DISSERTATION

Design of an Asynchronous Processor Based on Code Alternation Logic – Treatment of Non-Linear Data Paths

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften
unter der Leitung von

A.O.UNIV.-PROF. DIPL.-ING. DR. A. STEININGER

Inst.-Nr. E182/2
Institut für Technische Informatik
Embedded Computing Systems Group

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

DIPL.-ING. MARTIN DELVAI

Matr.-Nr. 9325765

Theodor Kramer-Str. 8/2/81
1220 Wien

Wien, im Dezember 2004 _____

Kurzfassung

Das synchrone Designparadigma sieht sich zunehmend mit immer größer werdenden Problemen konfrontiert: Die Illusion, dass alle Komponenten eines Chips synchron arbeiten, kann aufgrund steigender Taktraten und der immer größeren Chipflächen nur mit extremen (Hardware-) Aufwand aufrecht erhalten werden. Auch Leistungsverbrauch und Wärmeentwicklung stellen zunehmend kritische Faktoren dar. Vielversprechende Alternativen bilden asynchrone Designmethoden: Diese benötigen kein globales Taktsignal, sondern basieren auf lokalen Kontrollmechanismen und arbeiten ereignisgesteuert, wodurch die Verlustleistung erheblich reduziert werden kann.
Aus diesem Grund wurden in der vorliegenden Arbeit verschiedene Methoden zum Entwurf von asynchronen Schaltungen analysiert und die *Code Alternation Logic* (CAL) für die spätere Implementierung eines Prozessorprototypens ausgewählt. Dieser Ansatz kodiert die notwendigen Informationen zur Datenflusskontrolle in den Daten selbst: Für die logischen Zustände *LOW* und *HIGH* gibt es jeweils zwei unterschiedliche Darstellungen; wir sprechen in diesem Zusammenhang auch von der Phase eines Signals. Aufeinanderfolgende Daten werden in unterschiedlichen Phasen kodiert, wodurch Schaltungseinheiten diese voneinander unterscheiden und die dazugehörige Information eindeutig zuordnen können. Nichtlineare Schaltungsstrukturen stören jedoch den homogen alternierenden Datenfluss und beeinträchtigen somit die Datenflusskontrolle. Der Fokus dieser Dissertation liegt in der Behandlung solcher nichtlinearen Strukturen. Grundsätzlich werden zwei Arten von Nichtlinearität unterschieden: Vorwärts- und Rückkopplungsschleifen einerseits sowie selektive Schaltungskomponenten andererseits. Erstere führen dazu, dass Eingänge an Schaltungselementen in unterschiedlichen Phasen kodiert sind, obwohl sie demselben Kontext angehören. Um dennoch die korrekte Funktionalität der Schaltung zu gewährleisten, müssen gezielt sogenannte Phasenumkehrer in die Schaltung eingefügt werden. In dieser Arbeit wird gezeigt, dass deren Platzierung nicht ausschließlich von der Schaltungstopologie, sondern auch von der Initialisierung abhängt. Weiters bewirken nichtlineare Datenpfade eine Selbstregulierung der Schaltvorgänge, wodurch sich eine starke Abhängigkeit der Verarbeitungsgeschwindigkeit von der Initialisierung ergibt.
Bei der zweiten Art von Nichtlinearität handelt es sich um Schaltungselemente, die nur eine Teilmenge ihrer Eingänge benötigen um, den Ausgang zu bilden – z.B. Multiplexer – oder nur eine Teilmenge der Ausgänge setzen – z.B. Demultiplexer. Diese Komponenten bewirken, dass die Schaltungsteile, die an den nicht selektierten Eingängen bzw. Ausgängen angeschlossen sind, ihre Phasensynchronisation mit der restlichen Schaltung verlieren. Es wird gezeigt, dass dieses Problem durch Synchronisationsschaltungen oder durch die Verwendung von Platzhalterdaten vermieden werden kann.
Die Erkenntnisse dieser Arbeit wurden durch Simulationen bestätigt und im Design eines funktionierenden Hardwareprototypens für einen asynchronen Prozessor auch praktisch verifiziert.

Abstract

The synchronous design paradigm faces some limitations: The illusion that all components inside a chip receive the (active) clock edge at the same point in time can be sustained only under a considerable hardware effort. In addition, the power consumption of a CMOS circuit is proportional to the applied clock frequency – thus the increasing clock frequency coupled with today's high integration density escalates the heating problem. In contrast, asynchronous design methods promise to solve all these problems in a natural manner: On the one hand they require only (local) handshake mechanisms instead of a global time reference. On the other hand, asynchronous methods are event-driven – hence they consume energy only when useful work has to be performed, in contrast to synchronous circuit, which are permanently triggered by the clock signal.

With this motivation several asynchronous design methods were analyzed. A specific method, namely the *Code Alternation Logic* was selected to implement an asynchronous processor prototype. The principle of this approach is to encode the information necessary for the data flow control in the processed data itself, by defining two disjoint representations for the logical *HIGH* and two for the representation of *LOW*. To highlight that from a logical point of view both representants carry the same information we say that a signal can be coded in different phases. Since subsequent data waves are coded in alternating phases, each component inside the circuit can distinguish incoming data and associate it to a specific context. However, non-linear circuit structures disturb this alternating sequence of data waves and therefore affect the data flow control. The focus of this thesis is placed on such non-linear structures. Here, two types of non-linearity are distinguished: forward/feedback paths and selecting nodes.

The first one causes that components receive signals which belong to the same context, but are coded in different phases. To overcome this problem we have to place phase inverters in a selective manner. In this work it is shown, that the placement not only depends on the circuit topology, but also on its initialization. Furthermore it is pointed out, that feedback/forward paths cause a structural regulation of the data flow. As a consequence, the performance of a circuit depends strongly on its initialization.

The second source of non-linearity is constituted by nodes, which require only a subset of their inputs to generate the output (multiplexer, e.g.) and/or nodes which set only a subset of their outputs (de-multiplexer, e.g.). Consequently the parts of the circuit, which are connected to the unselected inputs/outputs of selecting nodes, loose their phase synchronization with the remaining circuit. This difficulty can be solved by using synchronizer circuits or "dummy" data.

The findings of this thesis were confirmed by simulation and verified by the implementation of a hardware prototype of an asynchronous processor.

Contents

List of Figures

Chapter 1

Introduction

The history of asynchronous logic design is quite long. Asynchronous design methods date back to the 1950s and two people in particular shall be mentioned, namely David A. Huffman [47] and David E. Muller [76]. Nevertheless clocked circuits dominate the market of digital circuits today, while a small segment is reserved for asynchronous chips [112] only. The triumphal procession of the synchronous approach is based on its discretization of time: This facilitates the description of the circuit behavior – the designer hypothesizes that all operations within the circuit finish in time to be sampled with the next clock edge. Hence neither glitches, signal delay and skew nor physical properties such as driver power or the real duration of a logical operation have to be considered during functional description. This circumstance yield to shorter design cycles and paved the way for *Hardware Description Languages* (HDL's) such as Verilog [3] and VHDL [4], boosting the productivity of chip designers again. Furthermore, the observation of the values at well-known discrete points in time facilitates the simulation and debug process of a design. In addition design verification "*... becomes a matter of checking the delays in the combinatorial logic functions between the (clocked) registers. This is a straightforward process ...*" [39], which can be automated.
The synchronous design style in conjunction with high level hardware description languages, elaborated tool and technological advances concerning integration density has enabled great strides to be taken in the design and performance of computers. In 1965 Gordon Moore predicted that chip density (and performance) doubles every eighteen months [74]. *"In 24 years the number of transistors on processor chips has increased by a factor of almost 2400, from 2300 on the Intel 4004 in 1971 to 5.5 million on the Pentium Pro in 1995 (doubling roughly every two years")* [27]. Moore's observation stays true until today (2004) and speaking at the International Solid-States Circuits Conference (ISSCC 2003), Moore has predicted, that this trend will proceed in the next decade [73]. As a result, processor cores clocked with several GHz and built out of more than 400 millions transistors [48][26] are standard for use in personal computers, today.

However, during the last decade there has been a revival in research on asynchronous circuits [89][39] – the intensive research activity is reflected by the exploding number

publications concerning asynchronous logic in the last years (see Figure 1.1).

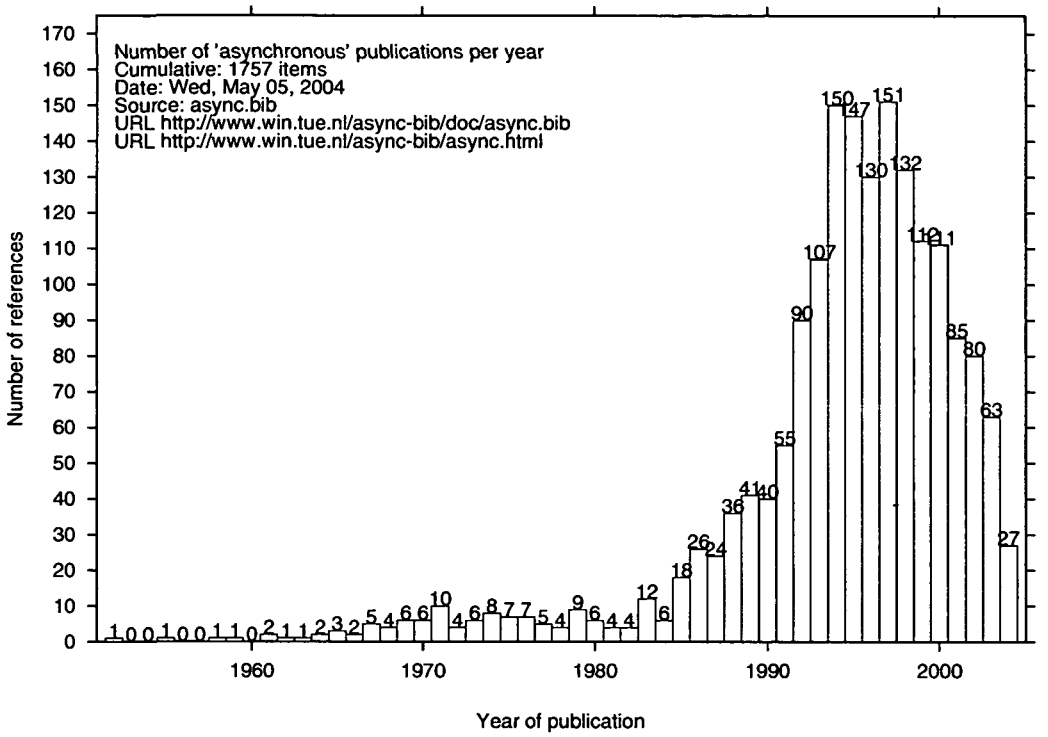


Figure 1.1. Number of "Asynchronous" Publications per Year [88]

1.1 Motivation

What is the motivation behind moving away from a well established and approved design methodology? With the improvements made in the last decades several already existing problems concerning the chip design style became increasingly critical and will be further aggravated by each new technology step. One root of the problem is that the signal propagation is limited by the speed of the light[1]. As soon as the signal propagation delay becomes a significant part of the clock period – clock frequencies beyond one GHz imply clock periods under one nanosecond – circuit designers have to pay a heavy price to keep up the illusion that all components receive the rising edge of the clock signal at the same time [102].
Another critical issue concerns power consumption [19]. The clock signal triggers the components always, regardless of whether they have to do useful work or not – increasing unnecessarily the energy consumption. Furthermore, the miniaturization aggravates the situation by escalating the heat density inside high performance chips.
In addition the combination of larger chips and faster transistors caused a fundamental change in the relation between gate and wire delay: In today's sub-micron designs, wire delays and not gate delays are the dominant factors for circuit timing (see figure 1.2).

[1]To be more precise electrical signals travel on chips with 2/3 of the speed of the light

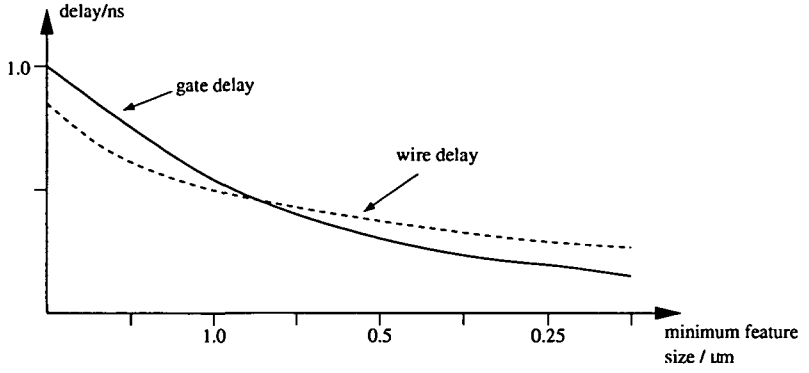


Figure 1.2. Gate vs. Interconnect Delay [103]

Thus a reliable verification of a circuit can be done after place & route only, and so it is performed at a very late point in time in the design process. In practice, however, timing problems often necessitate changes in the functional design. In this way the separation of functional design and timing analysis causes unnecessary long iteration cycles.

The asynchronous approach seems to solve most of the problems in a natural way: Being event-driven, asynchronous circuits (i) perform operations only when required, reducing the power consumption, (ii) do not require a global time reference, disarming the problems concerning clock distribution and signal skew.
As a fetch-ahead to the following chapters, important properties of asynchronous circuits, which can be advantages in some areas [104], are listed as follows:

- Achieve average case performance, [66][65][119][120]
  ... operating speed is determined by actual local delays rather than the global worst-case latency.
- Low power consumption, [37][36][81][12][11]
  ... consume power only when needed.
- Provide easy modular composition, [75][60][107][9]
  ... asynchronous components are combined with simple handshake protocols.
- Avoid clock distribution and clock skew problems,
  ... because there is no global clock.
- Lower electro magnetic emission and noise, [70][86][11]
  ... local triggered registers tend to be active at any point in time.
- Variations in fabrication process parameters, temperature and supply voltage are not as critical as in synchronous designs, [80][79][64]
  ... because the timing is based on the relationship of the delays instead of on the absolute values.

Convinced by the potential of the asynchronous design style our department started its research activity in this field four years ago. The aim was not to invent a new method, but to provide an in-deep analysis of one existing design style. We have chosen the four phase logic approach [69][21] due to the fact that it allows to build completely delay-insensitive circuits on gate level and it does not require a neutral state between valid data word such as the Null Convention Logic [31]. On account of the four phase logic alternates the data encoding style within consecutive data words, we call this logic CAL (Code Alternation Logic).
To perform our analysis, we have first developed a reference object, namely a synchronous processor core called *SPEAR* (Scalable Processor for Embedded Applications in Real-time Environments). In the second step we re-designed it using the CAL approach. This not only opened the way to perform conventional analyses but it also allows to compare the asynchronous processor with the synchronous one concerning i.e. speed, fault-tolerance, testability and so on.

1.2 Contribution and Objectives

The basic principle of CAL is to encode subsequent data with alternating phases. This allows all components in a CAL circuit to judge consistency of their input data and thus to decide if a new output has to be generated or not. In regular circuit structures such as in a linear pipeline, the aforementioned data flow control mechanism can be applied in a straight forward manner. In contrast to non-linear circuit structures, this data flow control mechanism is a delicate issue, as it is extremely prone to deadlock.

The contribution of this thesis is an in-depth analysis of non-linear structures in conjunction with the CAL design style. We distinguish between two types of non-linearity: selecting nodes and feedback/forward paths.
The latter one requires the placement of additional phase inverters inside the circuit in order to guarantee consistently encoded input data for all components in the circuit. For this placement not only the steady state of a circuit must be considered, but also the dynamic effects have to be taken into account. We found that feedback/forward paths cause a *structural regulation* of the data flow. As a consequence the performance of a circuit depends on its initialization.
The second source of non-linearity is selecting nodes. These are (i) nodes, which require only a subset of their input to perform their operation or (ii) nodes, which only activate a dedicated subset of their outputs. We can observe that in both cases the nodes – and subsequently the related data paths – which are connected to the unused ports of selecting nodes may loose their (phase-) synchronization with the remaining circuit. As a consequence, the control flow must be adopted in an appropriate manner. Finally, the validity of these findings are demonstrated by implementing an asynchronous processor core with CAL.

1.3 Structure of the Thesis

After this short introduction in Chapter 2 we will investigate the fundamental problem concerning digital design, i.e. to determine whether data is ready to be read and to ensure that no data gets lost in the circuit. Two basic domains are proposed in which these problems can be treated, namely the time domain and the information domain. Subsequently we are going to analyze which parts and in which domains current approaches solve the fundamental design problems. The section concludes with a comparison of presented methods. In Chapter 3 focus of our attention will be placed to a specific design method named *Code Alternation Logic*(CAL). The data flow regulation principle of this approach as well as the implementation of its basic gates is shown. Since tools constitute a huge challenge with respect to asynchronous logic design, we will conclude this chapter presenting our design flow and explaining how we adapted the synchronous tools for our purpose. In Chapter 4 the reader will be introduced to the reference processor SPEAR and the target technology platform where we have implemented the processor core, namely Altera's *Quartus II* and its FPGA *APEX 20kC*. Since the SPEAR processor is a highly non linear device we analyze in the impact of non-linear structure on CAL circuits Chapter 5. The influence of forward and feedback path as well as the difficulties concerning selecting nodes are going to be analyzed in detail. With these findings we are able to implement the ASPEAR, which is illustrated in chapter 6. The thesis ends with a conclusion in Chapter 7.

Chapter 2

State of the Art

Circuit design styles can be classified into two major categories, namely synchronous and asynchronous. The first approach is based on one or more globally distributed periodic timing signals, called clocks, which sequence the circuit [20]. The asynchronous design style is a event-driven circuit design technique where, instead of the components sharing a common clock and exchanging data on clock edges, data is passed on as soon as it is available [32].Although the asynchronous design methods have been studied for many decades, today the clocked circuits dominate the market of digital circuits. However the synchronous design style faces some fundamental limits: Propagation of electrical signals on chips is bounded by the speed of light: As the chips get bigger and the clocks run faster, this physical restriction becomes more and more a crucial factor in the design process of synchronous chips [67]. Another critical aspect constitutes the power consumption: In CMOS circuits the dissipated energy is proportional to the switch activity – in synchronous circuits the gate activity is driven by the clock signals, independent from the fact if useful work has to be done or not. Possible options to solve these problems are *clock gating* [41], where unused parts of the circuit are temporally disabled, or *speed down the clock frequency* during idle states [49][25]. However, these are compromise solutions which aim to compensate the weak points of the synchronous design principle and which have to be paid in terms of recovery time and circuit overhead.
Being an event-driven method, the asynchronous design style promises to solve the mentioned difficulties by its nature. Motivated by this circumstance a lot of asynchronous design techniques were developed [42]. Though all approaches have the same underlying principle, namely being event driven, their concrete effectuations look completely different.

Furthermore many approaches deal with only one particular design aspect. Hence implementing a complete chip requires often a combination of methods. This makes it difficult to classify asynchronous circuits and to compare them with the synchronous design style.
For completeness we will mention a third design style, namely the *Globally-Asynchronous Locally-Synchronous*(GALS) approach. The fundamental idea of this approach is applied successfully to compose systems on higher abstraction levels,

connect a printer with a PC, e.g. As more and more components can be integrated on a single silicon die, this method becomes attractive even for VLSI designs [28][55]. However, this design style can be traced back to the previously mentioned styles and therefore it will not be considered separately.

