7 for the largest implementation. In addition to the area of the startup logic controller comes the added area of the interfaces. The added area is 6157 for optimized and state machine implementation 1, 6427 for the second set of implementations and 5846 for the third set of implementations. The smallest asynchronous implementation had an area of 39520, 45366 including the interface. With the size of a 2-input NAND gate given as 38.7, the implementation consists of approximately 1172 gates. Even though this is a lot larger than the synchronous implementation the size isn't unreasonably large. A large digital system usually consists of millions of gates, making the startup logic a comparatively small part of the system.

C-elements represents a large part of the asynchronous implementations, 64 in optimized implementation 1 and comparable number in the others. The c-elements used in the synthesis consists of three and gates an a or gate. This

means that the c-elements have an area cost of about 14860. A simple way of reducing the size of the synthesized circuit would be by making a c-element primitive instead of making a gate implementation.

Timing differences between the synchronous and asynchronous implementations are negligible compared to the time used waiting for the analog cells. In fact most of the time is spent waiting for ready signals after enabling the cells. The timing of the synchronous implementation is more predictable than the asynchronous versions though. This is because it is clocked and its behavior easily predictable. The asynchronous implementations on the other hand have their timing determined by gate and line delays. A delay of 1 *ns* through all gates was used for testing purposes.

The timing isn't a problem for the most part, but in the loop controlling the incrementation of *selvdd* it has to be considered. When the voltage rises above the threshold for voltage monitor 3, there is a certain delay before the *vok* output is set high. If the *vok* signal is measured before it has been updated the startup logic will increment *selvdd* one extra time, making the output $VDDCORE$ higher than necessary. The synchronous startup logic solves this by adding the $INC2$ state, giving the voltage monitor enough time to update itself. The asynchronous implementation can't solve this as easily. It could either be solved by making sure the *vok* isn't measured prematurely by matching the delay within the startup logic or by having some external delay. The safest way of handling it is perhaps by having an external cell which when enabled returns a ready/finish signal after a suitable time. In the startup logic this could be implemented as a sync handshake where the request from the startup logic is used to enable the cell. The ready signal from the cell serves as the acknowledge signal to the startup logic. When the startup logic receives the acknowledge from the cell the request signal is lowered and the cell is disabled. This change leads to an area increase of 627. The voltage provided by the voltage regulator may rise over the threshold for voltage monitor 3 before it stabilizes. In this case the *vok* signal can be updated before the voltage regulator signals that it is ready, and the external delay cell wouldn't be needed.

As shown in [14], one of the advantages the asynchronous startup logic has over the synchronous version is the circuits ability to operate close to the sub-threshold area. This both lowers the power consumption and enables the circuit to begin its operation before its supply voltage has risen to its final level. Since the startup logic is implemented in the delay insensitive 4PDR protocol it is immune to delay variations because of sub-threshold operating

conditions and temperature. This is not the case for the synchronous circuit where the added delay means that the circuit may not meet its deadline constraints.

Another advantage of the asynchronous implementation is its lower power consumption. A lot of the time is spent waiting for the analog cells, and in that case the asynchronous implementations don't have any switching activity. The same is the case when the circuit reaches a stable state after it finishes incrementing *selvdd*. When the synchronous implementation is in a stable state or waiting for the analog cells, the clock still tick and the state machine loops in what ever state it currently is in. This means that the synchronous implementation constantly have some dynamic power consumption, while the asynchronous implementations won't have any dynamic power consumption when it is stable or waiting. The asynchronous implementation will have a somewhat larger static power consumption because of its high gate count, but as long as the technology used isn't very small (below 100 nm), the static power dissipation will be considerably smaller than dynamic.

Chapter 7

Conclusion

An asynchronous implementation of a startup logic controller has been designed, implemented for comparison to a synchronous version. The startup logic controller is part of a larger system where it is meant to interface with and control various analog cells. The startup logic controller was designed and tested in balsa, and a verilog netlist was then generated from the balsa description. Additionally interfaces between the startup logic and the analog cells were made to enable communication over the 4PDR handshake protocol. The total system with the generated netlist, analog cells and the interfaces between them was tested. A total of six asynchronous implementations were made and is presented in this thesis. With regards to area optimized implementation 3 is the superior solution.

Compared to the synchronous implementation the smallest asynchronous implementation and connected interfaces, is over 4 times larger, but as part of a larger design it is still comparatively small. An easy way of reducing the size of the asynchronous implementation is to make a c-element primitive instead of making the c-elements from AND and OR gates.