All methods and design styles have one point in common: If we take a look from a more abstract point of view, we could recognize that all methods, including the synchronous approach, aim to solve the same problem, namely to ensure that all data is correctly processed by the circuit. We call this problem the *fundamental design problem.*
To be able to depict this problem in greater detail, we will first provide a system model in Section 2.1. Based on this definition we will figure out the fundamental design problem and deduce its root in Section 2.2. In Section 2.3 we will distinguish between two basic strategies, which deal with the fundamental design, namely the use of time or the use of information. With this theoretical background we are able to analyze and classify characteristic types of design approaches in Section 2.4. This chapter concludes with a comparison of the presented design styles.

2.1 System Model

2.1.1 Terminology

Terms such as *signal, vectors, bits, e.g.* are used in many different fields of applications. As a consequence these terms are interpreted in a slightly different manner depending on its context. Due to this common usage a discussion inside our department flared up about the exact meaning and interpretation of several expressions. Also the literature could not help to clarify the situation due to the fact that some terms are defined differently. It is for this reason that we devote a section to define the used *terminology.*

We call the input of a Boolean logic function an *input vector.* It is constituted by a number of *signals* – one for each input. The Boolean logic function defines a specific mapping from the input vector to an output signal. This mapping is implemented by a *logic function unit.* Often several Boolean logic functions are applied to the same input vector in parallel, creating several output signals with a common semantic context (data path elements like adder, e.g.). The term logic function units is used in a broader sense to describe the implementation of this set of Boolean functions as well.

We call the smallest unit of information conveyed on a signal a *bit*, and the (consistent) vector of bits conveyed on an input vector a *data word.* A signal can be physically represented by one or more *rails*, whose *logic levels* define the signal's *logic state.* The two mandatory logic states of a signal are "high (HI)"and "low (LO)", but states such as "NULL", "illegal"or "in transition"are conceivable and sometimes used as well. A *signal-level code* relates the logic levels of the rails – viewed as a vector that represents a signal – to the logic state of the corresponding signal. For the digital rails we consider that the logic level may either be "0" or "1". In the conventional single-rail encoding a

signal is represented by only one rail whose logic level is directly mapped to the signal state.

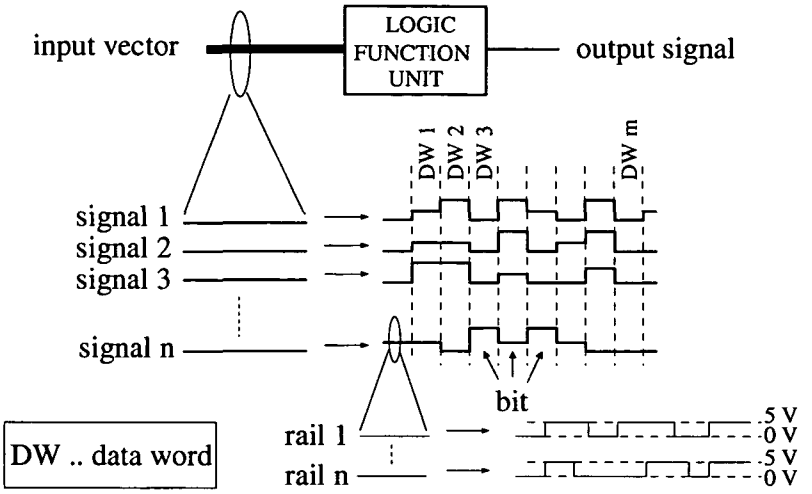


Figure 2.1. Terminology

We refer to an input vector as *consistent* at instant t_i, if the states of all its signals belong to the same context at instant t_i, i.e. if they represent one single valid data word, and inconsistent otherwise. We also call the involved signals consistent under this condition. We call a signal *valid* at instant t_i, if its state at instant t_i is the stable result of a logic operation performed on a consistent input vector, and invalid otherwise.

2.1.2 Data Flow

From the point of view of information flow every function unit FU is preceded by a data source SRC that provides the input vector for FU, and followed by a data sink SNK that further processes the output signal or vector of FU (maybe in context with the outputs of other function units). Both data sink SNK and source SRC represent an abstraction of the remaining circuit and may internally consist of further function units. We call an output bit b_y of FU *consumed* by the sink SNK at t_i, if b_y is still properly considered in the flow of information in SNK, regardless of whether b_y is overwritten by a subsequent bit b_z after t_i or not. Usually consumption implies the transfer of the information to some storage element.

An information flow is termed *lossless*, if all pertaining bits are properly consumed at all instances t_i. Also a signal path P_k is called lossless, if the information flow along P_k is lossless. To guarantee *losslessness*, SRC and SNK, have to be appropriately coordinated.

2.1.3 Timed Data Flow Relation

Considering the temporal relations and delays involved in the data transfer between SRC and SNK, we have to extend our model by timing issues. Figure 2.3 illustrates

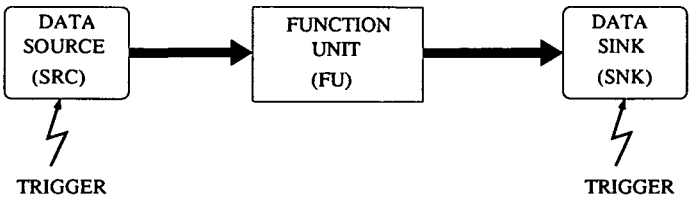


Figure 2.2. Circuit Model

this model. Thus a source trigger $TRG_{SRC,x}$ is employed to determine the instant $t_{issue,x}$ for a data word $DW_{SRC,x}$ to be issued by the data source SRC[1]. As soon as SRC is ready to accept a trigger, it will react by issuing the requested data word $DW_{SRC,x}$ which will – after some delay – become visible and consistent at the output of SRC at instant $t_{issue,x}$. The interval between trigger event ($t_{TRG,SRC,x}$) and actual visibility of the consistent data word $DW_{SRC,x}$ at the output ($t_{issue,x}$) is named the *issue delay* Δ_{issue}. Next $DW_{SRC,x}$ propagates to the function unit FU where it is processed. The corresponding result, $DW_{FU,x}$, propagates from the output of FU to the data sink SNK, passing SNK's input logic, until it is finally available as a consistent data word $DW_{SNK,x}$ within the sink and hence ready for consumption at instant $t_{SNKrdy,x}$. The interval between $t_{issue,x}$ and $t_{SNKrdy,x}$ is termed as *processing delay* $\Delta_{process}$. At some point in time $t_{SNKtrg,x} > t_{SNKrdy,x}$ the sink trigger $TRG_{SNK,x}$ is activated, which will – after some inherent delay – cause $DW_{SNK,x}$ to be actually consumed at instant $t_{consume,x}$. We call the interval between $t_{SNKrdy,x}$ and $t_{consume,x}$ the *consumption delay* $\Delta_{consume}$ and the interval between $t_{SNKtrg,x}$ and $t_{consume,x}$ the *sink trigger delay* $\Delta_{SNKtrig}$.

At instant $t_{issue,x+1} > t_{consume,x}$ it is safe to trigger the next data word $DW_{SRC,x+1}$ to be issued by the source. We call the delay until this actually occurs (i.e. the interval between $t_{consume,x}$ and $t_{issue,x+1}$) the *cycle delay* Δ_{cycle}. Notice that $DW_{SNK,x}$ does not necessarily become invalid immediately at $t_{issue,x+1}$ but only after $DW_{SRC,x+1}$ has propagated through FU to SNK. We describe this conservation of the previous data word by an *invalidity delay* $\Delta_{invalid}$. Consequently the system designers have the opportunity to choose a negative Δ_{cycle} thus increasing throughput by issuing the next data word $DW_{SRC,x+1}$ already before the current data word $DW_{SNK,x}$ has been consumed. Note that all delays may vary and hence some margins have to be considered in the timing.

2.2 The Fundamental Design Problem

Based on the aforementioned definitions the fundamental problem of digital logic design can be subsumed as follows: *Ensure a lossless information flow in the system.* Under this fundamental constraint systems are typically optimized for maximum information throughput. In order to achieve these aims the designer has to coordinate the

[1] As we will see later that on this trigger the essential means for controlling the data flow in the signal path

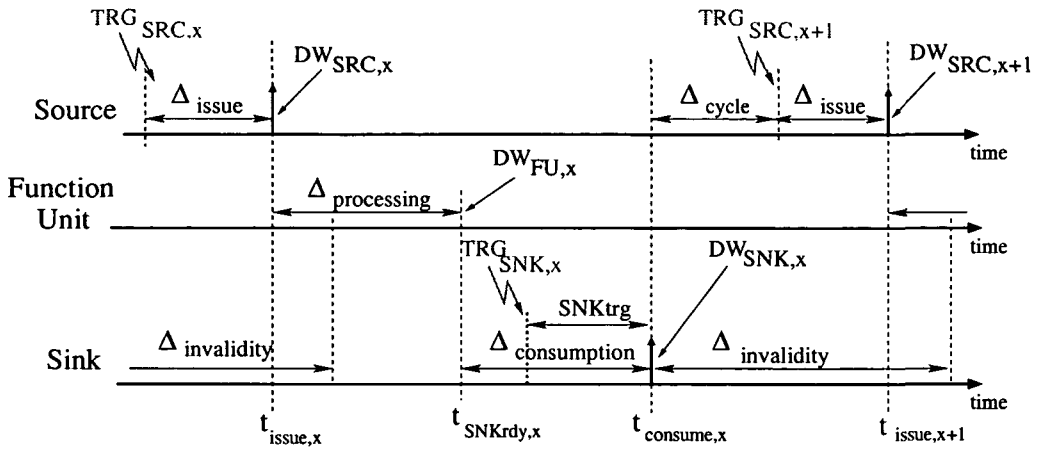


Figure 2.3. Timed Circuit Model

triggers of source and sink appropriately. In context with the timed data flow model presented above, this implies the following:

- The *trigger of the sink* $TRG_{SNK,x}$ must not be activated before $t_{SNKrdy,x}$ (ensure losslessness): $t_{SNK,trg,x} \geq t_{SNK,rdy,x}$. Less formally speaking this means that a new data word may only be captured by the sink after it has become consistent. To achieve maximum throughput capturing should, however, occur as soon after $t_{SNKrdy,x}$ as possible. As a consequence, every design method must allow a judgement of consistency of a data word in one way or the other (**fundamental requirement 1**). Considering that validity is a prerequisite for consistency it must be possible to judge on a signal's validity as well.

- The *trigger of the source* $TRG_{SRC,x}$ can safely be activated after $t_{consume,x}$ to guarantee losslessness, which means that the next data word may be issued only after the previous has been consumed: $t_{issue,x+1} \geq t_{consume,x}$. For maximum throughput it is desirable to place the trigger right after $t_{consume,x}$ or even prior to this instant (negative cycle delay). With respect to the design method this requires the existence of some kind of information feedback from the sink to the source (**fundamental requirement 2**).

Figure 2.4 illustrates these requirements. In practice requirement 2 has turned out to be relatively easily fulfilled by an appropriate circuit structure (micropipeline, e.g.), while the assessment of validity and consistency (fundamental requirement 1) is a notorious problem that we will analyze more closely in the following sections.

2.2.1 Formal Incompleteness of Boolean Logic

Boolean logic defines functions on a high abstraction level. In essence a Boolean function is a time-free mapping (truth table, e.g.) from the signals that form the input vector to an output signal. The output is reacting continuously to any change

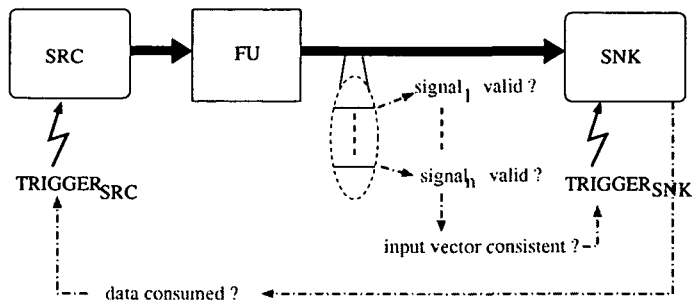


Figure 2.4. Fundamental Design Problem

of the input word – there is no such thing as a trigger. This further implies that only consistent data words are applied to the logic function. In other words Boolean logic does not provide any means for expressing or considering validity or consistency. Due to this fact Boolean logic is called "formally incomplete"in [31]. In fact there is no way of even expressing temporal relationships within the framework of Boolean logic – it is postulated that the input vectors are always consistent and the generated output is free of glitches. Unfortunately, due to signal delay and signal skew, none of these assumptions is fulfilled in a physical circuit implementation.

In conclusion, Boolean logic does not solve any of the fundamental requirements and so it does not contribute to solving the fundamental design problem in the first place. Still, Boolean logic is the established way of describing logic operations. All design methods have to compensate for this shortcoming in one way or another. In Section 2.3 we will analyze how different design styles solve this problem. However, before this action is performed we will analyze the roots of the problem in greater detail.

2.2.2 Signal Delay

Two constituents of signal delay are commonly distinguished, namely gate delay and interconnect delay. While gate delay is mainly determined by technology and fan-out, interconnect delay depends on many parameters that are specific to a given signal path: drive strength of the sender's output, capacitance and resistance of potential switch elements or vias along the wire, length and physical arrangement of the particular wire, and capacitance of the connected inputs,for instance. In addition, overall signal delay is a function of the operating conditions (supply voltage, temperature). As a consequence the time it takes an output to become valid is non-zero, which is contradictory to the assumptions made by the Boolean logic.

2.2.3 Signal Skew

Due to the uncertainties with respect to signal delays no pair of signals will exhibit exactly the same delay. The difference of delays within signals of the same input vector is called *skew*. Notice that by definition skew distorts the temporal relations between signal transitions. As a result, the assumption that the transition from one data word

to the next one will occur at once (as implied by the continuous, untriggered definition of a Boolean logic function) is unrealistic. The edges on the individual rails will rather arrive sequentially, causing inconsistent intermediate signals and input vectors that (temporarily) result in invalid outputs. In this sense the skew disproves the validity and consistency assumption made by the Boolean logic. Figure 2.5 illustrates this effect.

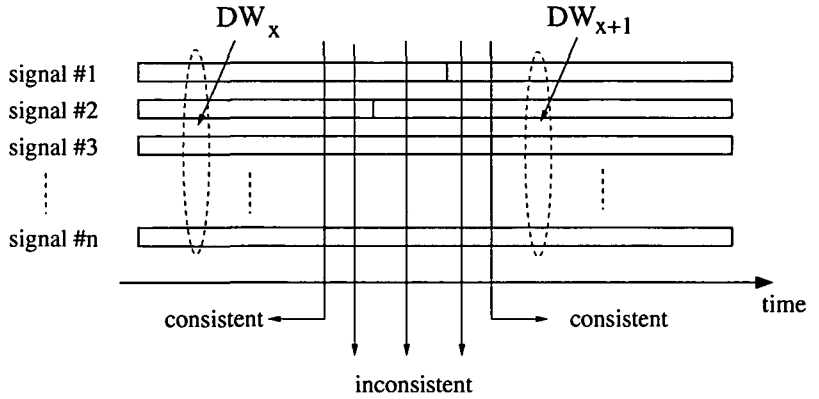


Figure 2.5. Transition between Consistent Data Words

As mentioned in Section 2.1 a signal may be represented by more than on rail. It is clear that in this case skew between the rails additionally compromises signal validity.

2.3 Strategic Options

In Section 2.2 we pointed out that it is an essential task of every digital design method to ensure that only consistent and valid data is consumed by the data sink and that the source is synchronized to the sink in such a way that no data get lost. In this section we will identify two basic domains where this can be performed. Remember, that it is not required to solve all aspects of the fundamental design problem in one domain – mixed solutions are also possible.

2.3.1 Time Domain

Having figured out *timing* issues – namely delay and skew – as one root of the fundamental design problem, one consequent solution is to compensate for their undesired effects directly in the time domain.

Concerning the validity and consistency requirement we can simply determine all relevant delays between source trigger $T_{SRC,x}$ at instant $t_{issue,x}$ and $t_{SNKrdy,x}$, the point in time when the data word is known to be ready for being captured at the sink. The sum of these delays constitutes the minimum time we have to wait after the source trigger until we can safely apply the sink trigger:

$$t_{consume,x} \geq t_{issue,x} + \Delta_{issue} + \Delta_{process} \tag{2.1}$$

The determination of Δ_{issue} and $\Delta_{process}$ involves a careful analysis of the (implementation-dependent) delays. In the same way we can relate the source trigger to the sink trigger:

$$t_{issue,x+1} \geq t_{consume,x} + \Delta_{SNKtrig} \tag{2.2}$$

Like above $\Delta_{SNKtrig}$ must be determined by means of a delay analysis of a given implementation. Remember, however, that delays vary, and therefore we cannot determine exact values, but we have to make conservative estimates to be on the safe side.

Based on this strategy we can use two different approaches to implement the control of the triggers:

1. The use of coupled *timers* that – started with one trigger event (source or sink) – generate the other respective trigger event (sink or source) after the appropriate amount of time (T_{SNK} or T_{SRC}).

2. The use of a global time reference for source and sink from which periodic triggers for source and sink are derived with an appropriate phase difference, T_{Phase}.

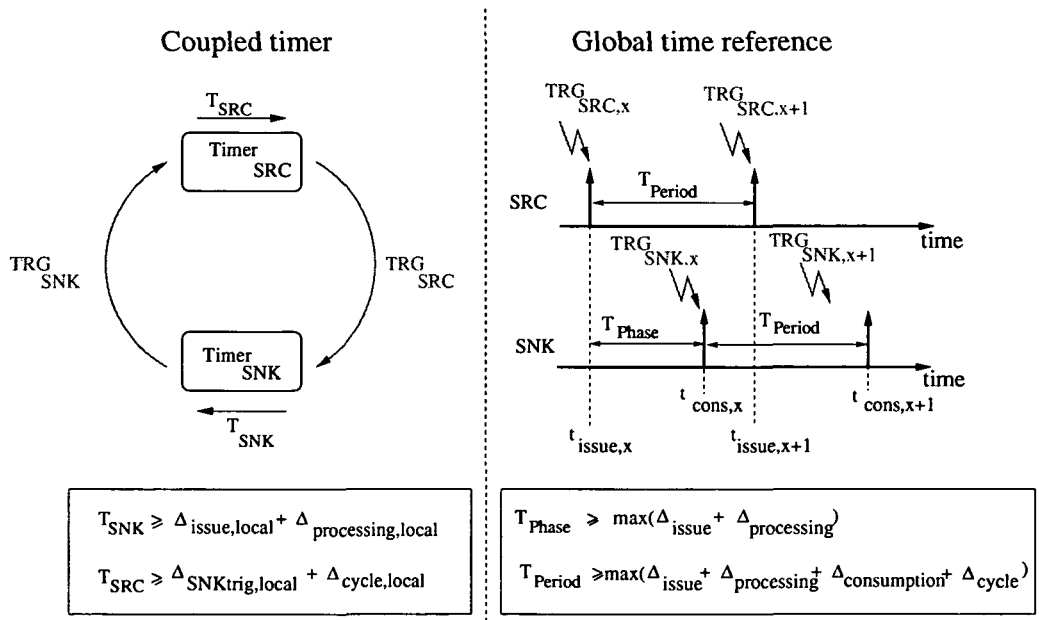


Figure 2.6. Fundamental Solutions in the Time Domain

The main difference between those methods is that the coupled timer approach only needs local delays, which are delays between the actual source-sink pair. In contrast the global timer has to use the worst case delays of the overall circuit. Another difference is that the latter uses $\Delta_{consumption}$ while the coupled timer has to consider only $\Delta_{SNK,trig}$: Due to the fact that a timer "starts" a new trigger event, only after an incoming trigger event had occurred, the difference between the point in time where data is ready to

be consumed and the point in time where the trigger is recognized at the destination side does not matter. These strategies are capable of solving the fundamental design problem on all levels, since all delays have passed and the circuit is stable at the trigger instants. In some sense we have thus overcome the formal incompleteness of Boolean logic by condensing the missing information on validity and consistency into the timer settings and using dedicated control signals to convey this information between source and sink. Notice, however, that we have just *postulate* that the input vector will be consistent and valid after $\Delta_{issue} + \Delta_{process}$, in fact we have no means to *directly* assess consistency and validity. As a result the determination of delays becomes a crucial issue. Two essentially contradicting arguments guide the choice of the timing settings:

1. *Restrictive assumptions:* It is not possible to determine any finite value for the delay without making assumptions on the implementation. Thus, the higher the delays the fewer assumptions must be made and the fewer restrictions apply to the implementation and the safer we can assume our losslessness property.