The asynchronous implementation have some advantages over the synchronous implementation. First of all, it is delay insensitive and therefore robust towards changes in temperature and voltage, which affects circuit delay. Secondly, it has a lower power consumption. The static power consumption a higher due to a larger gate count, but this is small compared to the savings in dynamic power consumption. When the asynchronous circuit is waiting or static, the dynamic power consumption is zero.

Further work on the startup logic should perhaps focus on a more accurate delay modeling and power estimations. Overall the asynchronous implementation seems to be a solution with good potential.

Bibliography

[1] J. Sparsø, "Asynchronous circuit design - a tutorial," in *Chapters 1-8 in "Principles of asynchronous circuit design - A systems Perspective"*. Boston / Dordrecht / London: Kluwer Academic Publishers, dec 2001, pp. 1–152. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?855

[2] P. A. Beerel, R. O. Ozdag, and M. Ferretti, *A Designer's Guide to Asynchronous VLSI*. Edinburgh Building, Cambridge CB2 8RU, UK: Cambridge university press, feb 2010.

[3] K. M. Fant and S. A. Brandt., "Null convention logic," [Online; accessed 16-August-2011]. [Online]. Available: http://www.theseusresearch.com/NCLPaper01.html

[4] T. Verhoeff, "Encyclopedia of delay-insensitive systems," [Online; accessed 16-August-2011]. [Online]. Available: http://edis.win.tue.nl/edis.html

[5] D. Linder and J. Harden, "Phased logic: supporting the synchronous design paradigm with delay-insensitive circuitry," *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 1031 – 1044, sept 1996.

[6] C. E. Molnar and I. W. Jones, "Simple circuits that work for complicated reasons," *Asynchronous Circuits and Systems, International Symposium on*, vol. 0, p. 138, 2000.

[7] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power cmos digital design," *Solid-State Circuits, IEEE Journal of*, vol. 27, no. 4, pp. 473 –484, apr 1992.

[8] M. Pedram and J. Rabaey, *Power Aware Design Methodologies*, 2002.

[9] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68 – 75, dec. 2003.

[10] Asic-world, "What is metastability?" 2011, [Online; accessed 16-August-2011]. [Online]. Available: http://www.asic-world.com/tidbits/metastablity.html

[11] Altera, "Understanding metastability in fpgas," 2009. [Online]. Available: http://www.altera.com/literature/wp/wp-01082-quartus-ii-metastability.pdf

[12] D. Edwards, A. Bardsley, L. Janin, L. Plana, and W. Toms, *Balsa: A Tutorial Guide.*, 2006. [Online]. Available: ftp://ftp.cs.man.ac.uk/pub/amulet/balsa/3.5/BalsaManual3.5.pdf

[13] T. U. o. M. APT group of the School Of Computer Science, "The balsa asynchronous synthesis system homepage." [Online]. Available: http://apt.cs.man.ac.uk/projects/tools/balsa/

[14] N. Lotze, M. Ortmanns, and Y. Manoli, "A study on self-timed asynchronous subthreshold logic."

Appendix A

Balsa Code

A.1 State machine implementations

A.1.1 State machine implementation 1

This is the balsa code for state machine implementation 1:

```
import [definitions]

procedure startup_state1 (
  sync mon1_ready;
  sync mon2_ready ;
  sync mon3_ready ;
  input vok1 : 1 bits ;
  input vok2 : 1 bits ;
  input vok3 : 1 bits ;
  sync ref1_ready ;
  sync ref2_ready ;
  sync vreg1_ready ;
  sync vreg2_ready ;
  output mon1_enable : 1 bits ;
  output mon2_enable : 1 bits ;
  output mon3_enable : 1 bits ;
  output ref1_enable : 1 bits ;
  output ref2_enable : 1 bits ;
  output vreg1_enable: 1 bits ;
  output vreg2_enable : 1 bits ;
  output selvdd : 3 bits
```

```
22 ) is
23
24 variable state_r : State
25 variable nextstate : State
26 variable prevstate : State
27 variable count : 3 bits
28
29 variable vok1_t : 1 bits
30 variable vok2_t : 1 bits
31 variable vok3_t : 1 bits
32 variable stop : 1 bits
33
34 shared mon_off is
35 begin
36   mon1_enable <- 0 ||
37   mon2_enable <- 0
38 end
39
40
41 shared mon3_on is
42 begin
43   mon3_enable <- 1 ||
44   select mon3_ready then
45     continue
46   end ;
47   vok3 -> then
48     vok3_t := vok3
49   end ;
50   prevstate := nextstate ;
51   nextstate := INC
52 end
53
54 procedure setstate is
55 begin
56   case state_r of
57
58     MON_ON then
59     mon1_enable <- 1 ||
60     select mon1_ready then
61       continue
62     end ||
63     mon2_enable <- 1 ||
64     select mon2_ready then
65       continue
66     end ||
67     nextstate := MON_SELECT
68
69
70   | MON_SELECT then
```

```
71      vok1 -> vok1_t ||
72      vok2 -> vok2_t ;
73      if vok2_t = 1 then
74        nextstate := REF2_ON
75      | vok1_t = 1 then
76        nextstate := REF1_ON
77      else
78        nextstate := MON_SELECT
79      end
80
81    | REF1_ON then
82      mon_off() ;
83      ref1_enable <- 1 ||
84      select ref1_ready then
85        nextstate := VREG1_ON
86      end
87
88    | REF2_ON then
89      mon_off() ;
90      ref2_enable <- 1
91      select ref2_ready then
92        nextstate := VREG2_ON
93      end
94
95    | VREG2_ON then
96      vreg2_enable <- 1 ||
97      select vreg2_ready then
98        continue
99      end ||
100     mon3_on()
101
102   | VREG1_ON then
103     vreg1_enable <- 1 ||
104
105     select vreg1_ready then
106       continue
107     end ||
108     mon3_on()
109
110   | INC then
111     if vok3_t = 0 then
112       count := (count +1 as 3 bits) ;
113       selvdd <- count ;
114       if prevstate = VREG1_ON then
115         select vreg1_ready then
116           continue
117         end
118       | prevstate = VREG2_ON then
119         select vreg2_ready then
```

```
120            continue
121          end
122        end ;
123        vok3 -> vok3_t
124      else
125        nextstate := DONE
126      end
127
128    | DONE then
129      stop := 1
130
131    else nextstate := MON_ON-- guard case
132    end
133 end
134
135 begin
136   loop  while stop = 0 then
137     setstate() ;
138     state_r := nextstate
139   end
140 end
```

A.1.2 State machine implementation 2

This is the balsa code for state machine implementation 2:

```
1 import [definitions]
2
3 procedure startup_state2 (
4   sync mon1_ready;
5   sync mon2_ready ;
6   sync mon3_ready ;
7   sync vok1 ;
8   sync vok2 ;
9   input vok3 : 1 bits ;
10  sync ref1_ready ;
11  sync ref2_ready ;
12  sync vreg1_ready ;
13  sync vreg2_ready ;
14  output mon1_enable : 1 bits ;
15  output mon2_enable : 1 bits ;
16  output mon3_enable : 1 bits ;
17  output ref1_enable : 1 bits ;
18  output ref2_enable : 1 bits ;
19  output vreg1_enable: 1 bits ;
20  output vreg2_enable : 1 bits ;
```

```
21   output selvdd : 3 bits
22 ) is
23
24 variable state_r : State
25 variable nextstate : State
26 variable prevstate : State
27 variable count : 3 bits
28
29 variable vok1_t : 1 bits
30 variable vok2_t : 1 bits
31 variable vok3_t : 1 bits
32 variable stop : 1 bits
33 variable vokmutex : 2 bits
34
35 shared mon_off is
36 begin
37   mon1_enable <- 0 ||
38   mon2_enable <- 0
39 end
40
41 shared mon3_on is
42 begin
43   select mon3_ready then
44     continue
45   end ;
46   vok3 -> then
47     vok3_t := vok3
48   end ;
49   prevstate := nextstate ;
50   nextstate := INC
51 end
52
53 procedure setstate is
54 begin
55   case state_r of
56
57     MON_ON then
58     mon1_enable <- 1 ||
59     mon2_enable <- 1 ||
60     select mon1_ready then
61       continue
62     end ||
63     select mon2_ready then
64       continue
65     end ||
66     nextstate := MON_SELECT
67
68   |     MON_SELECT then
69     arbitrate vok2 then
```

```
70         if vokmutex = 0 then
71           vokmutex := 2
72         end
73       |     vok1 then
74         if vokmutex = 0 then
75           vokmutex := 1
76         end
77       end;
78       if vokmutex = 1 then
79         nextstate := REF1_ON
80       | vokmutex = 2 then
81         nextstate := REF2_ON
82       end
83
84     | REF1_ON then
85       mon_off() ;
86       ref1_enable <- 1 ||
87       select ref1_ready then
88         nextstate := VREG1_ON
89       end
90
91     | REF2_ON then
92       mon_off() ;
93       ref2_enable <- 1 ||
94       select ref2_ready then
95         nextstate := VREG2_ON
96       end
97
98     | VREG2_ON then
99       vreg2_enable <- 1 ||
100      mon3_enable <- 1 ||
101      select vreg2_ready then
102        continue
103      end ||
104      mon3_on()
105
106    | VREG1_ON then
107      vreg1_enable <- 1 ||
108      mon3_enable <- 1 ||
109      select vreg1_ready then
110        continue
111      end ||
112      mon3_on()
113
114    | INC then
115      if vok3_t = 0 then
116        count := (count +1 as 3 bits) ;
117        selvdd <- count   ;
118        if prevstate = VREG1_ON then
```

```
        select vreg1_ready then
          continue
        end
      | prevstate = VREG2_ON then
        select vreg2_ready then
          continue
        end
      end ;
      vok3 -> vok3_t
    else
      nextstate := DONE
    end

  | DONE then
    stop := 1

  else nextstate := MON_ON-- guard case
  end
end

begin
  loop  while stop = 0 then
    setstate() ;
    state_r := nextstate
  end
end
```