2. *Performance:* Obviously an overly conservative delay estimation has a negative impact on the throughput in terms of data words per second. In order to keep the resulting performance degradation minimal, a minimal overestimation of delays should be striven for.

So ultimately the choice of timing settings turns out to imply a tradeoff between performance and assumptions that have to be made on (and finally be met by) the implementation. Many models and techniques exist that allow to determine delays for a given circuit topology and technology. However, since delay and skew depend on many parameters, an "aggressive"choice of timing settings towards maximum performance compromises the robustness of the circuit.

2.3.2 Information Domain

Alternatively we can tackle the other root of the problem, namely the formal incompleteness of Boolean logic. Different methods are available to enforce the different fundamental requirements:

Validity: Recall from Section 2.1 that a signal is termed valid if it is the stable result of a Boolean function performed on a consistent data word. There are several possibilities to judge on the validity of a signal:

- **Ensuring continuous validity:** If we can manage to build the logic function unit in such a way that it produces only valid outputs, judgement of the output signal's validity becomes trivial. A function unit of this type must change its output only in response to a consistent input word[2]. To this end it must (a) be able to judge on the consistency of the input word and (b) hold the last

[2]Notice that ensuring continuous validity does not enforce continuous consistency, since the combination of valid signals pertaining to a different context does not yield a consistent data word

valid output signal during transient phases of inconsistent inputs. This obviously requires some kind of storage element for each logic function unit.

Even with an input perfectly changing from one consistent state to the other, skew within the function unit may cause invalid transient spikes at the output signal. Therefore special care must be taken for the design of the function unit. This causes a trade-off with respect to the partitioning of a circuit into function blocks: A coarse-grained partitioning into few function units saves storage elements, while a fine-grained partitioning facilitates better control of skew effects.

If the signal is composed of more than one rail, continuous consistency in the rail domain is a necessary condition for continuous validity in the signal domain. This can be ensured by the employment of a grey-code on the rail level, e.g. [118]

- **Extending the signal code:** Another approach to make validity visible is to establish a more comprehensive alphabet in comparison to the binary Boolean logic (by using more than one rail per signal, e.g.) and to define a subset of all expressible codewords, which are considered as "valid". In contrast to the previous approach, direct transition from one valid codeword on the rail-level to the next is no more mandatory, (invalid) intermediate states are allowed, since they can be identified. In other words, if a valid codeword has been reached after a number of single transitions on k of n rails of a signal, there must be no other valid codeword that can be reached by transitions on the remaining $n - k$ rails. This allows us to unambiguously identify when a codeword is complete, irrespective of the order in which the transitions occur. The transition to the next codeword must include another transition on at least one of the k rails. The same condition – though in a different formulation – has been presented in [114].

- **Current sensing:** This method exploits the fact that transient effects in a circuit are associated with current flow. Unfortunately, however, the reverse is not necessarily true: The lack of dynamic current flow is indeed a reasonable indication that the inputs are stable (and hence consistent?) and the output is stable and hence valid. Without any restrictions on the delays, it may well occur that one slow rail transition arrives after the circuit has been considered stabilized. Another problem with this method is the lack of an event separating two successive identical data words, which substantially complicates consistency judgement. Finally the inclusion of analog circuitry for the current sensors causes additional technological efforts [45].

Consistency: Imagine the situation depicted in Figure 2.7: SNK has an input vector composed by two signals, each of which are valid. This does not necessarily imply that the input vector is consistent, because the bits on the signal could well belong to different contexts. Notice, that validity does not imply consistency, but consistency requires validity.

To judge consistency, a circuit must be able to differentiate between two consecutive bits carried on a signal line, even if they hold the same information. This

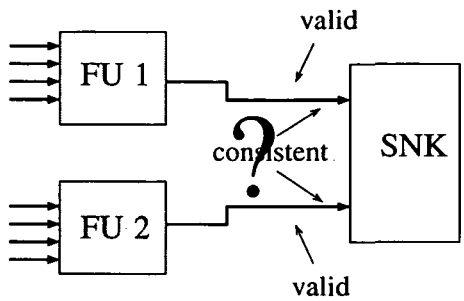


Figure 2.7. Validity vs. Consistency

means that we have to choose a signal level code which relates information to context. So in order to be applicable for our purpose, a coding scheme must meet two conditions:

Consistency Condition 1: *Existence of transitions*
There must be at least one signal transition between any two successive code words. While this naturally happens in transition based coding schemes, it requires special efforts to ensure a transition between two successive identical data words in state based coding schemes. A usual solution is to introduce a "neutral" code word (like all zero, e.g.) between any two data words in a "return to zero"manner.
Consistency Condition 2: *Membership to contexts*
As can be viewed in Figure 2.3, two data waves (belonging to a different context) will transiently coexist between SRC and the associated SNK: There is a finite interval when the new data wave has already been issued and propagates through the FU, but the previous one is still valid at the SNK's input. This procedure is properly synchronized by the trigger control. If more data waves were admitted between SRC and SNK we would loose control of them and in particular not be able to prevent one data wave from catching up with its predecessor (unless this is ensured by timing assumptions). As a consequence, if our basic requirement is to be able to distinguish data waves with different contexts, we normally come along with two disjoint code sets on the rail level, which allows us to unambiguously assign every bit to one of the two data waves. which includes when

Losslessness: As already outlined, the losslessness property requires us to provide the data source, SRC, with information when new data can be issued and the data sink, SNK, with information on when data can be consumed. The latter can be achieved by checking consistency and validity of the SNK's input vector without relying on the time domain.

The source trigger can only be derived from information explicitly provided by the data sink such as a control signal acknowledging the consumption of the previous data word. Since there is only one single bit of information required on this backward path, there is no potential for skew effects. Nevertheless, the consumption of a data word can usually not be directly measured, which gives rise to conceptually weak

compromises in this respect.

From a higher level of abstraction we can consider the function unit as part of the data source/sink and map the lossless requirement of a communication process problem (see Figure 2.8).

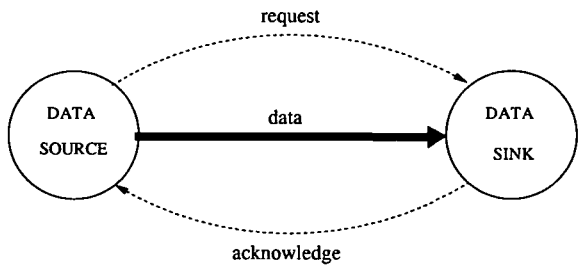


Figure 2.8. Communication Process

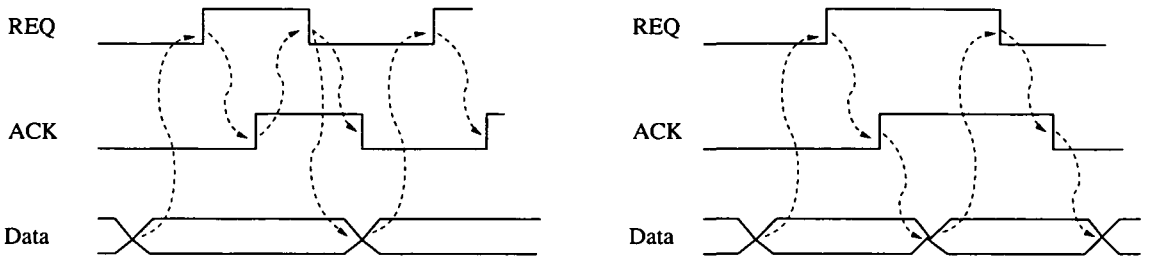
In fact there is a strong relation between communication channels and delay insensitive circuits [77]. However it is essential to realize that communication channels solve only a part of the fundamental design problem, namely losslessness. Consistency and validity cannot be answered by a communication channel alone, other mechanisms for this purpose are still required. Due to the fact that a lot of literature concerning communication channels in context with asynchronous logic [115] [123] [106] exists, we will give only a brief overview it in this section. A data source and a data sink are connected over a *communication channel*. The point were a channel is connected is called a *port*. We distinguish between *unidirectional* and *bidirectional* channels. For the following we will consider only unidirectional channels which reflect the natural of communication in digital circuit. A port can be *active*, this means that such a port initializes a communication process, or *passive*, where the port reacts on incoming events.

Obviously there must be an agreement between source and sink, in which way data is transmitted over the communication channel – a so-called *communication protocol*. Basically we can distinguish between a *2-phase protocol* and a *4-phase protocol*. In contrast to the 2-phase protocol, the 4-phase protocol returns back to its "neutral state", after each communication cycle. (see Figure 2.9)

Furthermore we have to distinguish between *push channel*, where the data source is the active party, and *pull channel*, where the data source reacts on requests of the data sink. A detailed description of communication mechanism with respect to asynchronous circuits can be found in [77].

2.3.3 Hybrid Solutions

It is not necessary to solve the fundamental design problem in one domain only. Quite on the contrary, many design approaches are based on a hybrid solution. Huffman codes [47] or micropipelines [107], e.g., solve only a part of the fundamental design problem and only their combination with other methods yields the desired result.



a) 4–phase protocol b) 2–phase protocol

Figure 2.9. Communication Protocols

In most cases library cells, such as AND, OR, latches, .., are implemented by making local timing assumptions e.g. isochronic fork [62] or fast local feedbacks [31][30], since it is quite easy to consider timing assumptions within such atomic cells and yield more efficient implementations in terms of speed and silicon area.

This leads to a further classification of circuits with respect to the assumptions made about timing [104]. Figure 2.10 shows a circuit fragment comprising three gates, where the output signal of gate A is connected to the inputs of gate B and C. The delays inside the gates Δ_A, Δ_B and Δ_C, represent the processing delays, while Δ_1, Δ_2 and Δ_3, form the propagation delays of each wire segment.

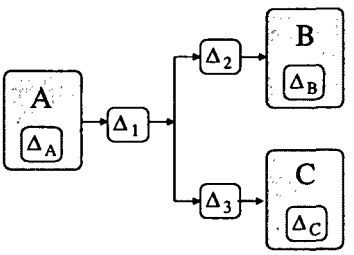


Figure 2.10. Circuit Fragment with Gates and Delays

Depending on the assumption made with respect to delays, circuits can be classified as follows [104]:

Delay-insensitive circuits (DI): We consider a circuit delay-insensitive if its correct operation depends neither on gate delays nor on wire delays. [62] shows that only circuits composed by Muller-C-gates and inverters can be delay insensitive using single output gates. This is a strong restriction, which limits the practical applicability of such type of circuit. However, this is the only class of circuits, which solves all aspects of the fundamental design problem exclusively in the information domain.

Quasi-delay-insensitive circuits (QDI): These circuits are delay-sensitive with the exception of some carefully identified wire forks. Related to Figure 2.10 this would

require that $\Delta_2 = \Delta_3$. In other words, the QDI approach hypothesizes that all transitions at the end point of (carefully selected) wired forks occur at the same time. Such forks are called *isochronic forks.*

Speed-independent circuits (SI): These circuits operate correctly, assuming that gate delays are bounded but unknown and that the wires are ideally zero delayed. Hence a SI implementation of the circuit depicted in Figure 2.10 would require that $\Delta_1 = \Delta_2 = \Delta_3 = 0$.

Self-timed circuits (ST): Forcing always (Q)DI or SI could result in an overkill – sometimes a tradeoff between implementation complexity and delay assumptions is reasonable. In this sense circuits whose correct operation relies on more elaborate and/or engineering assumptions are called self-timed circuits.

Timed circuits (TI): In this class of circuits all delays, gates and wire delays, have to be taken in to account in order to ensure a correct behavior of the circuit. In other words, such types of circuits solve the fundamental design problem entirely in the time domain.

Furthermore, we different abstraction levels of a circuit implementation have to be considered. Until now we have dealt with abstract logic function blocks only, disregarding whether we are considering a simple inverter built from two transistors or a complex ALU. The distinction between abstraction levels is vital because several design approaches use speed-independent or quasi-delay-insensitive library cell implementations (on transistor level) and combine them yielding to a delay-insensitive circuit on gate level. In this way the timing analysis of arbitrary circuits is restricted to a small number of (little) library elements and hence has to be performed only one time during library compilation. This allows us to build circuits, such as an ALU, for which the fundamental design problem is entirely solved in the information domain (on this higher level of abstraction).

2.4 Design Techniques

This section is intended to give an overview about current design techniques with the aim to illustrate how they solve the fundamental design problem. Obviously not all design methods developed in the last half century can be covered. Instead, characteristic representatives of each design approach will be dealt with.

2.4.1 Synchronous Approach

Basic principle: The synchronous approach answers all subproblems concerning the fundamental design problem in the time domain using a common time reference (see Figure 2.6). It employs a unique control rail, the clock signal, to indicate validity, consistency and losslessness at the same time. At every active edge of the clock all

signals have to be consistent and valid by definition and therefore ready to be consumed. Due to the fact that data sources get the same clock signal as data sinks, the active clock edge signalizes also the point in time where new data can be issued. In this way the regulation of the data flow is also strictly based on time and occurs without feedback (see Figure 2.11). By assuming that all data sources and sinks get a common global time reference from the clock signal, it is implied that all these components actually get the active edges at the same point in time. However, since skew and delay effect also affect the clock signal, this claim is not justified for deep sub-micron technologies. Quite on the contrary, [67] predicts that in the near future only a small percentage of the die will be **reachable** during a single clock cycle. Furthermore, the clock signal has no immediate relation to consistency/validity of signals or rails and the clock signal – it is just a strictly periodic and time driven control signal.

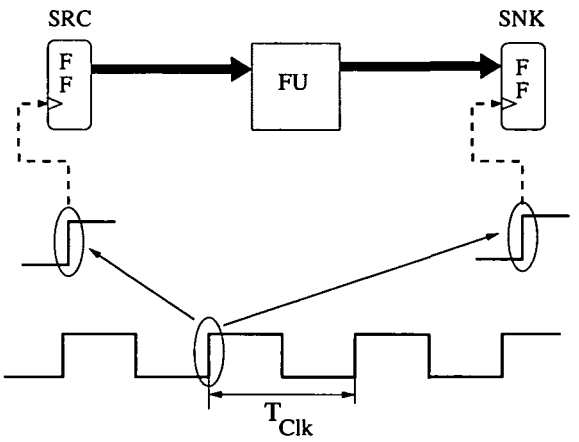


Figure 2.11. Synchronous Design Approach

The minimum distance between active clock edges T_{Clk} is derived from $\Delta_{process}$ and $\Delta_{consumption}$. Its calculation is based on worst case assumptions concerning physical properties, performable operations, applicable data and operation conditions [40]. Δ_{Cycle} and Δ_{Issue} are reflected in hold and setup time of registers. Note, that in the synchronous approach data is consumed and issued in exactly the same point in time. Further it is assumed that both data sources and sinks are always ready to perform their operations on each active clock edge – flipflops have no means to signalize that they are busy at the moment.

Efficiency: The synchronous approach is extremely hardware-efficient, since it uses one single global control signal, which is easy to generate by means of a crystal oscillator. The highly efficient single-rail encoding can be used to represent all signals. If the logic state of a signal changes from one data word to the next, a signal transition is performed; if the state remains the same, no transition is required. Assuming a random distribution of state patterns on a signal, this yields to an average of 0.5 signal (=rail) edges per bit, which means that the energy consumption caused by data transitions is extremely low. Assuming a properly chosen clock frequency, no consideration of

transient effects and consistency issues is required during functional design.
Through the insertion of so-called pipeline registers the signal path is often structured into smaller sub-paths. The timing of these smaller sub-paths can be more easily analyzed, and in addition pipelining yields some performance gains [44].

Problems: So apparently all problems of logic design are solved by the synchronous approach, and indeed millions of synchronous designs have been working properly and reliably over the past decades. Still, however, substantial problems have remained unsolved on the conceptual level, and the current technology trends make these problems more and more evident:

- The indirect conclusion from time to consistency and validity of signals is the main conceptual deficiency: Time is easy to measure but not by itself an indication for consistency and validity. In fact, an artificial correlation between time and consistency and validity is extremely hard to establish and can not never be guaranteed.
- The assumption of stable states during functional design does not eliminate the need for consideration of transient effects. In fact it only postpones the problem to an explicit timing analysis that is required later on. This timing analysis is often much more complicated than the functional design. With the increasing clock rates and the proceeding miniaturization this problem becomes more and more stringent.
- With its wide extension and the strong drivers required to keep delay and skew low, the clock network dissipates a significant share of the power of a chip. In order to be able to keep the clock skew within 300 picoseconds, the designers of the DEC Alpha CPU [102] developed a clock driver circuit, which dissipates over 40% of the power of the entire chip ([20]). Unfortunately this outweighs the advantage of low power consumption in the data path. In addition, substantial heat problems are caused by the fact that switching activities are periodic rather than demand driven.
- A solution of the delay and skew problems in the timing analysis phase is possible only if restrictions on the timing behavior are made. This, however, has severe consequences:
  - Considering that interconnect delays already dominate gate delays [98] realistic timing estimations can only be constructed after the place & route, i.e. at a very late point in the design process. In practice, however, timing problems often necessitate changes in the functional design. In this way the separation of functional design and timing analysis causes unnecessarily long iteration cycles.
  - Any change in the circuit or technology requires a complete revision of the timing analysis.

- As already mentioned the actual delays of a given implementation still depend on the operating conditions and are affected by type variations. Hence the delay assumptions made during the timing analysis *must* be arbitrary to some extent. While assuming the worst case scenario within the specified range of operating conditions clearly leads to performance loss in the average case, there is still a residual risk of exceeding the assumed limits: *"...In order to achieve a reasonable shield against these variables, the clock period is extended by a certain margin. In current practice these margins are often 100% or more in high speed systems."* [20]. Some innovative design methods [110][84] soften this rationale by adopting the clock rate to the actual condition. However this requires an additional effort in terms of silicon die and control mechanisms.

- As a matter of fact no restrictions can be made for asynchronous inputs at synchronous/real-world borderlines and interfaces to other clock domains. Consequently these signals cannot properly be considered in the timing analysis, and so metastability problems arise [33]. By use of additional synchronizer circuits metastability can be made sufficiently improbable, but no conceptual remedy to completely eliminate it has been found so far.

- Synchronous designs have a very problematic behavior with respect to EMC, since most of the energy is concentrated in one single spectral line.

2.4.2 Bundled-Data Approach

Basic principle: The basic concept of **bundled-data** [104] is to arrange several (data-) signals in a group and to use a common control signal, which serves as a trigger to signalize validity and consistency of these (data-) signals. The control signal is generated at the same time as the related data signals by the source node and hence to operate correctly, the data path must be at least as fast as the control path. To ensure this procedure it may be necessary to insert additional delays, so-called *matching delays* in the control signal path. In this sense bundled-data solves consistency and validity in the time domain. The control signal can be used as a trigger for data sinks only and therefore the bundled data approach does not provide any means for data flow control. This requirement has to be fulfilled by other methods or on a higher system level.