A.1.3 State machine implementation 3

This is the balsa code for state machine implementation 3:

```
import [definitions]

procedure startup_state3 (
  sync mon1_ready;
  sync mon2_ready ;
  sync mon3_ready ;
  input vok : 1 bits ;
  input vok3 : 1 bits ;
  sync ref1_ready ;
  sync ref2_ready ;
  sync vreg1_ready ;
  sync vreg2_ready ;
  output mon1_enable : 1 bits ;
  output mon2_enable : 1 bits ;
  output mon3_enable : 1 bits ;
```

```
16   output ref1_enable : 1 bits ;
17   output ref2_enable : 1 bits ;
18   output vreg1_enable: 1 bits ;
19   output vreg2_enable : 1 bits ;
20   output selvdd : 3 bits
21 ) is
22
23 variable state_r : State
24 variable nextstate : State
25 variable prevstate : State
26 variable count : 3 bits
27
28 variable vok3_t : 1 bits
29 variable stop : 1 bits
30
31 shared mon_off is
32 begin
33   mon1_enable <- 0 ||
34   mon2_enable <- 0
35 end
36
37 shared mon3_on is
38 begin
39   select mon3_ready then
40     continue
41   end ;
42   vok3 -> then
43     vok3_t := vok3
44   end ;
45   prevstate := nextstate ;
46   nextstate := INC
47 end
48
49 procedure setstate is
50 begin
51   case state_r of
52
53     MON_ON then
54     mon1_enable <- 1 ||
55     mon2_enable <- 1 ||
56     select mon1_ready then
57       continue
58     end ||
59     select mon2_ready then
60       continue
61     end ||
62     nextstate := MON_SELECT
63
64   | MON_SELECT then
```

```
 65      vok -> then
 66        case vok of
 67          1 then
 68            nextstate := REF2_ON
 69        | 0 then
 70            nextstate := REF1_ON
 71        end
 72      end
 73
 74    | REF1_ON then
 75      mon_off() ;
 76      ref1_enable <- 1 ||
 77      select ref1_ready then
 78        nextstate := VREG1_ON
 79      end
 80
 81    | REF2_ON then
 82      mon_off() ;
 83      ref2_enable <- 1 ||
 84      select ref2_ready then
 85        nextstate := VREG2_ON
 86      end
 87
 88    | VREG2_ON then
 89      vreg2_enable <- 1 ||
 90      mon3_enable <- 1 ||
 91      select vreg2_ready then
 92        continue
 93      end ||
 94      mon3_on()
 95
 96    | VREG1_ON then
 97      vreg1_enable <- 1 ||
 98      mon3_enable <- 1 ||
 99      select vreg1_ready then
100        continue
101      end ||
102      mon3_on()
103
104    | INC then
105      if vok3_t = 0 then
106        count := (count +1 as 3 bits) ;
107        selvdd <- count ;
108        if prevstate = VREG1_ON then
109          select vreg1_ready then
110            continue
111          end
112        | prevstate = VREG2_ON then
113          select vreg2_ready then
```

```
114           continue
115         end
116       end ;
117       vok3 -> vok3_t ;
118     else
119       nextstate := DONE
120     end
121
122   | DONE then
123     stop := 1
124
125   else nextstate := MON_ON-- guard case
126   end
127 end
128
129 begin
130   loop while stop = 0 then
131     setstate() ;
132     state_r := nextstate
133   end
134 end
```