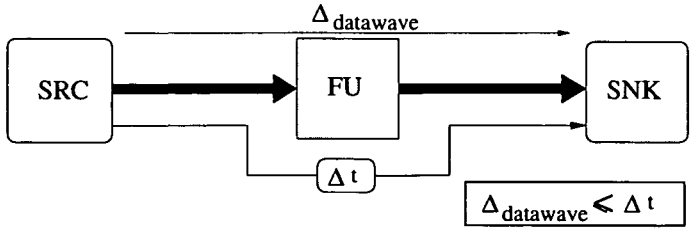


Figure 2.12. Bundled-Data Design Approach

As illustrated in Figure 2.12 consistency and validity are ensured in a similar manner as in the synchronous approach. This allows asynchronous designers to use standard (i.e. non hazard-free) implementation of logic function units [52]. The main difference between the synchronous and the bundled data approach is that the latter requires only local timing information (see coupled timer in Section 2.3.1) instead of taking into account the whole circuit to determine the temporal sequence of trigger events.

Efficiency: The most efficient representation of data is to use one single wire per bit – the higher the number of data bits, which are bundled, the closer the bundled-data approach moves to this maximal efficiency rate. Apart from the matching delays, which can be implemented using inverter chains or by duplicating the critical path of the stage between source/sink, no extra completion detection circuits are required. Assuming a random distribution of state patterns on a signal and a reasonable number of bundled signals, the bundled data leads similar to the synchronous approach to an average of 0.5 signal (=rail) edges per bit. Thus, the bundled data approach is highly efficient not only in terms of silicon area, but also in terms of energy efficiency. Due to this fact bundled data was used in several asynchronous designs implementations [38][54][82][108].

Problems: Although the major difficulty of the synchronous design style, namely provide a global time reference anywhere in the circuit, is defused by requiring only local timing information the bundled data still faces some problems:

- Time is still used to determine consistency and validity of signals. The basic problem with this indirect conclusion is similar to those in synchronous systems, even if the locality makes it more manageable.
- The matched delays have to be calculated considering worst case scenarios. These yield to waste of performance.
- Due to the increasing dominance of wire delay over gate delay [125], matching delay can be determined reliably only after place&route. Furthermore a validation of the final circuit is required, due to the fact that some variations during the fabrication may affect the (data-) signal path but not its related matching delay for example.
- Moving to a new technology all delay elements have to be re-calibrated.
- Bundled-data is usually used to model data busses. However means to control the data flow are not provided.

2.4.3 Huffman Approach

Basic principle: D.A. Huffman [47] can be considered as one of the spiritual parents of the asynchronous logic design. Huffman developed the so-called fundamental mode circuits. These circuits [78] are intended to be used for asynchronous state machines. As depicted in Figure 2.13 Huffman circuits have primary inputs, primary outputs,

and a the feedback loop. The fundamental mode requires that only one input signal changes at a time. The current state is "stored" in the feedback path and thus may need delay elements to prevent state changes from occurring too rapidly. However the feedback signals are inputs of the combinational logic as well – hence it is even required that by passing from one to the next state, only one bit changes. Therefore the state encoding scheme has to be carefully chosen [104]. A further requirement of Huffman circuits is, that the combinational logic is glitch-free, which can be achieved through redundant terms in the KV-map [78].
While validity is answered in the information domain (glitch-free functions) and by the environment (only one bit changes at the input side), consistency is solved by the delay element in the feedback path. The lossless property has to be guaranteed by the environment: It is assumed that a new input vector is issued only when the circuit has reacted a stable state.

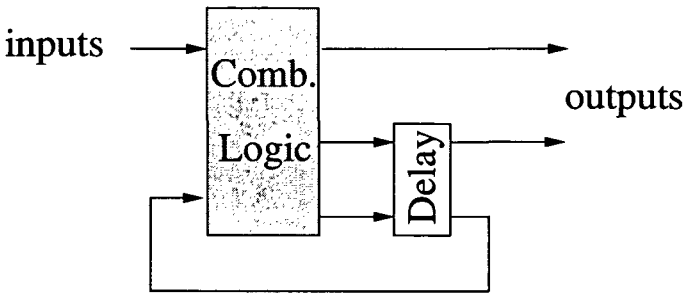


Figure 2.13. Huffman Circuit [77]

There are some enhancements of the Huffman circuit which soften the fundamental mode requirement. The *multiple input changes*(MIC) [34][58] extension is based on the assumption that the input changes happen within some tightly bounded interval of time, and hence they can be considered to have occurred simultaneously. Stevens [105] allows input changes at any time as long as they are grouped together in bursts. This yields to the so called *burst mode* circuits. The most general mode of operation is the *unrestricted input change* mode (UIC) [111]. The UIC design method demands that an input does not change twice in a given time period.

Efficiency: Just like the approaches presented previously Huffman circuits use a single rail encoding. However the Huffman approach does not allow glitches, albeit delay elements are used. The reason is that the delay element is not used to primarily signalize consistency, but prevents the circuit to become unstable, due to the feedback signal. The demand of being glitch free limits potential optimizations during synthesis and leads to larger circuits. However, a lot of work has be done in this field, the interested reader can find further information about Huffman circuit synthesis approaches in [17][18] [90][116][122]. The restriction, that a new input can occur only when the system has settled in a stable state, limits the throughput: A new input must be delayed at least two times the delay of the combinational logic (in the first step the next state is calculated, in the second step the output is settled according to the input and the

new state information) and one time the delay of the delay element. Using a one-hot state encoding simplifies the associated logic but worsens the throughput further: *"... For a one-hot encoding, this means that a new input must be delayed long enough for three trips through the combinational logic and two trips through the delay element."* ([42] p.71)

Problems: The requirement posed by the fundamental mode but also by its extensions, MIC and UIC, lead to several limitations of the circuit design:

- One big handicap of Huffman circuits is that data paths cannot be implemented – a data bus carries information, which is arbitrary per definition and hence restrictions can not be applied. This limits the practical applicability of the Huffman approach to control circuits only.
- The implementation of hazard free circuits, requires an additional effort during system design. An in-deep discussion about *Hazard-free two-level logic synthesis* can be found in [77] on page 165 ff.
- Some boolean functions may not change monotonically during a multiple input change. Such functions are considered to have a *functional hazard.* Eichelberg [29] shows that it is impossible to build a hazard-free gate level implementation of a function, which has function hazards.
- Although of glitch-free function units have to be used, delay elements are still required. The same drawbacks concerning delay elements, as mentioned in the previous sections, are true for the Huffman circuits.
- No means for data flow control are included – it is postulated that data is issued in a correct manner by the environment. The fact that points in time where data can be issued depend not only on a straight forward delay calculation of function units, but to the delay calculation of circuits containing loops, aggravates this weak point.

2.4.4 Design Techniques Using Signal Coding – The NCL Example

Basic principle: Many approaches exist, which use signal encoding to ensure validity of signals and make consistency of signal vectors directly visible [57]. NCL (Null Convention Logic) which was developed by *Theseus Logic* was chosen as the representant of this class of implementation approaches due to the fact that it is the most mature one and because some industrial experiences have been already made [70]. This design approach extends the Boolean logic by a so-called NULL state [31]. In particular an NCL signal can assume a DATA state – which is either a valid HI or a valid LO, in NCL called "TRUE"or "FALSE", respectively – or a NULL state. For encoding these three states the single-rail approach is obviously not sufficient, and a two-rail signal representation is used instead, with NULL being represented as (0,0), TRUE as (1,0) and FALSE as (0,1). The NULL state does not convey any information, it serves only

as a neutral state separating two consecutive codewords. Figure 2.14 illustrates this behavior.

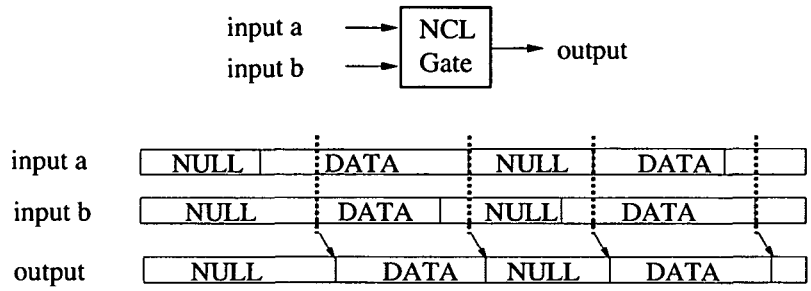


Figure 2.14. Sequence of DATA and NULL Waves

Feedback ensures that a new data (DATA or NULL) can be processed only when the input vector is consistent. To realize this behavior so-called threshold gates are used [56]. These gates change their output only when the complete input vector is either DATA or NULL. This hysteresis provides a synchronization of the wavefronts on the gate level. In other words consistency and validity check on signals are implemented at gate level. With the proposed encoding on signal level exactly one rail changes its logic level upon the transition from NULL to DATA and vice versa, regardless of whether DATA is TRUE or FALSE. Due to the mandatory introduction of the NULL waves a neutral state (NULL) is assumed on every signal after every single data word, which enforces the edge required to meet the consistency condition 1 (see Section 2.3.2). From this neutral state an edge on any one of the two rails leads to the TRUE or FALSE state, which guarantees that the codeword itself is always valid. The NCL approach does not provide any mechanism to ensure losslessness.

Efficiency: A NULL state between each pair of DATA states regulates the data flow in onward direction and ensures consistency. From a performance point of view this convention is very expensive – in fact the maximal achievable throughput is halved by the NULL wave. However, due to the fact that this approach does not require any delay elements, the resulting circuit operates as fast as it can, which partially compensates the drawback of the NULL wave.
In contrast to single-rail encoding styles where the average of 0.5 signal (=rail) edges per bit can be assumed, NCL requires in any case 2 edge per bit on the rails.
The usage of two rails per bits yield by its nature to larger circuit compared to single rail implementations. Furthermore each NCL primitive requires some kind of storage element which increases again the price in terms of silicon area. However, Theusus Logic proposes some tricky hardware solutions which keep this overhead within reasonable limits [30].

Problems: The NCL approach integrates data and control information in a single expression. This merger combined with the alternation of DATA and NULL waves makes

validity and consistency directly visible, without making (apparently) any assumption about timing – this feature has its price:

- Higher effort in terms of gates and interconnect: the dual rail encoding doubles the number of wires and multiplies size of logical gates: A gate with two single-rail inputs has four possible input combinations to take in account, while a two dual-rail input gate has sixteen possible input combinations.
- The convention that NCL gates start to produce a new output value only when all inputs are in the NULL/DATA state, requires that the gate holds its output value in between. As consequence a NCL gate must contain some kind of memory element inside. Theseus Logic proposes *threshold gates* for this purpose. The functionality of these gates is basically implemented using feedback signals inside the gate. Although NCL does not require timing assumption on gate level, to operate properly the feedback signals inside the gates have to be fast enough to settle the gate before the next input vector change occurs. This is a sustainable requirement, however due to the fact that a timing assumption has to be made, NCL circuits have to be classified as quasi-delay-insensitive circuits rather than delay-insensitive ones.
- The NULL waves reduce throughput on the one side and energy efficiency on the other side (see previous paragraph)
- NCL does not provide any means for data flow control. This means NCL has to be combined with other design techniques such as *Micropipelines* .e.g. For this purpose consistency of a signal vector has to be provided explicitly to the additional design method. This requires a further circuit, so-called *Completion Detection Circuit* (CMPD).

2.4.5 Transition Signalling Approach

Basic principle: In conventional coding techniques logic states of signals are mapped to voltage levels of physical rails. In contrast transition signaling [104] uses *edges* on rails to convey the information. Transition signaling also employs two-rail coding on the signal level. A transition on one rail indicates a HI, a transition on the other rail a LO. From a more abstract point of view transition signaling uses a one-hot encoding scheme for HI and LO and therefore fulfils the validity property on code level. The neutral state between consecutive codewords is defined by the absence of transitions on the rails. In contrast to NCL, where the neutral state must be explicitly generated, transition signaling provides this state automatically and hence a new codeword is recognized even if it carries the same information as the previous one. In this sense consistency is integrated directly in the coding style. In [63] it has been shown that the only single output gates that can be used in conjunction with transition signalling circuits are Muller-C-Gate and inverter. This limits the usability of this scheme for real circuits.

Efficiency: Transition signalling can be compared to a NRZ coding style. This obviously favors the achievable throughput and hence promises higher performance for circuits using this approach. Albeit transition signalling uses a dual rail encoding, only one single transition/edge per bit is required. Note that a transition occurs in any case, even if the same bit information is transmitted consecutively by the same signal. Thus data content itself does not influence the number of edges required to convey the information.
Compared to single rail encoding the dual-rail approach doubles the number of wires. However, the main weak point of transition signalling with respect to area efficiency is the complexity of gates, which are able to operate on signal transitions instead of signal levels.

Problems: Coupling information to events is an extremely elegant method to solve the fundamental design problem concerning validity and consistency. Nevertheless there are some (practical) problems which inhibit the breakthrough of this design technique:

- Gates require a high implementation effort due to the fact that they operate on edges instead of signal levels. Furthermore the set of allowed gates is limited, this restricts the practical applicability of this design style.
- The basic principle of digital design is to distinguish between two discrete signal states/levels, namely LOW and HIGH or '0' and '1'. Transition signalling based on transitions of signals instead of levels of signals, means that transition signalling is event based instead of state based. Hence, this approach requires to completely change the well established and approved way of thinking concerning digital circuit design. This radical change demands not only new tools but also a complete re-education of engineers.
- Transition signalling circuits are susceptible to interferences. Each glitch even the smallest one produces two edges, which are interpreted per definition by a transition signalling circuit as two valid bits. Muller-C gate implementations as proposed in [107] moderate this problem, since they are more robust against glitches. In spite of the risk that a small impulse generated by an electrical interference e.g. causes a malfunction is much higher than in other design approaches.
- The fact that transition signalling is events based makes it extremely difficult to debug transition signalling circuits. Debug tools cannot directly derive the logical information carried by signal – instead the event sequence must be journalized to determine information, which is currently conveyed by the signal.

2.4.6 Handshake Protocols: The Micropipeline Approach

Basic principle: There are several choices of handshake protocols, which can be used to control the communication inside a circuit [77](see Section 2.3.2). The **micropipelines** introduced by Sutherland [107] in particular uses a 2-phase signalling

for the handshake protocol. Basically micropipelines are a means for structuring complex logic designs in general and data path designs in particular. In contrast to the synchronous pipelines they employ local handshake signals between any two pipeline stages to interlock the inter-operation between the individual stages so that the speed of data flow can be adapted to the local situation. They provide an elastic pipeline for the handshake signals that allows to buffer requests. In this way the micropipeline approach provides a straightforward solution for data flow control.

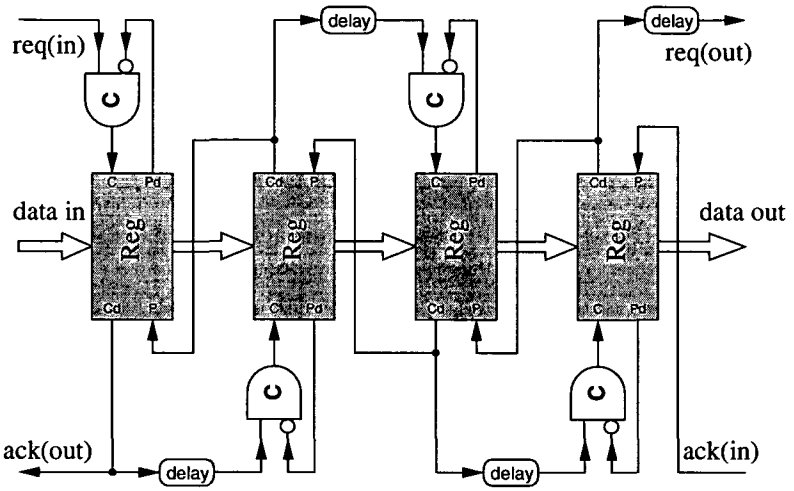


Figure 2.15. Micropipeline

The latches inside the micropipeline have two operation modes:

- Transparent: input data is passed directly to the output.
- Frozen : the latch maintains the value of the output independently of the input data.

As illustrated in Figure 2.15 the latches have four control signals, by means of which their behavior can be controlled: *Capture*(C), *Capture_done*(Cd) and *Pass*(P), *Pass_done*(Pd).
The *Pass* input sets the register in the transparent mode. After a certain delay the register achieves this state which is signalized by the *Pass_done* signal. Similarly the *Capture* and the *Capture_done* signals freeze the latch and signalize that the latch is effectively frozen. The *Muller-C gate* [107], which acts as AND concatenation of events, ensures that the latch freezes only when the new input data has been passed through the register. The original micropipeline approach employs *delay elements* to ensure consistency. Fundamentally this corresponds to the bundled data circuits between pipe-latches. However, it is possible to generate the completion signals by combining the micropipeline with other design approaches [117].

Efficiency: First of all the micropipeline approach provides a mechanism to control the dataflow. Like its synchronous counterpart, the micropipeline can be further used

to enhance throughput of circuits. Especially the micropipeline introduced by Sutherland seems to be particularly suitable for this purpose due to fact that it implements a 2-phase-handshake protocol. This means that no *Return-to-Zero* is required, which shortens the cycle time. However, practical experience shows something quite different: Based on this argument the first asynchronous ARM processor *Amulet1* [121] was developed using techniques based extensively on Sutherland's Micropipelines. For the second processor generation, Amulet2(e) [37], a 4-phase-handshake protocol rather than a 2-phase-handshake protocol was chosen because it was discovered to be simpler and more efficient.
Examination of area efficiency is not meaningful when we only are speak of about communication protocols. To implement the function unit between pipe stages micropipelines have to be combined with some other design style – bundled data was suggested by Sutherland e.g.. Therefore the area efficiency depends strongly on the chosen method to make consistency visible.

Problems: The Micropipeline approach is intended to solve only one part of the fundamental design problem, namely the data flow control. Weak points of this approach are:

- The original micropipeline introduced by Sutherland provides a bundled data approach to signalize consistency. This moves all problems mentioned in the section about bundled data to the micropipeline approach as well. Also the bundled data design style can be replaced by any other approach which solve the consistency problem.

- Apart from the delay element parallel to the data path, two additional delay elements are required inside the latches: a *capture_done delay* and a *pass_done delay*. This vast use of delay elements cuts down the potential advantage of asynchronous circuits.

- Although the 2-phase-signalling used for the handshake protocol promises higher performance compared to a RTZ protocol, the practice has shown the opposite: The second generation of the AMULET processors was based on a 4-phase-signalling handshake protocol due to the fact that the 2-phase-signalling in the first processor generation permitted only a slow and complicated implementation.

2.4.7 High Level Description Approaches

Basic principle: In contrast to all methods discussed so far, *High Level Description Approaches* do not explicitly consider the effective hardware implementation of the circuit, but outsource this aspect to an (automated) synthesis process instead. Hence the main task of these high level methods is to purvey a description, which fulfills specific constraints/requirements in order to enable the synthesis tool to build correct operating circuits. However the synthesis process on its part has to revert to one of the "low level" design approaches described previously. Therefore related to strategic

options high level description methods do not pose a new design technique, they provide a framework to develop circuits and to formally verify its behavior instead. High level descriptions fall roughly into 2 categories, namely *Graphical methods* and *Translation methods.*

Graphical methods: Due to the fact that Petri nets [99] are used to describe concurrent systems, almost all of the graph-based methods are based on this graph model or on a restricted form of it [53]. *Signal Transition Graphs* (STG) introduced by Chu [16] are such a restricted form of a Petri net, which allow only limited options to select alternative responses of the circuit. Other variants of Petri nets are *Interface nets* (I-nets) [72] *Machine nets,* (M-nets) [97] or Change Diagrams [113]. *Timed Event/Level structure* (TEL) is a graphical method which, allows specify timing information [94], in order to permit efficient circuit implementations.

Translation methods Almost all high-level description languages for asynchronous circuits are based on the use of a language that belongs to the *Communicating Sequential Processes* (CSP) [14] [13] family, rather than to classical hardware description languages such as VHDL [4] or Verilog [3]. The characteristics of CSP are described in [77] as follows:

- Concurrent processes
- Sequential and concurrent composition of statements within a process
- Synchronous message passing over point-to-point channels (supported by the primitives send, receive and – possible – probe)

OCCAM [109][83] *LOTOS* [124] and *CCS* [71] are programming languages which are able to describe parallel processes. *Tangram* [10], *CHP* [61] and *BALSA* [8] are languages which are specially designed to model (concurrent) asynchronous circuits.

Efficiency: In general high level descriptions permit shorter development cycles due to automated processes below the abstraction level of the description. Today global optimization techniques for asynchronous logic are difficult to utilize during the translation process and hence automated synthesis often produces inefficient results [53]. However it is a matter of time until asynchronous synthesis tools achieves the same quality as it synchronous counterparts.

Problems: There are mainly three problems which can be identified concerning high level descriptions:

- Although the asynchronous design style has a long history, interest arose only in the last decade and thus researchers and engineers started to investigate this discipline. It is clear that existing approaches and tools are not fully developed yet.

- Only circuits with limited complexity can be modelled. This is especially true for graphical based approaches due to their awkwardness in specifying input choices [53].
- The automated synthesis process hides the information about the implementation on gate level. Having a well approved and established tool chain this may be a desired property, but as the asynchronous design techniques are being still in the fledgling stage this circumstance limits the possibilities to investigate the implemented circuit and to find out possible improvements.

2.5 Comparison

Due to the fact that different design techniques are intended for different purposes – Huffman circuits for ASFMs, bundled data for data path modelling e.g. – and because each design style has a lot of extensions on its part, it is difficult to make a comparison. Thus we will confront the presented design techniques with respect to basic aspects and compare them only in a qualitative manner. This should still enable the reader to judge the presented design techniques and visualize their advantage and drawbacks.

Covered part of the fundamental design problem: The most characteristic features of a design technique are the aspects of the fundamental design problem it covers and the domain (time or information) in which the related problems are solved. Hence in Table 2.1 the presented methods are compared with respect to the domain, in which they solve consistency, validity and losslessness. The column *E* (Environment) is used to express that the design technique does not solve the corresponding subproblem, but moves the responsibility to the environment. Column *I* (Information) and *T* (Time) are used to express whether the problem is solved in the information or in the time domain.

	Validity			Consistency			Lossless		
	T	I	E	T	I	E	T	I	E
Sync	x	-	-	x	-	-	x	-	-
Bundled Data	x	-	-	x	-	-	-	-	x
Huffman	-	x	-	x	-	x	-	-	x
NCL	-	x	-	-	x	-	-	-	x
Trans. Sig.	-	x	-	-	x	-	-	-	x
Microp.	(x)	-	-	(x)	-	-	-	x	-
High Level Desc.	-	x	-	-	x	-	-	-	x

Table 2.1. Comparison wrt. the fundamental design problem

In contrast to all other methods the synchronous approach provides a complete solution of all subproblems of the fundamental design problem in the time domain. On the one hand the clock signal guarantees consistency and validity at the instant when

data is taken over and on the other hand it regulates the data flow. The bundled data approach is intended to soften the problems concerning distribution of a global time reference by using local timing assumptions only. It makes consistency and validity "visible", but leaves data flow control issues unconsidered. Similarly the Huffman circuits move the responsibility to provide only "allowed" inputs at the right time to the environment. In the same way the NCL approach alone does not provide any means to control the data flow. However, the alternating data waves in combination with the completion detection signal make this approach particularly suitable to be extended by a communication protocol, which controls the data flow. Due to the event based approach and the one-hot-encoding for events, transition signalling also solves consistency and validity in the information domain. Means to control the data flow are not provided. In contrast, the micropipeline is a concert implementation of a handshake circuit and thus intended for data flow control. Sutherland suggested to combine the micropipeline with the bundled data approach to build function units inside pipe stages. Therefore consistency and validity are solved in the time domain. It is difficult to classify high level design methods due to the multitude of different techniques covered by this category. In general these methods demand some restrictions concerning input vectors, which have to be abided by the environment. Consistency and validity are largely solved in the information domain by these methods instead.

Area and energy efficiency: Other important aspects are the area and energy efficiency. Basically the number gates, which are required to implement a given functionality depend on the used design style. However specific technologies favor certain design styles – furthermore the degree of customization of basic gates has a crucial impact on the resulting circuit size. So to provide a quantitative expression not only the design style, but also the used technology (CMOS, NMOS, ...) and the degree in which basic (library) gates are adapted to a given design approach, has to be considered. The same is true for power consumption. As a consequence a quantitative analysis permits a comparison of circuits with highly specific implementations as illustrated in [57]. Instead, this section claims to provide generic overview and hence the design styles will be investigated with respect to area and energy efficiency from a qualitative point of view only. In Table 2.2 the comparison with respect to area is subdivided in three aspects: (i)*wires per bit*, which indicates the number of wires representing a bit. (ii) *gate size* this defines the number of boolean basic gates (AND, OR, INVERTER), which are necessary to build an AND-gate of the analyzed method. It is clear that specific implementations yield to a much better solution in terms of transistor count. However, we will use standard logic basis gates as a reference points, to get a suitable comparision. (iii) *add. circuits* indicates if the design technique requires additional circuits apart from the implementation of the logical function itself to build working circuits.
Based on the fact that (C)MOS poses the state of the art technology for circuit implementation, the energy efficiency can be roughly drawn back to the number of edges which occur within a circuit. Hence with respect to energy efficiency we distinguish three scenarios: (i) *worst case*, where it is assumed that the signal toggles in each cycle

from *TRUE* to *FALSE* and vice versa. (ii) *average case*, where a random distribution of the signal states is assumed, and (iii) *best case*, where the signal always keeps the same information.

	Area			Energy (transition per bit)		
	wire/bit	gate size	add. circuits	worst-	average-	best case
Sync	1	1	Clock tree	1	0.5	0
Bundled Data	1	1	Delay elem.	1	0.5	0
Huffman	1	1+	Delay elem.	1	0.5	0
NCL	2	8	CMPD circ.	2	2	2
Trans. Sig.	2	?	MullerC gate	1	1	1
Microp.	1	1	Delay elem.	1	0.5	0
H. L. Descr.	n/a	n/a	n/a	n/a	n/a	n/a

Table 2.2. Comparison wrt. Area and Energy Ef£ciency

Synchronous and bundled data approaches have similar characteristics concerning their area efficiency. The main difference lies in the method to distribute the timing information: The synchronous style uses a global time reference which is distributed over a clock tree, while bundled data uses coupled timers, which can be implemented using delay elements. The *1+* entry in the gate size column of Huffman codes should indicate that this approach can basically use the same gates as the previous methods, but an additional effort in terms of gates is required to ensure that the resulting function unit is glitch free. Using a signal coding, the NCL style requires 2 wires to represent a bit. As a consequence the size of the basic gates increases exponentially: From the true table depicted in [56] it is easy to derive that a NCL-AND gate can be built using six conventional gates (four AND and two OR gates). To guarantee that the output keeps its old value having inconsistent inputs two additional gates to memorize the output value of each wire are required.[3] Based on the bundled data approach, the micropipeline also shows its characteristics concerning area efficiency. With respect to energy efficiency the first three approaches quoted in Table 2.2 show foreseeable behavior: If the signal state does not change then no edges occurs, if the state changes each time then an edge occurs always. The NCL approach instead shows a more surprising characteristic: in each scenario (even in the best case!!) **two** edges occur per bit: Based on a RTZ scheme NCL has to transmit each information bit twice – in the first step the effective information is emitted and to return back to the neutral state the previous information has to be inverted and sent again. Also the transition signalling approach does not show any difference concerning number of edges between the best and the worst case. The reason is that the information itself is coupled to the signal edges and hence even if the same information is consecutively transmitted

[3]It is clear, that a memory element is much more complex than a simple AND gate for instance. Due to the fact that a NCL basic gate does require a full memory element, but a solution similar to a *transition gate* lasts out in a dynamic logic style, we equated these memory elements with two standard gates in Table 2.2

over the same signal line, one edge per information bit takes place. As expected the micropipeline shows similar to the area efficiency considerations the same characteristic as the bundled data approach.
At this point is important to highlight the distinction between energy efficiency and power consumption. The first one describes the energy which is required to transmit one single bit. The latter one is the energy which a circuit dissipate over time. In general asynchronous circuits operate only when it is required (=event driven), a synchronous circuit is always triggered by a periodic clock signal. Therefore, asynchronous circuits having a worse energy efficiency than the synchronous ones, may still consume less power than their synchronous counterparts.

Chapter 3

Code Alternation Logic – CAL

The CAL system was developed by Professor A. Steininger and his PhD. students W.Huber and myself. Due to this fact there are common parts in the theses of his students. In order to allow them to invest more time and as a consequence to provide a higher quality of work, Professor Steininger suggested that each PhD. student has to write only one of the common chapters, "State of the Art" and "Code Alternation Logic". Thus following chapter was written by W.Huber.

As implied by the name the major part here will consist of the coding of signals, but CAL provides much more. The system consists of a tool-set to realize asynchronous circuits which are automatically compiled in several stages. All these steps are performed with synthesis-scripts with the synopsys design compiler. Furthermore, a simulation concept is added to be able to prove the functional description of the circuit as well as to ensure the correctness of the synthesis. This tool-set allows us to design a 16 Bit processor based on CAL, and to put it successfully into operation. This chapter will give a detailed step-by-step introduction into CAL.

3.1 Background of CAL

CAL can be classified as a *signal coding method*, which solves the fundamental design problem from Section 2.2 in the *information domain*. Let us recall these terms:

With delay-insensitive circuits a method is provided to design asynchronous circuits in a way that their behavior is independent of the speed of their components or the delay on the wires. They are correct by design. A further big advantage of such circuits is that the circuit can derive information whether the computation has finished or not. Only the time needed for this computation is used for waiting rather than the worst-case time.

Signal coding describes a coding system, which is widely used to design self-timed systems. Design methods using signal coding can be split up into several approaches by means of how data is encoded. The traditional style – the 4-phase dual-rail[1] approach

[1] In this context, we use the term dual-rail to describe a signal consisting of two rails. The instance how data is coded is not defined so far.

– uses tree logic states, which can be formed with the two rails: "1", "0"and "invalid". There is a separate spacer used between every change of the state. This spacer token is necessary to distinguish whether a new data wave had begun or not. So the throughput is reduced to the half of the original one.

This disadvantage of needing such spacers is not given by the other popular dual rail technique – the transition signalling. But this approach has also its drawback: As shown in Section 2.4.5 the actual state of rails cannot be determined by just looking on it: However, an internal state "00"of this two rails could represent a logic "1"as well as a "0". It depends on the context in which this transition happened.

So the idea is to combine these two approaches and to try to eliminate the two drawbacks: On the one hand it should be possible to transport information every cycle, on the other to determine the value without considering the history. We designed a coding scheme that is based on the alternation of code sets as shown in detail in the rest of this chapter. There are two similar approaches from the early nineties: [21] introduced the *Level-Encoded 2-Phase Dual-Rail (LEDR)* and [69] named the same coding technique *Four State Asynchronous Architecture*:

Level-Encoded 2-Phase Dual-Rail (LEDR) [21] presents three different hardware implementations of the LEDR principle: The first is based on a PLA-structure, the second on a self-timed Domino logic structure with dynamic storage, and the third implementation uses series stack of transistors. There is, however, no design methodology given how to build logic with this gates. Further work in the LEDR field is done by [96, 95] where four input Phased Logic gates are used as computational elements. Here a net-list of D-Flip-Flops and combinational logic driven by a single clock can be automatically synthesized.

Four State Asynchronous Architecture This approach uses only multiplexors and the authors claim that this allows to reduce complexity. Furthermore, the multiplexors has been optimized at the transistor level and it has been implemented in 2 μm CMOS technology in 1991. This approach is optimized to speed and the best performance is given using dynamic latches because they are smaller and faster. [68]

As pointed out in Section 2.2 the fundamental design problem leads to the 2 fundamental requirements, which are the main parts of the next sections.

3.2 Coding Scheme

The key idea of CAL is to use two disjoint code sets for representing the logic state of a signal. The additional information which code set is being applied is called the *phase* of a signal, $\varphi 0$ and $\varphi 1$ respectively. The representations are used alternatively, so within a sequence of data words each bit can uniquely be assigned to the corresponding data word.

Figure 3.1 shows the flow of data waves in CAL: Due to the alternation of $\varphi 0$ waves and $\varphi 1$ waves it becomes easy to synchronize signals within a data word even in the case of arbitrary skew.

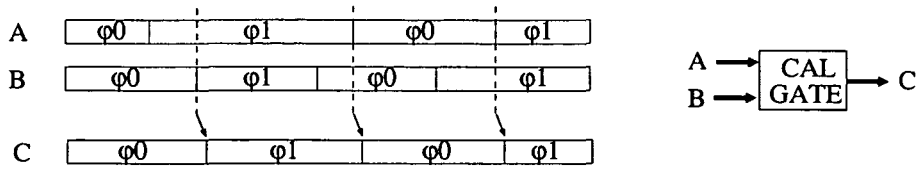


Figure 3.1. Flow of Data Waves in CAL

Two logic states in two representations lead to the need of four code words, which can be encoded with at least two rails a and b. Table 3.1 shows the used state assignment:

logic state	code $\varphi 0$	code $\varphi 1$
"LO"	(a,b)=(0,0)	(a,b)=(0,1)
"HI"	(a,b)=(1,1)	(a,b)=(1,0)

Table 3.1. CAL Coding Scheme

Table 3.1 and Figure 3.2 show the important property of CAL: If data words are coded in alternate phases $\varphi 0$ and $\varphi 1$, every valid transition from one phase to the other changes exactly one level of one rail:

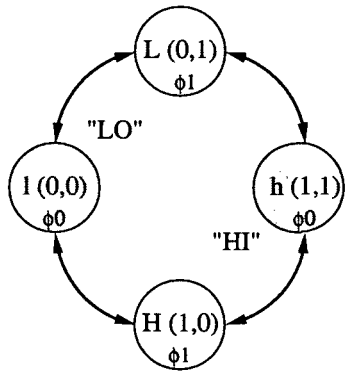


Figure 3.2. Possible Phase Transition

A logic "LO" in phase $\varphi 0$ can only be followed by a "LO" or a "HI" in phase $\varphi 1$. In the first case the level of the rails changes from (0,0) for "LO" in $\varphi 0$ to (1,0) for a "HI" in $\varphi 1$. The second transition leads to (0,1) for the representation of "LO" in phase $\varphi 1$.

As seen in Table 3.1 CAL uses a dense code which means that every bit combination is used for describing a valid code word. There is no representation for the state *invalid*. Recall that one of the three requirements in the information domain (see Section 2.3.2) is validity: In the case of CAL *continuous validity* is ensured. So every gate has to guarantee a valid output signal. As described above exactly one transition is needed to change from one valid code word to another valid one. This fulfills both conditions for *consistency* needed as second part of the fundamental requirement: The demand for the *existence of a transition* is given due this to fact as well as the *membership to the contexts*: If there is *exact* one transition between every code word, every transition

will change the context and so the membership can be derived. The impact on CAL designs leads to the following important rules which are summarized here:

I: Data values of each signal must be coded in alternating phases.

II: The calculation is performed when all input phases are in the same phase.

III: In the case that the input signals are in different phases the output remains in its last valid state.

Until now it looks as if CAL solves all problems in the information domain – in other words it is delay insensitive. In fact CAL is a hybrid solution as described in Section 2.3.3. Concerning a higher level – a design built with CAL-gates – the CAL approach is *delay insensitive*. There are no assumptions made neither on the gate delays nor on the wire delays. A closer look at the CAL gates shows that there are timing assumptions, e.g. local feedback loops in latches. The resulting constraints for the design can be solved within the basic gates. The information to build these gates in a correct manner is stored in specific libraries.
Both validity and consistency are needed to solve the *fundamental requirement 1*. The second one will be the target of the next section.

3.3 Control Flow

The design rules of Section 3.2 must be true for the whole design, so they must be valid for pipeline structures too. Rule I defines that the code set used in CAL alternates with every data word. This means that a bit that has been part of a valid code word in $\varphi 0$ becomes invalid in $\varphi 1$. Recall the *fundamental requirement 2* from Section 2.2 where some kind of feedback is needed. Figure 3.3 shows the pipeline structure where the feedback is represented in terms of capture_done signals to trigger the source firing. The source can derive the trigger condition directly from the data wave: If all bits of a data word are in $\varphi 0$, the data word is consistent and can be consumed. As soon as several bits change to $\varphi 1$, the $\varphi 0$ bits become obsolete and the data word is inconsistent until the last bit has changed to $\varphi 1$ as well. Obviously, some kind of synchronization is required to prevent that a fast $\varphi 0$ bit, e.g., catches up with the preceding $\varphi 0$ data wave.

This is, however, easy to achieve by the inclusion of a hysteresis in the logic functions: Similar to the approach used in NCL the output of a logic gate in CAL changes only when the data word at the input is consistent as defined in rule III. In Figure 3.3 a simple linear pipeline is shown:

To explain the functionality of the pipeline structure the stage in the middle is used. There are two conditions when this stage fires:

1. The upstream logic function $f(x)$ has completed the calculation and so the data on the input of this stage is ready to be captured. This information can be retrieved directly from the data word.

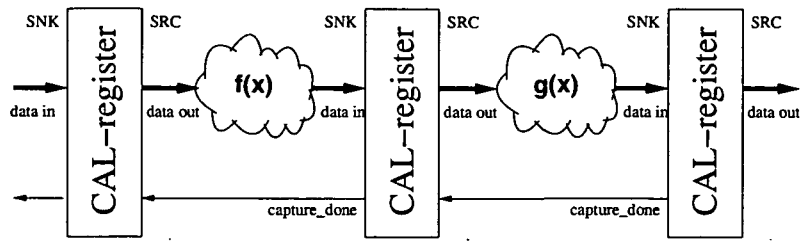


Figure 3.3. CAL Pipeline Structure

2. The downstream stage has already caught the previous wave (the result of $g(x)$) and so the data of this stage is not needed any longer. The downstream stage provides this with the *capture_done* signal.

Recall Figure 3.1 which shows the flow of data waves in CAL: Due to the alternation of $\varphi 0$ waves and $\varphi 1$ waves it becomes easy to synchronize signals within a data word even in case of arbitrary skew. It can be verified that all three rules defined in Section 3.2 are fulfilled.

3.4 Levels of Abstraction

It is not very comfortable to design logic circuits using a rail representation as described in Table 3.1. Furthermore, it is not possible to use existing synthesis tools, because they are designed for single rail logic used in synchronous designs. This leads to the need of two different descriptions for CAL: One for the designer and another one for the tools. Both definitions are written in *VHDL* in our case, but it is also possible to transform the representations to Verilog or any other hardware description language.