A.2 Optimized implementations

A.2.1 Optimized implementation 1

This is the balsa code for optimized implementation 1:

```
1  import [definitions]
2
3  procedure startup_simple1 (
4    sync mon1_ready;
5    input vok1 : 1 bits ;
6    output mon1_enable : 1 bits ;
7    sync mon2_ready ;
8    input vok2 : 1 bits ;
9    output mon2_enable : 1 bits ;
10   sync mon3_ready ;
11   input vok3 : 1 bits ;
12   output mon3_enable : 1 bits ;
13   sync ref1_ready ;
14   output ref1_enable : 1 bits ;
15   sync ref2_ready ;
```

```
16   output ref2_enable : 1 bits ;
17   sync vreg1_ready ;
18   output vreg1_enable: 1 bits ;
19   sync vreg2_ready ;
20   output vreg2_enable : 1 bits ;
21   output selvdd : 3 bits
22 ) is
23
24 variable vok1_t : 1 bits
25 variable vok2_t : 1 bits
26 variable vok3_t : 1 bits
27 variable count : 3 bits
28
29 shared inc is
30 begin
31   if count < 7 then
32     count := (count +1 as 3 bits)
33   end
34 end
35
36 begin
37   mon1_enable <- 1 ||
38   mon2_enable <- 1 ||
39   select mon1_ready then
40     continue
41   end ||
42   select mon2_ready then
43     continue
44   end ;
45   loop while (vok1_t or vok2_t) = 0 then
46     vok1 -> vok1_t ||
47     vok2 -> vok2_t
48   end ;
49   if vok2_t = 1 then
50     ref2_enable <- 1 ||
51     select ref2_ready then
52       continue
53     end ;
54     vreg2_enable <- 1
55   | vok1_t = 1 then
56     ref1_enable <- 1 ||
57     select ref1_ready then
58       continue
59     end ;
60     vreg1_enable <- 1
61   end ;
62   mon1_enable <- 0 ||
63   mon2_enable <- 0 ;
64   mon3_enable <- 1 ||
```

```
65    select mon3_ready then
66      continue
67    end ;
68    loop while vok3_t = 0 then
69      inc() ;
70      selvdd <- count ;
71      if vok2_t = 1 then
72        select vreg2_ready then
73          continue
74        end
75      | vok1_t = 1 then
76        select vreg1_ready then
77          continue
78        end
79      end ;
80      vok3 -> vok3_t
81    end
82  end
```

A.2.2 Optimized implementation 2

This is the balsa code for optimized implementation 1:

```
1  import [definitions]
2
3  procedure startup_simple1 (
4    sync mon1_ready;
5    --input vok1 : 1 bits ;
6    sync vok1 ;
7    output mon1_enable : 1 bits ;
8    sync mon2_ready ;
9    --input vok2 : 1 bits ;
10   sync vok2 ;
11   output mon2_enable : 1 bits ;
12   sync mon3_ready ;
13   input vok3 : 1 bits ;
14   output mon3_enable : 1 bits ;
15   sync ref1_ready ;
16   output ref1_enable : 1 bits ;
17   sync ref2_ready ;
18   output ref2_enable : 1 bits ;
19   sync vreg1_ready ;
20   output vreg1_enable: 1 bits ;
21   sync vreg2_ready ;
22   output vreg2_enable : 1 bits ;
23   output selvdd : 3 bits
```

```
24 ) is
25
26 variable vok3_t : 1 bits
27 variable count : 3 bits
28 variable vokmutex : 2 bits
29
30 shared inc is
31 begin
32   if count < 7 then
33     count := (count +1 as 3 bits)
34   end
35 end
36
37 begin
38   mon1_enable <- 1 ||
39   mon2_enable <- 1 ||
40   select mon1_ready then
41     continue
42   end ||
43   select mon2_ready then
44     continue
45   end ;
46   arbitrate vok2 then
47     if vokmutex = 0 then
48       vokmutex := 2
49     end
50   |      vok1 then
51     if vokmutex = 0 then
52       vokmutex := 1
53     end
54   end;
55   if vokmutex = 1 then
56     ref1_enable <- 1 ||
57     select ref1_ready then
58       continue
59     end ;
60     vreg1_enable <- 1
61   | vokmutex = 2 then
62     ref2_enable <- 1 ||
63     select ref2_ready then
64       continue
65     end ;
66     vreg2_enable <- 1
67   end ;
68   mon1_enable <- 0 ||
69   mon2_enable <- 0 ;
70   mon3_enable <- 1 ||
71   select mon3_ready then
72     continue
```

```
73   end ;
74   loop while vok3_t = 0 then
75     inc() ;
76     selvdd <- count ;
77     if vokmutex = 2 then
78       select vreg2_ready then
79         continue
80
81       end
82       --selvdd_inc()
83     | vokmutex = 1 then
84       select vreg1_ready then
85         continue
86       end
87     end ;
88     vok3 -> vok3_t ;
89   end
90 end
```