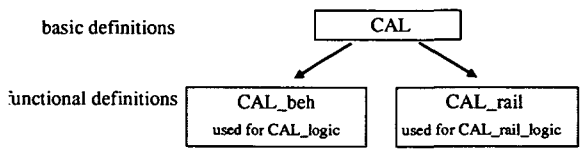


Figure 3.4. Library dependencies

As shown in Figure 3.4 the library structure is built hierarchically: The CAL library is the root of all other libraries and provides basic type definitions for all others. All common definitions for the behavioral and the rail style of CAL are given here. Furthermore, some basic conversion functions are provided. This library will be used in every step of the asynchronous design as well as in the testbench. As an addition to the CAL library the `cal_beh` and `cal_rail` contain functionality needed for the corresponding logic. In both logic systems – `cal_logic` and `cal_rail_logic` – for example a logic AND is available. In `cal_logic` this function has two single rail inputs and one single rail output, while in `cal_rail_logic` the same function requires dual-rail signals. In summary, these two libraries provide functions with the same purpose

but with the logic types needed for the logic system actually used – `cal_logic` or `cal_rail_logic`.

3.4.1 Behavioral Description – cal_logic

The definition of `cal_logic` is the interface for the human designer. A single rail, multi value code is used to describe the four states of CAL. As shown in Table 3.2 the different states are specified with lower and uppercase letters "l" and "h". For the phase $\varphi 0$ the lower case versions "l" and "h" are used for the logic states "LO" and "HI" and likewise "L" and "H" are applied for $\varphi 1$.

logic state	code $\varphi 0$	code $\varphi 1$
"LO"	l	L
"HI"	h	H

Table 3.2. `cal_logic` **Coding Scheme**

To provide full simulation and synthesis support of traditional design tools it is necessary to define several types and classes. At first, a new data type has to be declared. In the case of `cal_logic` the four states have to be defined. Furthermore, it is not enough to build a four-value type, because a reasonable simulation tool needs more possible values. There has to be a value which sets a signal to undefined, e.g. at the startup. Moreover, the simulation should be able to handle the situation when two outputs drive one signal and both of them want to assign a different value. This definition is very similar to *std_logic* data type for the *std_logic_1164* standard for the *VHDL* language [5]. Furthermore, the type is expanded to a vector of n such signals and so the `cal_logic_vector` type is created. As shown in Source 3.4.1 the VHDL definition for the `cal_logic` type consists of eight characters:

```
type cal_ulogic is ( 'U',          -- Uninitialized
                     'X',          -- Forcing  Unknown
                     'l',          -- 0 type phi0
                     'h',          -- 1 type phi0
                     'L',          -- 0 type phi1
                     'H',          -- 1 type phi1
                     'Z',          -- High Impedance
                     '-'           -- Don't care
                     );
```

Source 3.4.1: `cal_logic` VHDL definition

As described above, the definition and some basic conversion functions, e.g. from `std_logic` to `cal_logic` and vice versa are part of the `cal` library. The definition of the data type is the starting point of the whole system to build logic devices with CAL. Furthermore, several logic functions have to be designed to support the simulation and

the synthesis of CAL designs. These are special parts for the behavioral description of designs with CAL and therefore they are part of the `cal_beh` library.

Boolean functions: To build designs various functions have to be defined. Such functions describe the relationship between the inputs and the output. They are also used by the synthesis to build e.g. conditions of *if*-clauses. If we consider a two-input AND-gate, we can define the function between the two inputs and the output in case the two input signals are in the same phase. So rule I and II from Section 3.2 can be implemented. Considering the condition in an *if*-clause again, it is not possible to use methods which use any kind of history or context. Therefore, it is not possible to remain with in the old state with simple functions, because they can just derive the new value. So it must be ensured, that these gates process only input signals that are in the same phase. This is done by inserting so called *stable*-procedures into the *VHDL* code.

***stable*-procedure :** In VHDL this procedure is inserted into the behavioral code to ensure that the VHDL-process continues only if all inputs of this *stable*-procedure are in the same phase. So rule III can be enforced. The procedure is implemented with VHDL *wait until*-statements to suspend the current process until the condition is met. Notice, that this function is only necessary in `cal_logic`.

Register and latches: One of the big differences between `cal_logic` and usual synchronous designs is the methodology by which storage elements are implemented. In the case of synchronous designs this is usually done with clock edges. As shown on the left side in Source 3.4.2, the active clock edge is the point in time where the current value is accepted and frozen:

```
p2_SM : process (clk, reset)
begin
  if reset = RES_ACT then
    Pc          <= (others => '0');
  elsif clk'event and clk = '1' then
    Pc          <= PcNxt;
  end if;
end process p2_SM;
```

```
p2_reg: cal_reg
    generic map (
      w => 108,
      reset_value => 01)

    port map (
      d       => PcNxt,
      q       => Pc,
      c_done  => c_done,
      pass    => pass,
      reset   => reset);
```

Source 3.4.2: Register Implementation in `std_logic` and CAL

The right side of the source code shows the register implemented in CAL. Both implementations have input (*PcNxt*), output (*PC*), the reset signal, and the value which should be used after reset. In the synchronous approach *others =¿ '0'* is used to specify the value after reset the CAL uses *reset_value =¿ 01* as a generic map. The big difference is given when the register stores the data. In the

synchronous version it is done with the rising clock edge. In contrast CAL uses a handshake protocol (*c_done and pass*).

Conversion functions: A CAL design should be able to interact with "normal" `std_logic` circuits as well as the environment. For this purpose a set of conversion functions is needed. In the case of `cal_logic` the transformations are done by simple translation tables.

The issues above have been described in detail for the behavioral description, because they constitute the main differences between the synchronous design and the CAL logic design. The process of transforming a regular synchronous design to CAL starts with renaming the data types from `std_logic` to `cal_logic`, followed by inserting the *stable*-procedure to ensure rule III. Furthermore the registers must be converted from the *if clk'event* style to the instances of the `cal`-register and the required acknowledge signals. To interact with the environment the appropriate conversion functions must be applied.

3.4.2 Functional Description – cal_rail_logic

Table 3.3 shows the `cal_rail_logic` type consisting of two rails of the `std_logic` type. The two rails are bound together and have one name.

logic state	code $\varphi 0$	code $\varphi 1$
"LO"	(0,0)	(0,1)
"HI"	(1,1)	(1,0)

```
type cal_rail_logic is
  record
    line1 : std_logic;
    line0 : std_logic;
  end record;
```

Table 3.3. `cal_rail_logic` Coding Scheme and the VHDL Definition

Boolean functions: All logic functions are available as pre-synthesized elements. So only existing functions are used and the design consists of instances of them. In the case of `cal_rail_logic` *AND*, *OR*, and *IV* are defined and all other logic functions are put down to them. Notice, that here the gates fulfill the rules I – III themselves because each of them is built with a kind of hysteresis or a memory element as seen later.

Special gates: For the synthesis of CAL a set of specialized gates is needed. For example, the φ-detector or the components of the `cal`-register are some of them. The gates and their functionality are defined and so they are available for the rest of the design flow.

Conversion functions: The transformation from `cal_rail_logic` to `std_logic` logic is quite easy, because a-rails in CAL directly represent the signal state in Boolean logic. Hence, in the inverse case only minor coding effort is required to add the adequate phase to the conventional Boolean signal.

The implementation of some selected gates is presented in the next chapter.

3.5 Basic Gates

To illustrate how logic functions can be implemented in CAL we discuss the example of a 2-input AND here. The derivation of the required functions is quite straightforward, and essentially the same is true for other basic functions such as OR, NAND, NOR and XOR.

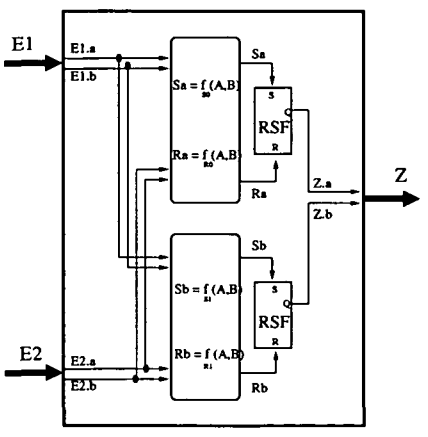
3.5.1 AND Gate

Table 3.4 shows the truth table on the signal level:

Z		E1			
		h	l	H	L
E2	h	h	l	hold	hold
	l	l	l	hold	hold
	H	hold	hold	H	L
	L	hold	hold	L	L

Table 3.4. Truth Table of a 2-input AND in CAL

For inputs that are within the same phase the respective AND function is simply applied and the output is represented in the same phase. For inputs in different phases the last valid output is retained ("hold"). On the rail level this truth table has to be expanded to two rails per signal, yielding one separate truthtable for each rail of the output – Z_a and Z_b, each with four input rails as shown in Figure 3.5:



E1.a E1.b	E2.a E2.b	Z.a	Sa	Ra	Z.b	Sb	Rb
00	00	0	0	1	0	0	1
00	11	0	0	1	0	0	1
11	00	0	0	1	0	0	1
11	11	1	1	0	1	1	0
01	00	HOLD	0	0	HOLD	0	0
01	11	HOLD	0	0	HOLD	0	0
10	00	HOLD	0	0	HOLD	0	0
10	11	HOLD	0	0	HOLD	0	0
00	01	HOLD	0	0	HOLD	0	0
00	10	HOLD	0	0	HOLD	0	0
11	01	HOLD	0	0	HOLD	0	0
11	10	HOLD	0	0	HOLD	0	0
01	01	0	0	1	1	1	0
01	10	0	0	1	1	1	0
10	01	0	0	1	1	1	0
10	10	1	1	0	0	0	1

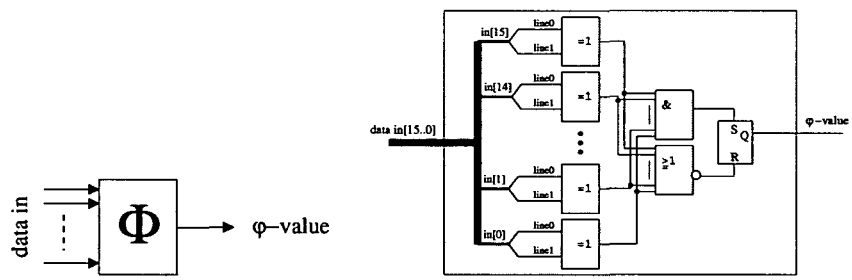
Figure 3.5. Schematic and Truth Table of the AND-gate

The resulting circuit for one AND-gate consists of two RS-flip-flops – one for each rail a and b of the output signal Z. Furthermore, for each of the RS-FF's logic functions are used to derive the correct set and reset action. This results in the need of four 4-input and 1-output functional blocks for set and reset: R_a, S_a, R_b, and S_b.

The initial hardware implementation requires 6 logic elements (LEs) for one CAL-AND-gate. In comparison with a standard AND the gate count increases significantly, but it should be considered, that we are mapping the design to a standard FPGA library that has not been specifically optimized for CAL.

3.5.2 Phase Detector

Considering that there are two possible phases for each signal which is used to associate a bit to a data word there is the need to detect the phase of a signal. This is very simple for a single signal: Both rails have to be combined with an XOR and the result is the phase – *0* for the phase $\varphi 0$ and *1* for $\varphi 1$. As shown in Figure 3.6(b) this scheme can be expanded to an n-bit wide bus: The rails of each single signal are combined with an XOR-gate and the n results are tied together with an and-gate ("all-ones detector") and an or-gate ("all-zero"detector). The RS-Flip-Flop ensures that the output only changes if all inputs are in the same phase as demanded by rule III. This circuit acts as a multi-input Muller-C gate.



(a) Symbol (b) Implementation

Figure 3.6. The φ-detector

Notice that the φ-detector can also be used for completion detection, because the value at the output changes only if *all* input values are in the same phase. This is necessary, e.g., for register implementation used in pipeline structures.

3.5.3 φ-Converter

Sometimes it is necessary to convert the phase of a signal. Remember the pipeline of Figure 3.3 and consider the case that the signals from the first and second stage are both inputs of the same gate. So the values should be used when the signals are in different phases. Here a φ-converter is used to convert the phase of one of the signals so that they can be combined. Fortunately, the implementation is very simple:

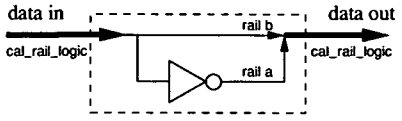


Figure 3.7. Implementation of a φ-Converter

Due to the fact, that only one bit may change, the delay of this one inversion cannot cause an invalid output as a result of skew. If *rail a* changes due the phase change, the result will be delayed. When *rail b* changes there is no impact on the circuit caused by the φ-converter.

3.5.4 CAL Register

The implementation of registers used in the pipeline structure in Section 3.3 is now discussed in detail. In Figure 3.8 the proposed implementation of such a register is shown. The chosen implementation represents a hybrid solution (see Section 2.3.3) to solve the fundamental design problem. As described further the solutions in higher abstraction levels are done in the information domain and on this higher level there are no requirements on the design in terms of delay and skew. The needed timing assumption on gate level must be met inside one register. If we can guarantee these requirements on this local area, the registers can be used without paying attention to the timing.

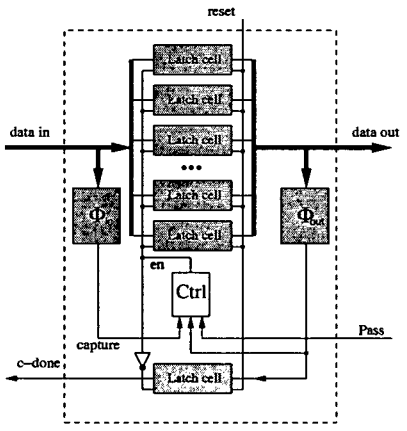


Figure 3.8. Implementation of CAL Register

The basic function of a register in a pipeline is given in Section 3.3. Remember that the latches get transparent if (i) the phase at the input differs from the phase at the output and (ii) the phase of the downstream stage is the same as the phase stored in the latches (i.e. visible at the output). Condition (i) can be checked by comparing the outputs of the φ-detector, both at the input (Φ_{in}) and at the output (Φ_{out}). Condition (ii) implies that the pass signal from the downstream stage equals the output of (Φ_{out}). The Latches will lock if the phase on the output is equal with

that on the input ($\Phi_{in} = \Phi_{out}$). The *reset* signal is used to start the latches up with a predefined state.

There is one important detail. The *capture done* signal is generated by a latch with an inverse *en* input. That shall ensure that the *c-done* signal is not given to the upstream stage before the latches have actually stored their values. This works under the following timing assumptions:

- All latches must have the same *gate-delay*. This can be ensured when all of them are taken from the same library, so that they are built equally.
- The *en*-signal for the latches inside the registers must hold the isochronic fork assumption, this can be achieved by a well known routing process.

Further discussions on this topic will be given in [117]

3.6 CAL Design-Flow

We have described the basic gates so far. However, now we need a methodology to build hardware from a description of the design. Similar to the synchronous case there should be a behavioral description as a starting point. If the description meets the specification it is the input of a tool chain which generates the associated hardware.

Therefore, as outlined in Section 3.4.2 we have defined a type to describe each signal with a single-rail 4-value data type called *cal_logic*. In a library the basic boolean functions for this type are defined and so the design can be simulated on behavioral level. At this state the design is described with *cal_rail_logic*. Recall that the data type used in this description consists of two rails of conventional 2-value `std_logic` signals. The steps performed to get a design in `std_logic` vectors to be placed and routed with usual tools are described in Figure 3.9.

The difference between the conventional Design Flow and the approach used with CAL logic is clearly visible: Both approaches start with a behavioral description and the result of each of them is a description understood by the place&route tool. This final description may only use gates of the target library – Altera APEX (see Section 4.2.1) in our case. After performing this last step the design can be downloaded to the FPGA.

In the conventional case the *VHDL*-code is elaborated and transformed into an intermediate language used by the synthesis tool. This functional description is the starting point for the synthesis during which the design is finally mapped to gates of the target library. In our case this is the *APEX*-library. As a result we get the prelayout representation of our design. This file is used for simulation on the one hand and as input for Quartus to perform place&route and the download to the FPGA on the other hand.

The result of the elaboration step performed in the CAL design flow is the functional description where the design is built with CAL gates. The functionality of the CAL gates is described in a special library (CAL-beh) to facilitate the simulation, which is described in detail in Section 3.7. For the synthesis an other library is needed which provides synthetic operators to build design specific gates. One of this operators is used to build a φ-detector with the width needed by the design. So with the first

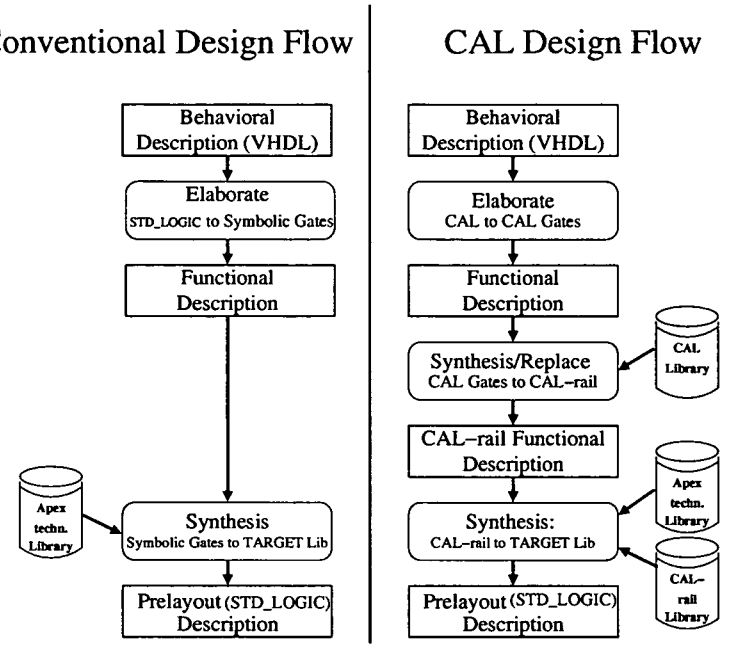


Figure 3.9. CAL-Design Flow

synthesis the design is transformed from the four-value *cal_logic*-description to the dual-rail *cal_rail_logic*. As described above this representation uses pairs of *std-logic* signals and the functionality of the gates is provided by the *cal_rail*-library. This representation is used for simulation purposes as well as input for the second synthesis which is very similar to the synthesis in the synchronous case. The APEX technology library is used as target library which results in a design constructed with APEX-gates.

It is important to note that the design flow allows us to change the actual type of pipeline register used for synthesis quite easily, because the functionality is added by the appropriate library. This led us to experiment with several implementation options that all turned out to have their specific benefits and drawbacks. A discussion of these different options will be the focus of [117].

3.7 Simulation Concept

In this section the simulation of a CAL design is discussed. At the beginning the four simulation steps shall be defined as follows (compare Figure 3.9):

behavioral simulation Simulation with `cal_logic` signals and any timing information of the resulting hardware. The input is the source code of the designer without any synthesis applied.

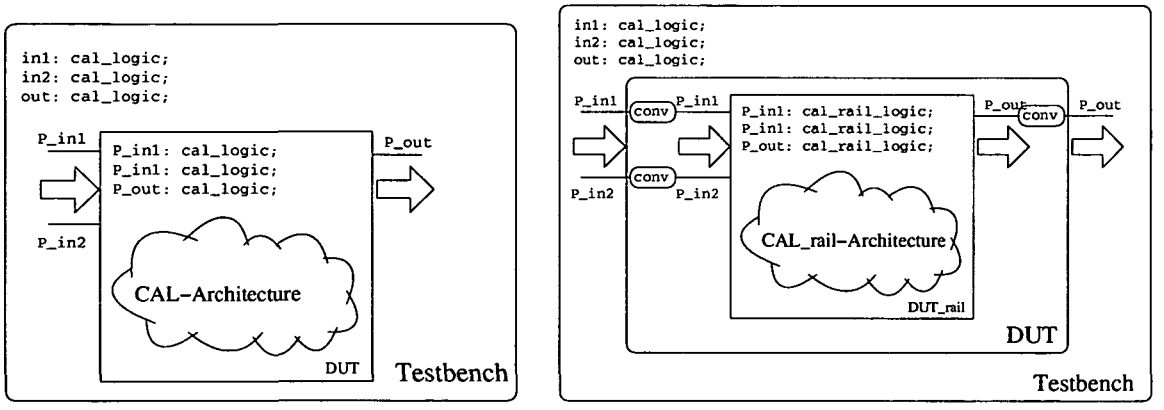
functional simulation The first synthesis has already transformed the `cal_logic` code is into the `cal_rail_logic` format. However, no timing information has been added in this step.

pre-layout simulation The second synthesis has mapped the circuit to the target library – in our case the Altera APEX library. The functionality is constructed just with gates of the target library and so the number of gates and their standard delay is known and used for the simulation. However, the place&route has not been performed and for the delay of the wire just default values are used.

post-layout simulation This representation contains the whole timing information of the design. Every gate as well as each wire delay is known an used for the simulation. This leads to a high complexity of the simulation and consequently to a very long simulation duration.

The motivation for a clever simulation method is clear and it is based on the design flow: The data types of the signals change with every step towards the real hardware. Still it should be possible to use the same testbench for all four simulation levels. As described earlier in this chapter the starting point of the designs is the behavioral style – in our case `cal_logic`. The signals in the design as well as the ports are `cal_logic`. With the next step these types are transformed to `cal_rail_logic`. Therefore, the ports are also translated and the signals with the same names as before are now composed of `cal_rail_logic`.

The two rails are combined with the specific type to one record. After place&route the design consists of `std_logic` signals and so the ports are converted once again. Furthermore, the number of ports doubles with the last step and so each signal becomes a vector of two `std_logic` rails. In the same way the width of each vector doubles.



(a) Behavioral Simulation

(b) Functional Simulation

Figure 3.10. Simulation Concept

Although with each step the level of detail and therefore the refinement of delay increases, this three formats still represent the same design with the same functionality. The testbench is also written by the designer and therefore the `cal_logic` style is used. As shown in Figure 3.10(a) it is straightforward to perform the first simulation – the

behavioral simulation – because the types of the ports match with the signal types of the testbench.

The next simulation steps cannot be performed so easily. Here the types of the ports out of the device under test (DUT) do not equal those of the testbench. Conversion functions have to be inserted to connect the DUT to the signals of the testbench. As shown in Figure 3.10(b) this is done automatically by our tool: To be able to simulate a design a *configuration* is used anyway to select and combine the architecture for a specific entity. By means of some scripts we create a new architecture *DUT* in which the original design *DUT_rail* is instanced. The architecture itself consists of just this instance and the appropriate conversion functions. So while the designer has to write the testbench and the first configuration as in the synchronous case, the CAL specific parts are generated automatically.

In Figure 3.11 the postlayout simulation example shows the value of the program counter and the output of a ROM:

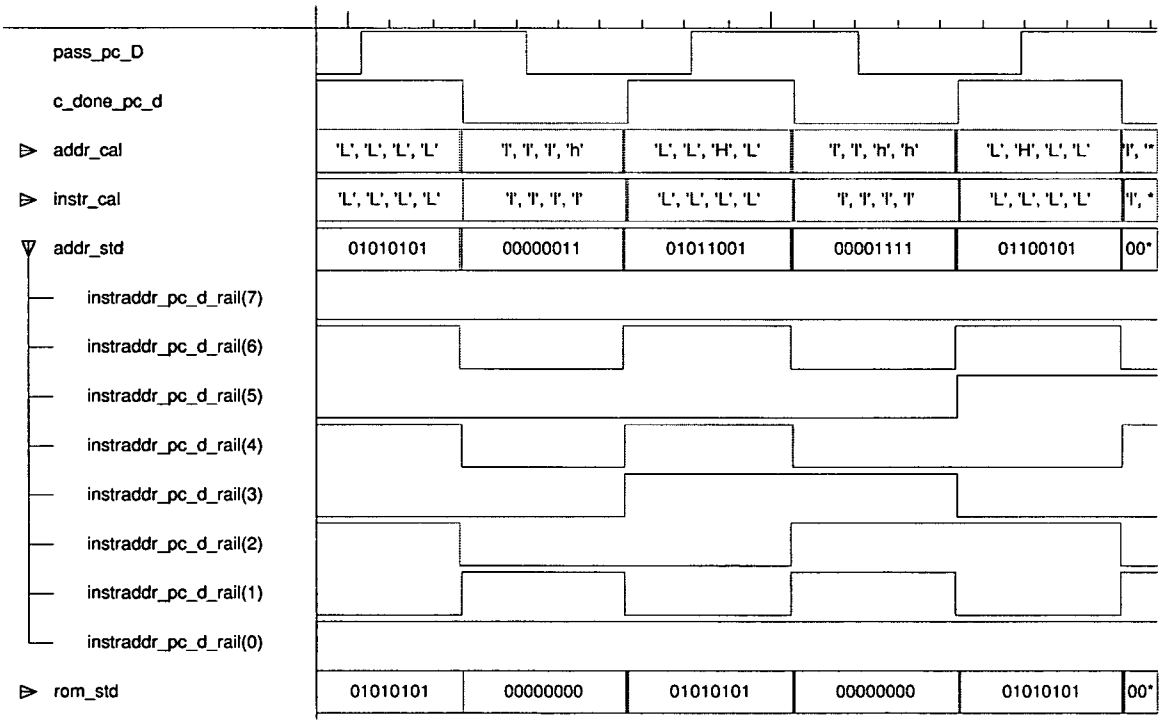


Figure 3.11. Postlayout Simulation Example

In the first two lines the *pass* and *capture_done* signals are shown, followed by the address and the instruction. In this example only the lowest four Bits are shown. In line tree and four they are shown in `cal_logic` style as they can be seen at every level of simulation. This is followed by the signals without the mapping to `cal_logic`. Every vector consist of eight `std_logic` signals, those of address is shown in detail. As depicted in Figure 3.11, it very difficult to derive the value of the busses from the `std_logic` description: The address is incremented by one every step and the instruction remains at zero.

This strategy has finally allowed us to reach the goal mentioned at the beginning of this chapter – one testbench for all simulations. In the same way it is possible to automate the verification process: The behavioral simulation must be checked by the designer manually whether the specification is met or not. If the wanted functionality is given the remaining simulation steps are performed by the tool and the result can be crosschecked with those from the behavioral simulation automatically.

3.8 Summary

CAL is a design technique using signal coding and a dense code where two representations – one for $\varphi 0$ and one for $\varphi 1$ – of each logic value "LO"and "HI"are given. Our approach is similar to NCL with some important advantages: There is no need for the so called spacer or NULL-waves in CAL which doubles the throughput compared with NCL. Furthermore, the energy overhead in terms of transitions per bit is low: Exactly one rail transition per bit is required.

CAL is classified as hybrid solution to manage the fundamental design problem. A design built just with CAL-gates is delay insensitive and so validity and consistency are needed to tackle the problem in the information domain. The basic gates have internal delay assumptions yielding to design constraints – this is the part of the system solved in the time domain. The implementation of basic gates is demonstrated on appropriate candidates: The internal structure of an AND-gate as well as a complex CAL register is described in detail.

The human interface to build CAL circuits – `cal_logic` – and the coding style on gate level – `cal_rail_logic` – are introduced. Furthermore, the methodology and the used libraries for the CAL design flow demonstrate the automated way from the design written by the engineer to the download file. This and the simulation concept show the practical applicability of our CAL approach.

Chapter 4

Prototyping Environment

In this chapter the environment for the evaluation is presented: The synchronous reference design is shown, which is the starting point of our asynchronous implementation. The motivation to build a processor ourselves was the possibility to have a deep knowledge of design details, because it is very hard to derive the internal functionality from a standard microprocessor – like an ARM. Furthermore, the dependencies between the control signal among pipeline stages are very hard to explore, which is, however, one of the key points of our design. To avoid such troubles we decided to build our own processor – SPEAR.
The target platform for the design is an FPGA evaluation board. In the following a look at the underlying concepts and the evaluation boards is given and the advantages and drawbacks of the FPGA implementation are discussed.

4.1 The SPEAR Processor

4.1.1 Core Architecture

SPEAR is the acronym for "Scalable Processor for Embedded Applications in Real-time environments" [23] and the main goal of several design decisions [22] was to build a processor which has a well known temporal behavior [24]. The processor executes *every* instruction in exactly one cycle and the instructions are also one word wide. The SPEAR design has been developed to provide moderate computational power and represents a RISC architecture which executes instructions through a three-stage-deep pipeline. The instruction set comprises 80 instructions, further a compiler suite [51] comprising the GCC [92] and the LCC has been developed supporting this instruction set. Most of these instructions are implemented as *conditional instructions* [93] which means an instruction is executed or replaced by a NOP depending on the condition flag. A preceding test instruction sets this flag once and it is valid until the next test instruction. For example, a move instruction with condition false is executed when the result of the test instruction is false.

Instruction and data memory are both 4 kB in size, but it is possible to add up to 128 kB of external instruction memory and 127 kB of additional data memory. The

uppermost 1 kB of the data memory is reserved for memory mapping of the extension modules. These modules (see Section 4.1.2) are used to customize SPEAR to the needs of the environmental interaction. As a result of the memory mapping, no dedicated instructions for extension module access are needed – common load/store instructions are used – which satisfies the RISC [44] philosophy of our approached design. The register file holds 32 registers which are split up into 26 general purpose and 6 special function registers, three of them are used to construct stacks efficiently using frame pointer operations. The remaining three are used to save the return address in case of an interrupt or subroutine call. SPEAR supports 32 exceptions, 16 of them are hardware exceptions – interrupts – and 16 can be activated by software, we call them traps. The entries of the exception vector table hold the corresponding jump addresses to the interrupt/exception service routines for each interrupt or exception. The SPEAR ALU performs all provided arithmetic and logical functions, but it is also responsible for offset calculation on jumps. Furthermore, the ALU is used to pass through data from the exception vector table or register file. Figure 4.1 shows a block diagram of the SPEAR processor.

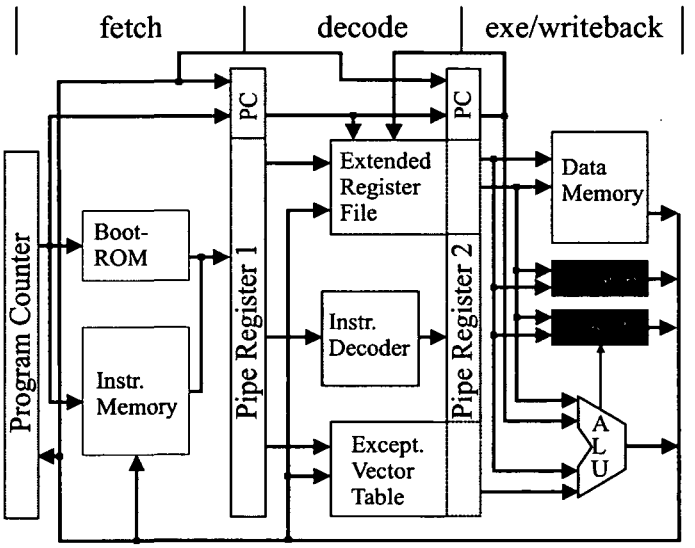


Figure 4.1. SPEAR Architecture

The SPEAR pipeline is structured into an instruction fetch (FE), an instruction decode (DE) and a combined execute/write-back (EX/WB) stage. In the fetch cycle, the instruction memory is accessed and one instruction opcode is passed to the decode stage. During the decode cycle the control signals for the memories and the ALU are generated, furthermore the operands of the instruction are retrieved from the register file. The execute/write-back stage performs the intended operation of the instruction and writes the resulting value to the appropriate memory location. When an extension module access (EXT) happens, it is also executed during the EX/WB cycle.

4.1.2 Extension Modules

As mentioned above extension modules are used to fit the processor for different applications. For reasons of simplicity and lucidity, the integration of and the access to extension modules should work the same way. Thus a generic interface for all extension modules has been defined [46]. All extension modules are mapped to a unique location at the uppermost region of the data memory. The modules are accessed via eight registers using simple load and store instructions, as from the processor's point of view the extension modules are simply memory locations. A block diagram of the generic extension module interface is shown in Figure 4.2. The first two registers are the *status* and *config* register of the module. The *status* register tells the processor the current state of the extension module. Among other things it shows if an interrupt has been activated, an error has occurred, or if the extension module is still busy. The *config* register is used to specify parameters for the operations of the module. Next to a soft-reset bit, which is used to deactivate the extension module, an interrupt acknowledge bit exists to reset the interrupt status. The remaining six registers Data 0 – Data 5 are available for module specific issues.

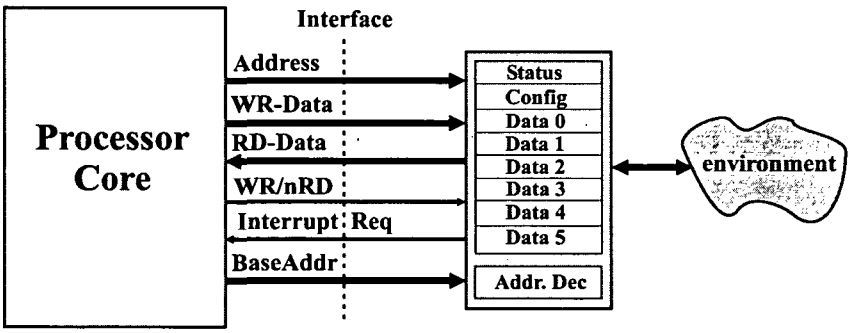


Figure 4.2. Generic Extension Module Interface

There is a special extension module – the *processor control unit* which has to be used in every design. It comprises functional blocks which are essential for the processor, e. g. the *processor status word.*

4.1.3 Implementation Results

Some implementation details are given here to finish the description of the synchronous reference design: Our processor SPEAR utilizes 1,794 logic elements of the APEX20KC FPGA (see 4.2.1). This is about 15 % of the total number of logic elements. Further, the on-board data and instruction memories as well as the register file use more than 70,000 memory bits - which is about 47 % of the number available. Finally, SPEAR runs with a maximum clock frequency of 46 MHz on this FPGA.

4.2 The Hardware Platform

The target technology for the synthesis and the following place&route steps are FPGAs[1]. The decision to build hardware on FPGAs instead of using full- or semi -custom ASIC-chips is based on the fact, that it is much faster and much cheaper to get a prototype. The SPEAR processor as well as the asynchronous designs should be tested as a physical implementation to prove the functionality – e.g. by displaying several buses on a logic analyzer. Modern FPGAs are nowadays quite fast and big enough to contain a processor design. Unfortunately, the use of FPGAs does not only cause advantages: The performance of a processor built with the FPGA basic gates is not as high as the value which can be reached with an ASIC design, but designs should be proof-of-concept and therefore the performance is not the key achievement.

Our prototyping board called megAPEX [6] is built by *El Camino* and it is equipped with an FPGA out of the APEX Family, which is described in detail in the following section.

4.2.1 APEX FPGA Family

An FPGA (Field Programmable Gate Array) is an integrated circuit that consists of an array, or a regular pattern, of logic cells. The logic cells can be configured to represent a limited set of functions. These individual cells are connected by a matrix of programmable switches. The developer's design is implemented by specifying the logic function for each cell and selectively closing switches in the interconnect matrix. The array of logic cells and the interconnect matrix are taken form a set of basic building blocks for logic circuits. These basic blocks are combined to achieve the intended behavior of more complex designs.

The logic cell architecture varies between different device families. In general, each logic cell combines a few binary inputs (typically between 3 and 10) to one or two outputs according to a boolean logic function specified in the programmed design. In most FPGA families, there exists the possibility of registering the combinatorial output of the cell, so that clocked logic (like counters or state-machines) can be implemented easily. The combinatorial logic of the cell can be physically implemented as a small look-up table (LUT) or as a set of multiplexors and gates.

The APEX family represents highly integrated FPGA devices which are manufactured in 0.22 μm to 0.15 μm processes. APEX devices are available in ranges from 30,000 to over 1.5 million gates. The APEX architecture consists of so-called MegaLABs[7]: These function blocks can be connected with each other as well as to I/O Pins. LUT-based logic provides optimized performance for data-path and register-intensive designs, whereas product-term-based logic is optimized for combinational paths, such as state machines. Embedded system blocks (ESB)[7] can implement a variety of memory functions, including first-in-first-out (FIFO) buffers, ROM or dual-port RAM functions. The ESBs support memory block sizes of 128x16, 256x8, 512x4,

[1]We use the term FPGA for off-the-shelf components. However, there are some approaches for bundled-data systems STACC[87], PGA-STC[59] and for general purpose architectures – Montage[43].

1024x2 and 2048x1, but can be cascaded to implement larger sizes. The MegaLAB Structure comprises a set of logic array blocks (LABs), one ESB, and a MegaLAB interconnect, which routes signals within the MegaLAB structure. The amount of LABs inside each MegaLAB depends on the specific APEX device, and can range from 10 to 24 LABs. Signal interconnections between MegaLABs and I/O pins are provided by the FastTrack Interconnect, a set of fast column and row channels (additionally LABs at the edge of MegaLABs can be driven by I/O pins via the local interconnect).

Each LAB consists of 10 logic elements (LE) and the associated local interconnect. Signals are transferred between LEs in the same or adjacent LABs, ESBs or IOEs via high-speed local interconnects. The LAB-wide control signals can be generated from the LAB's local interconnect, global signals, or dedicated clock pins.
The logic element (LE), the smallest addressable logic unit in the APEX architecture, is very compact and provides efficient logic usage. Figure 4.3 shows a block diagram of an LE. Each logic element contains a four-input LUT, which is a function generator that is able to implement any function of four input variables. Furthermore, carry and cascade chains as well as a programmable register for D-, T-, JK-flip flop and a shift register implementation are part of each LE. LEs can drive the local interconnect, the MegaLAB interconnect, and the FastTrack interconnect structures.

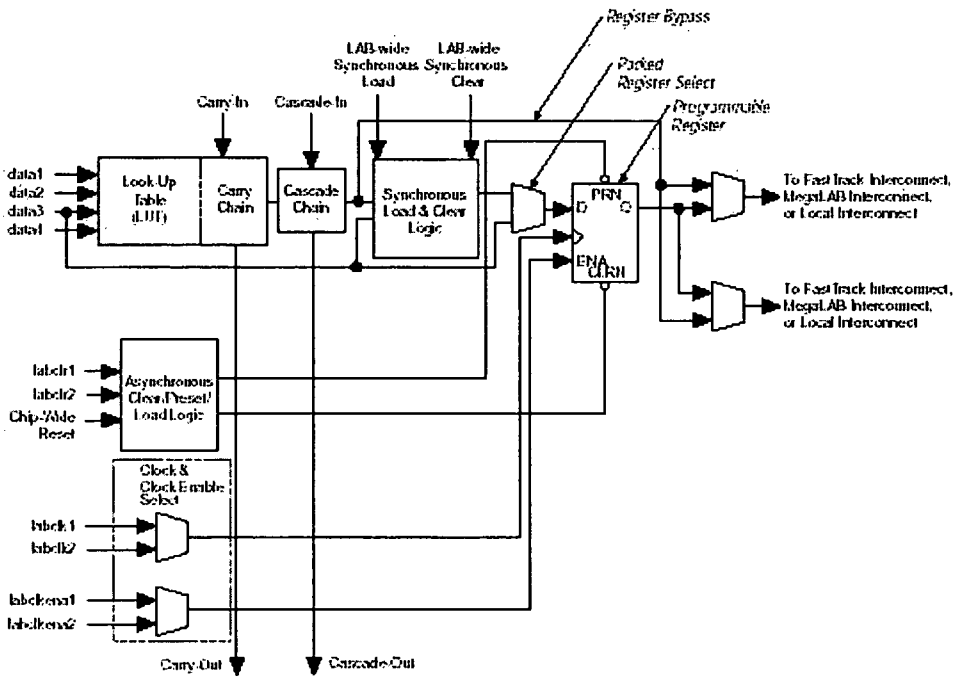


Figure 4.3. Logic Element Structure [7]

For our experiments we use 20KC1000 devices, that feature the 0.15 μm process and all-layer-copper interconnect. This FPGA is equipped with 38.400 Logic Elements – it is comparable to 1,000,000 typical gates. Further information can be found at [7].