A.2.3 Optimized implementation 3

This is the balsa code for optimized implementation 1:

```
1  import [definitions]
2
3  procedure startup_simple3 (
4    sync mon1_ready;
5    input vok : 1 bits ;
6    output mon1_enable : 1 bits ;
7    sync mon2_ready ;
8    output mon2_enable : 1 bits ;
9    sync mon3_ready ;
10   input vok3 : 1 bits ;
11   output mon3_enable : 1 bits ;
12   sync ref1_ready ;
13   output ref1_enable : 1 bits ;
14   sync ref2_ready ;
15   output ref2_enable : 1 bits ;
16   sync vreg1_ready ;
17   output vreg1_enable: 1 bits ;
18   sync vreg2_ready ;
19   output vreg2_enable : 1 bits ;
20   output selvdd : 3 bits
21 ) is
22
23 variable vok1_t : 1 bits
```

```
24 variable vok2_t : 1 bits
25 variable vok3_t : 1 bits
26 variable count : 3 bits
27
28 shared inc is
29 begin
30   if count < 7 then
31     count := (count +1 as 3 bits)
32   end
33 end
34
35 begin
36   mon1_enable <- 1 ||
37   mon2_enable <- 1 ||
38   select mon1_ready then
39     continue
40   end ||
41   select mon2_ready then
42     continue
43   end ;
44   vok -> then
45     case vok of
46       1 then
47         vok2_t := 1
48     | 0 then
49         vok1_t := 1
50     end
51   end ;
52   if vok2_t = 1 then
53     ref2_enable <- 1 ||
54     select ref2_ready then
55       continue
56     end ;
57     vreg2_enable <- 1
58   | vok1_t = 1 then
59     ref1_enable <- 1 ||
60     select ref1_ready then
61       continue
62     end ;
63     vreg1_enable <- 1
64   end ;
65   mon1_enable <- 0 ||
66   mon2_enable <- 0 ;
67   mon3_enable <- 1 ||
68   select mon3_ready then
69     continue
70   end ;
71   loop while vok3_t = 0 then
72     inc();
```

```
73      selvdd <- count ;
74      if vok2_t = 1 then
75        select vreg2_ready then
76          continue
77        end
78        --selvdd_inc()
79      | vok1_t = 1 then
80        select vreg1_ready then
81          continue
82        end
83      end ;
84      vok3 -> vok3_t ;
85    end
86  end
```

Appendix B

Synchronous startup logic

This is the verilog code for the synchronous startup logic.

```
1  `timescale 1ns/1ps
2
3  module startup_logic2(/*AUTOARG*/
4    // Outputs
5    mon1_enable_r, mon2_enable_r, mon3_enable_r, ref1_enable_r,
6    ref2_enable_r, regul1_enable_r, regul1_selvdd_r,
       regul2_enable_r,
7    regul2_selvdd_r,
8    // Inputs
9    clk, porneg, mon1_vok, mon1_ready, mon2_vok, mon2_ready,
       mon3_vok,
10   mon3_ready, ref1_ready, ref2_ready, regul1_ready, regul2_ready
11   );
12
13     //parameter VThreshold = 1000; //mV
14
15 `include "startup_parameters.v"
16
17
18     input clk;//for testing purposes, internal oscillator
19
20     input porneg;//inputs from power on reset
21     input mon1_vok;//inputs from voltage monitor 1
22     input mon1_ready;
23     input mon2_vok;//inputs from voltage monitor 2
24     input mon2_ready;
25     input mon3_vok;//inputs from voltage monitor 3
26     input mon3_ready;
```

```
27   input ref1_ready;//inputs from voltage reference 1
28   input ref2_ready; //inputs from voltage reference 2
29   input regul1_ready; //input from voltage regultaror 1
30   input regul2_ready; //input from voltage regultaror 2
31
32   output mon1_enable_r;//outputs to voltage monitor 1
33   output mon2_enable_r;//outputs to voltage monitor 2
34   output mon3_enable_r; //outputs to voltage monitor 3
35   output ref1_enable_r;//outputs to voltage reference 1
36   output ref2_enable_r; //outputs to voltage reference 2
37   output regul1_enable_r;//output to voltage regultaror 1
38   output [AVR32_VREG_SELVDD_MSB:0] regul1_selvdd_r;
39   output            regul2_enable_r;//output to voltage
       regultaror 2
40   output [AVR32_VREG_SELVDD_MSB:0] regul2_selvdd_r;
41
42
43  /*AUTOREG*/
44
45  /*AUTOWIRE*/
46
47  reg                                 mon1_enable_r;
48  reg                                 mon2_enable_r;
49  reg                                 mon3_enable_r;
50  reg                                 ref1_enable_r;
51  reg                                 ref2_enable_r;
52  reg                                 regul1_enable_r;
53  wire [AVR32_VREG_SELVDD_MSB:0]       regul1_selvdd_r;
54  reg                                 regul2_enable_r;
55  wire [AVR32_VREG_SELVDD_MSB:0]       regul2_selvdd_r;
56
57
58  reg                                 mon1_enable;
59  reg                                 mon2_enable;
60  reg                                 mon3_enable;
61  reg                                 ref1_enable;
62  reg                                 ref2_enable;
63  reg                                 regul1_enable;
64
65  reg                                 regul2_enable;
66
67
68
69
70  reg                                 mon1_ready_m;
71  reg                                 mon1_ready_r;
72  reg                                 mon1_vok_m;
73  reg                                 mon1_vok_r;
74  reg                                 mon2_ready_m;
```

```
75   reg                                    mon2_ready_r;
76   reg                                    mon2_vok_m;
77   reg                                    mon2_vok_r;
78   reg                                    mon3_ready_m;
79   reg                                    mon3_ready_r;
80   reg                                    mon3_vok_m;
81   reg                                    mon3_vok_r;
82   reg                                    ref1_ready_m;
83   reg                                    ref1_ready_r;
84   reg                                    ref2_ready_m;
85   reg                                    ref2_ready_r;
86
87   reg                                    regul1_ready_m;
88   reg                                    regul1_ready_r;
89   reg                                    regul2_ready_m;
90   reg                                    regul2_ready_r;
91
92
93   reg [STARTUP_FSM_MSB:0]                state_r;
94   reg [STARTUP_FSM_MSB:0]               state_nxt;
95
96   reg [AVR32_VREG_SELVDD_MSB:0]          selvdd_r;
97   reg [AVR32_VREG_SELVDD_MSB:0]          selvdd;
98
99
100  // Synchronisers for async input signals
101  always @(posedge clk or negedge porneg) begin
102    if(porneg == 1'b0) begin
103      /*AUTORESET*/
104      // Beginning of autoreset for uninitialized flops
105      mon1_ready_m <= 1'h0;
106      mon1_ready_r <= 1'h0;
107      mon1_vok_m <= 1'h0;
108      mon1_vok_r <= 1'h0;
109      mon2_ready_m <= 1'h0;
110      mon2_ready_r <= 1'h0;
111      mon2_vok_m <= 1'h0;
112      mon2_vok_r <= 1'h0;
113      mon3_ready_m <= 1'h0;
114      mon3_ready_r <= 1'h0;
115      mon3_vok_m <= 1'h0;
116      mon3_vok_r <= 1'h0;
117      ref1_ready_m <= 1'h0;
118      ref1_ready_r <= 1'h0;
119      ref2_ready_m <= 1'h0;
120      ref2_ready_r <= 1'h0;
121      regul1_ready_m <= 1'h0;
122      regul1_ready_r <= 1'h0;
123      regul2_ready_m <= 1'h0;
```

```
124        regul2_ready_r <= 1'h0;
125        // End of automatics
126      end
127      else begin
128        mon1_vok_m     <= mon1_vok;
129        mon1_vok_r     <= mon1_vok_m;
130        mon1_ready_m <= mon1_ready;
131        mon1_ready_r <= mon1_ready_m;
132
133        mon2_vok_m     <= mon2_vok;
134        mon2_vok_r     <= mon2_vok_m;
135        mon2_ready_m <= mon2_ready;
136        mon2_ready_r <= mon2_ready_m;
137
138
139        mon3_vok_m     <= mon3_vok;
140        mon3_vok_r     <= mon3_vok_m;
141        mon3_ready_m <= mon3_ready;
142        mon3_ready_r <= mon3_ready_m;
143
144        ref1_ready_m     <= ref1_ready;
145        ref1_ready_r     <= ref1_ready_m;
146
147
148        ref2_ready_m     <= ref2_ready;
149        ref2_ready_r     <= ref2_ready_m;
150
151        regul1_ready_m <= regul1_ready;
152        regul1_ready_r <= regul1_ready_m;
153
154        regul2_ready_m <= regul2_ready;
155        regul2_ready_r <= regul2_ready_m;
156
157
158      end
159    end
160
161    // Registers for state_r and analog control signals (to ensure
         glitch free control)
162    always @(posedge clk or negedge porneg) begin
163      if(porneg == 1'b0) begin
164        state_r <= RESET;
165        /*AUTORESET*/
166        // Beginning of autoreset for uninitialized flops
167        mon1_enable_r <= 1'h0;
168        mon2_enable_r <= 1'h0;
169        mon3_enable_r <= 1'h0;
170        ref1_enable_r <= 1'h0;
171        ref2_enable_r <= 1'h0;
```

```
172        regul1_enable_r <= 1'h0;
173        regul2_enable_r <= 1'h0;
174        selvdd_r <= {(1+(AVR32_VREG_SELVDD_MSB)){1'b0}};
175        // End of automatics
176      end
177      else begin
178        state_r <= state_nxt;
179        mon1_enable_r    <=    mon1_enable;
180        mon2_enable_r    <=    mon2_enable;
181        mon3_enable_r    <=    mon3_enable;
182        ref1_enable_r    <=    ref1_enable;
183        ref2_enable_r    <=    ref2_enable;
184        regul1_enable_r <=    regul1_enable;
185        regul2_enable_r <=    regul2_enable;
186        selvdd_r <=    selvdd;
187
188      end
189    end
190
191    assign vddio1_good = mon1_ready_r && mon1_vok_r;
192    assign vddio2_good = mon2_ready_r && mon2_vok_r;
193
194    assign regul1_selvdd_r = selvdd_r;
195    assign regul2_selvdd_r = selvdd_r;
196
197
198
199    always @* begin
200      mon1_enable    =   0;
201      mon2_enable    =   0;
202      mon3_enable    =   0;
203      ref1_enable    =   0;
204      ref2_enable    =   0;
205      regul1_enable =   0;
206      regul2_enable =   0;
207      selvdd =   0;
208      state_nxt = RESET;
209      case(state_r)
210        RESET: begin
211          state_nxt = MONITOR12_ON;
212        end
213
214        MONITOR12_ON: begin
215          mon1_enable = 1;
216          mon2_enable = 1;
217          // Wait for both monitor to be ready
218          if(mon1_ready_r && mon2_ready_r) begin
219            state_nxt = MONITOR12_SELECT;
220          end
```

```
221          else begin
222            state_nxt = MONITOR12_ON;
223          end
224
225        end
226
227        MONITOR12_SELECT: begin
228          mon1_enable = 1;
229          mon2_enable = 1;
230          if(vddio2_good) begin
231            state_nxt = REF2_ON;
232          end
233          else if (vddio1_good) begin
234            state_nxt = REF1_ON;
235          end
236          else begin
237            // Stay in this state
238            state_nxt = MONITOR12_SELECT;
239          end
240        end
241
242        REF1_ON: begin
243          ref1_enable = 1;
244          if(ref1_ready_r)
245            state_nxt = VREG1_ON;
246          else
247            state_nxt = REF1_ON;
248        end
249
250
251        REF2_ON: begin
252          ref2_enable = 1;
253          if(ref2_ready_r)
254            state_nxt = VREG2_ON;
255          else
256            state_nxt = REF2_ON;
257        end
258
259
260        VREG1_ON: begin
261          ref1_enable = 1;
262          regul1_enable = 1;
263          mon3_enable = 1;
264          selvdd = VDD_600;
265          if( mon3_ready_r)
266            state_nxt = INC;
267          else
268            state_nxt = VREG1_ON;
269
```

```
270        end
271
272        VREG2_ON: begin
273          ref2_enable = 1;
274          regul2_enable = 1;
275          mon3_enable = 1;
276          selvdd = VDD_600;
277          if( mon3_ready_r)
278            state_nxt = INC;
279          else
280            state_nxt = VREG2_ON;
281
282        end
283
284        INC: begin
285          mon3_enable = 1;
286          ref1_enable= ref1_enable_r;
287          ref2_enable= ref2_enable_r;
288          regul1_enable = regul1_enable_r;
289          regul2_enable = regul2_enable_r;
290          // We take a shortcut here - The two regulator are not
               supposed to be ready at the same time
291          if((regul1_ready_r || regul2_ready_r) && mon3_ready_r &&
                !mon3_vok_r && (selvdd_r != VDD_1800) && (selvdd_r
               != VDD_2000)) begin
292            selvdd = selvdd_r + 1;
293            state_nxt = INC2;
294          end
295          else begin
296            selvdd = selvdd_r;
297            state_nxt = INC;
298          end
299        end // case: INC
300        INC2: begin
301          // we wait before
302          mon3_enable = 1;
303          ref1_enable= ref1_enable_r;
304          ref2_enable= ref2_enable_r;
305          regul1_enable = regul1_enable_r;
306          regul2_enable = regul2_enable_r;
307          selvdd = selvdd_r;
308          state_nxt = INC;
309        end
310        default: begin
311
312
313
314
315        end
```

```
316         endcase
317     end
318 endmodule
```