4.2.2 Limitations

FPGAs are designed and optimized for synchronous designs and this clearly has an impact for the implementation of purely asynchronous circuits. Our experience with APEX devices lead to the following points:

Wire delay: As mentioned in the introduction the wire delay gets more and more important in the chip design. In ASICs this drawback can be tackled by optimizing the routing. In FPGAs, however, this is not possible, because the wires are built during the manufacture of the FPGA and only the interconnects are programmed by the design. This leads to longer wires and thus to a larger delay. It can be seen that the wire delay limits the performance in synchronous FPGA designs. The design of the super-scalar variant of the SPEAR namely LANCE shows this effect [35].

Logic elements (LEs): As shown in the section above, Altera FPGAs are composed of LEs. Four input signals can be combined to one output. This does not meet our requirements: In a CAL design each gate has a dual-rail output and in the case of feedbacks it has more than four inputs. If more than four inputs of one output are required, additional LEs have to be utilized and so the design grows very fast.

Synchronous register: Every LE is equipped with an edge-triggered register which reflects the optimization for synchronous designs. In the case of CAL however, they are useless.

RS-flipflops: In basic gates (see 3.5) an RS-flipflop is used as a memory cell to hold the old state of the output. Unfortunately, the APEX FPGA does not offer an RS-flipflop as a component in an LE. It must be built with an LE and an external feedback. This external feedback can lead to problematic race conditions with other signals.

Place&route tools: The tools for place&route as well as the timing analyzing tools are also optimized for the use with synchronous designs. They are built to optimize the register to register delay. This leads to very long execution times for the tools as well as to not optimized results for asynchronous designs.

In summary, FPGAs are principally not intended and well suited for asynchronous logic designs. Asynchronous designs implemented in FPGAs have many disadvantages compared to synchronous FPGA designs on the one hand and asynchronous ASICs on the other. Still we found the reconfigurability of the FPGA platform worth the price and as shown later, we have built an asynchronous version of SPEAR on an FPGA.

Chapter 5

Non-Linear Dataflow

In contrast to the previous sections where the data flow control mechanism was illustrated between directly adjacent CAL components, this chapter will focus on circuit structures that exhibit a **non-linear data flow**.
The CAL approach distinguishes sequenced data words by their alternating phase encoding. Data sources have to take this fact into account and alternate the encoding style between $\varphi 0$ and $\varphi 1$ with each new data package issued. By doing so, all CAL components inside the circuit can associate their input signals to a specific context and thus judge consistency. Therefore, in linear circuits no further arrangements have to be performed to ensure that data propagates through the circuit and that it will be processed correctly.
Non-linear circuit structures change this situation completely – the non-linearity causes an interference of the alternating encoded information packages and hence a malfunction or even a deadlock may occur.

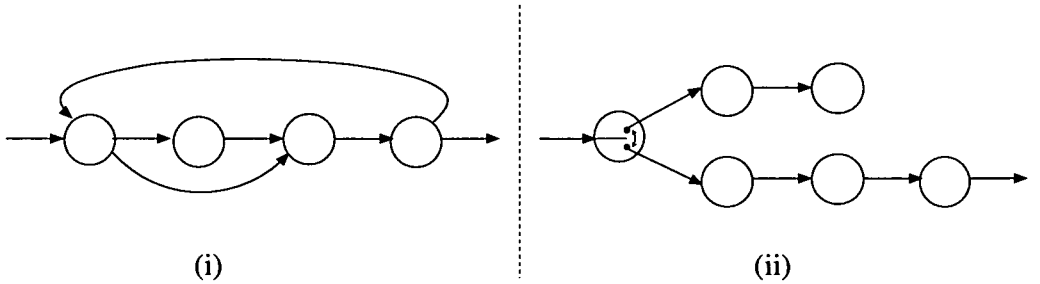


Figure 5.1. (i)Forward and Feedback Path, (ii) Selecting Node

As depicted in Figure 5.1 we distinguish between two types of non-linearity: The first one has its origin in a forward or feedback signal path of the circuit. This means that data is directly fed from the pipe stage where it is generated to the pipe stage where it is consumed (bypassing the pipe register(s) in-between). This may cause an inconsistent input vector and as a consequence a deadlock. This problem can be solved by the selective placement of phase inverters, which is discussed in Section 5.1.

The second type of non-linearity has its roots in selecting nodes: These nodes spread the input data to selected outputs only or require only a selected subset of the input to produce the next output value. As a consequence, nodes connected to non-selected data paths may loose their (phase-)synchronization with the remaining circuit. This problem can be handled either by providing/reading dummy data to/from the unselected data path and generate handshake signals accordingly, or by inserting a so-called synchronizer circuit. This point will be referred to in Section 5.2.

5.1 Avoiding Deadlocks

One of the advantages of asynchronous circuits over their synchronous counterparts is their elastic characteristic. An asynchronous pipeline, for instance, works similar to a FIFO – new data can be issued until the pipeline is full on the one hand and data can be consumed until the pipeline is empty on the other hand. Hence data source and sink are decoupled and the average throughput will be improved. As illustrated in the previous chapter this requires some kind of data flow regulation inside the circuit. If we move away from the simple linear pipeline and consider a more complicated structure, then we have to pay careful attention so that this non-linear structure does not cause a malfunction or a deadlock: A CAL pipeline is based on the assumption that consecutive pipe stages carry alternating phases. If we have a forward path from one pipe stage to another a deadlock may occur. To prevent this procedure, we must put a phase inverter in this forward path. However, this is not imperative, because if a forward path skips an even number of pipe stages no deadlock occurs. Furthermore we have to consider the dynamic behavior of the circuit: In contrast to the synchronous approach, where all pipe registers switch at the same time, latches fire consecutively in an asynchronous pipeline. This yields to short, but intended periods where adjacent latches in a pipeline carry the same phase. This fact has to be considered also when we decide whether a phase inverter has to be placed in a feedback/forward path. In order to get a more concise picture we represented the circuit as a graph. Using this graphical description in the following sections we will analyze the influence and the impact of forward and feedback paths in asynchronous pipelines.

5.1.1 Introduction to Graphs

A graph *G=(N,E)* is defined by the set of *nodes* $N = \{n_1, n_2, \ldots n_n\}$ and by the associated connection (*edges*) $E = \{e_1, e_2, \ldots e_m\}$ between nodes. We distinguish between *undirected graphs* and *directed graphs* or *digraphs* in which the edges are directed. More formally expressed: A digraph is a (usually finite) set of nodes N and a set of *ordered* pairs (a,b) (where a, b are in N) called edges. The node *a* is the *initial node* of the edge and *b* the *terminal node* [15]. Two nodes are *adjacent* if they are connected by an edge. A *weighted graph* associates a value *weight* with every edge in the graph [2].

Under certain consideration weights are used to provide a relation between different edges (or paths). For instance, weights often are associated with the price of a connection in terms of geographical distance. Hence instead of weight the expression *cost*

is used, too. A *path* is a sequence of consecutive edges in a graph and the *length of the path* is the number of edges traversed. A *loop* is a path which ends at the node it begins.

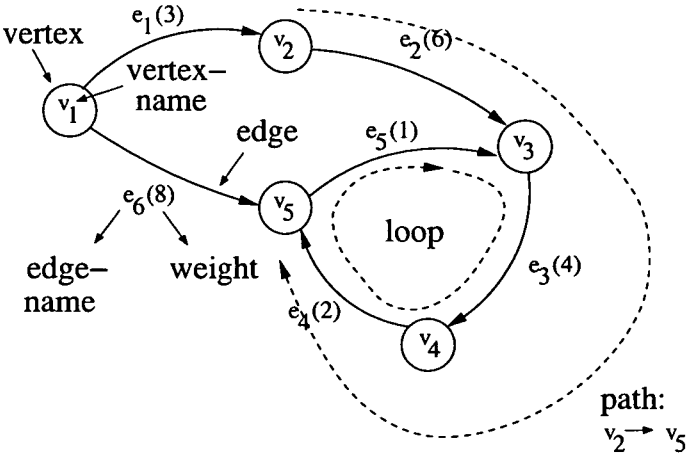


Figure 5.2. Directed Weighted Graph

5.1.2 From the Circuit to the Graph

The first intuitive way to represent a circuit as a graph is to map signal paths to edges and components to vertices or nodes. In digital circuits, however, there are two types of components, namely combinational and sequential ones. With respect to asynchronous circuits, the main difference between these components concerns the need of handshake signals. While sequential components require handshake signals to control the data flow (see chapter 3), combinational circuits do not need any additional signals apart from the input vector to perform their operation, generate new output signals, and hence propagate the information. Due to this fact, combinational circuits are said to be *transparent* with respect to handshake signals. This transparency has also another effect: In the stable state the input vector and output signal carry the same phase. In contrast, sequential components such as pipe registers have data coded in different phases between input and output ports in the stable state. Thus we distinguish between **transparent** (=combinational) and **non-transparent**(=sequential) nodes in our graph representation.

In contrast to conventional CAL-signals (data and control signals such as a *write-enable* signal, for instance), handshake signals are single-rail encoded. As a consequence, they do not carry any phase information and would require a special indication in the graphical representation. Nevertheless, the handshake signals do not provide any additional (useful) information on this abstraction level – they only react to events and can be easily reconstructed for a given event sequence. Therefore, to simplify the presentation we will not draw handshake signals explicitly. However, we have to bear in mind that sequential nodes which are connected directly or through an arbitrary number of combinational (transparent) components share handshake signals

for the data flow regulation purpose.

Memory blocks have to be treated separately from pipe registers and combinational elements, due to the fact that they can operate in two modes: *read mode* and *write mode.*
How should a memory element be modelled in the graphical representation? Basically we have come up with three possibilities:

1. Define both, read and write access, as *transparent* operations.
2. Define both, read and write access, as *non-transparent* operations.
3. Map a memory block to two logical nodes – a *transparent* one for the read access and a *non-transparent* node for the write access.

Due to the fact that a write access consumes the input data, handshake signals are required to signalize the termination of the write operation to the next upstream sequential node. So the first option cannot be taken in account, as transparent nodes do not provide handshake signals.
Thus we have to choose between the second and the third possibility. The second option has the advantage that the graphical representation and the real hardware structure are congruent – we have one memory node in the graphical representation and one memory element in the real hardware implementation. However, this approach does not reflect the real behavior of the circuit: While the write access "consumes" the input vector (which corresponds to the behavior of a non-transparent node), the read access acts similar to a combinational circuit: The *address* can be viewed as an input vector and the related memory content as the result of a transformation of this input vector into the output vector. This corresponds to a (programmable) function unit with an extremely efficient implementation technique and therefore to a transparent node. In fact, in some microprocessors the instruction decoder is replaced by a ROM to reduce the size of the circuit [44], or in many FPGA architectures [1] LUTs[1] are used to implement combinational functions. Modelling a read access as a non-transparent process would give way to a falsified representation of these memories behavior. Though the third option does not yield a direct matching between the physical implementation and the graphical representation, this approach allows the closest mapping from a logical point of view: The splitting of a memory node into a virtual *memory_write node* and a virtual *memory_read node* opens the way to model both types of access in a natural manner. The read access can be performed completely asynchronous – this means that after an arbitrary time the new output is generated in response to a new address – while the write access explicitly signalizes its completion through handshake signals. Figure 5.3 shows how a graphical representation of a circuit can be built following the convention defined in this section.

Latch 1, Latch 2, and *Latch 3* are mapped to the non-transparent nodes *L1, L2,* and *L3,* while the function unit *FU* between *Latch 2* and *Latch 3* is represented by a transparent node. The memory block *MEM* is split into two nodes,

[1]LUT stands for *Look Up Table,* which are small programable memory elements

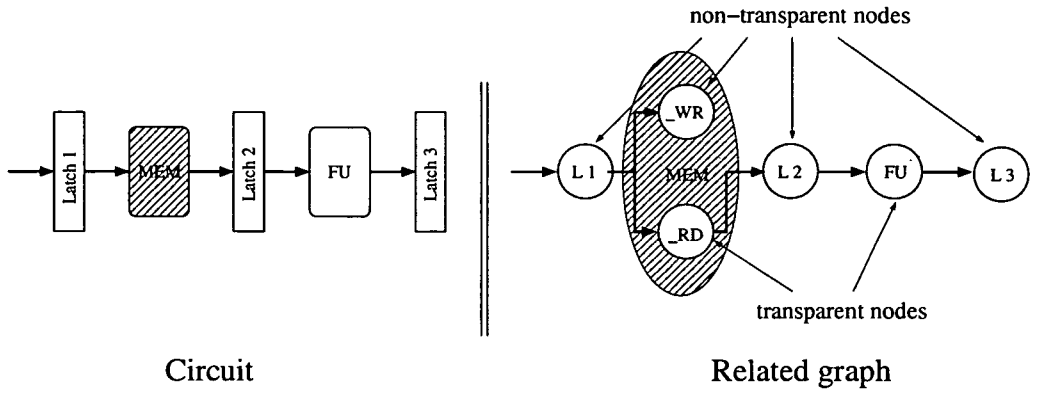
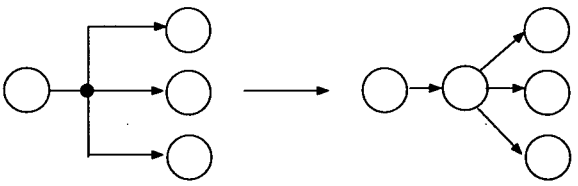


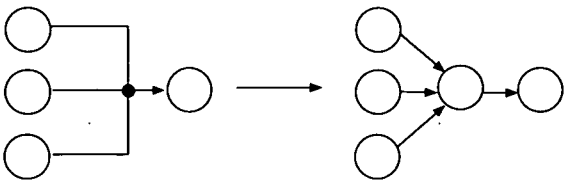
Figure 5.3. Graphical Representation of a Circuit

namely a non-transparent write node *Mem_WR* and a transparent read node *Mem_RD*.

The last circuit structures, which we have to model in order to be able to represent any circuit by a graph are buses. Until now we have considered only point-to-point connections. Nevertheless an output signal can be consumed by more than one component. Such a fork structure can be modelled by inserting a (virtual) transparent node with one input and a suitable number of outgoing edges (see Figure 5.4a). Similarly, merge structures can be modelled by additional (virtual) nodes with n inputs and one output, where n is the number of sources as illustrated in Figure 5.4b.



(a) Multi–destination structure



(b) Multi–source structure

Figure 5.4. Bus Model

By modelling merge and fork structures in above portrayed way the number of in- and outputs of physical nodes are left unchanged, which makes it easier to re-associate the edges of the graph to the signal lines of the circuit.

Furthermore, this approach allows to model even tri-state buses. Tri-state buses may be critical in conjunction with CAL, however: if all sources switch to high impedance,

the bus reaches either an undefined state or its state is fixed by pull-up or pull-down resistors. In any case a deadlock is going to occur. Hence, the designer has to pay attention that the bus is always driven – which questions the benefit of this type of circuits.

In the next section we will identify the position of the phase inverters that have to be added due to non-linear circuit structures – if such an inverter has to be placed in bus structure then we have to consider that placing the inverter at the output of the upstream node causes the phase of all downstream nodes to be inverted, which may not be intended. Thus, a good rule-of-thumb is to place inverters directly on the input side of components and never on their output side. This may not lead to an optimal solution with respect to area-efficiency but it will certainly avoid undesired side effects.

5.1.3 Steady State

In order to operate properly a CAL circuit requires that consecutive pipe stage carry different encoded data. So a pipeline is addressed as "full", if all adjacent stages in a pipeline carry alternating coded data. In contrast we define a pipeline as "empty" if all pipe stages carry data which is encoded in the same phase [104].

Before we can start finding out the position of the phase inverters required to accommodate the feedback/forward paths in the circuit, we have to decide, whether we will consider a full or an empty pipeline. As we will see, the difference between this two configurations has an impact on the result. As a starting point we will assume a full pipeline – the findings of this section will be projected to an empty circuit later on in this chapter.

To determine which phase each node has to carry, we start from the first node in the graph and set its input ports to an arbitrary phase[2]. Then we set the phase of the outgoing edges: If the node in question is a transparent one, then the outgoing edges have to be set to the same phase as the incoming edges, otherwise the phase has to be inverse to the phase of its incoming edges. Subsequently we can go through the output/input edges to the next node. Obviously, at least one input edge of this node is defined as a result of the previous step. This allows us to define the phase of all other incoming and outgoing edges. In this manner we can pass through the entire graph. If two incoming edges of the same node have different phases, then a phase inverter has to be placed.

To illustrate this approach, we will apply it to the graph in Figure 5.5. Let us choose node *L 1* as a starting point – we assign the input edge $\varphi 0$ (see Figure 5.5a) and the output $\varphi 1$, because it is a non-transparent node (see Figure 5.5b). The next node is function unit *FU 1*. Being a transparent node the output edge has to carry the same phase as its input edge. The phase of the latter is already defined by the output of *L 1* and hence the output of *FU 1* is set to $\varphi 1$. (see Figure 5.5c). This implies that the

[2]Due to the fact that the phase itself is not essential – only its alternation between adjacent nodes is decisive – we can start from any node of the graph, although the starting node impacts the result (see Section 6)

output edge of node *L 2* is set to $\varphi 0$, and so on.

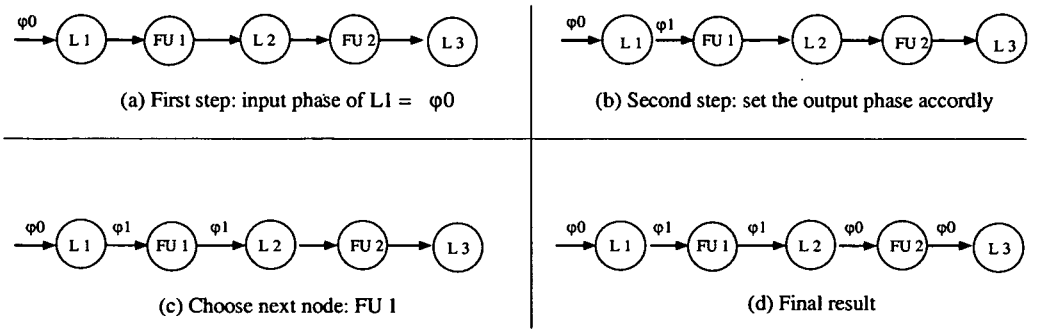


Figure 5.5. Determination of Phases in a Graph

Figure 5.5(d) illustrates the final result. Being a linear circuit no phase inverters are required. Now we will add a forward path, which yields to a nonlinear structure (see Figure 5.6a. Note that the forward path relates the events of *L 3* with the events of *L 1* - from a logical point *L 3* consumes the output of *L 1*. This requires that *L 1* has to consider not only *L 2* but also *L 3* to decide whether it can fire or not. Hence, when adding a forward / feedback path we must not forget to add proper handshake signals (which are not depicted in Figure 5.6b).

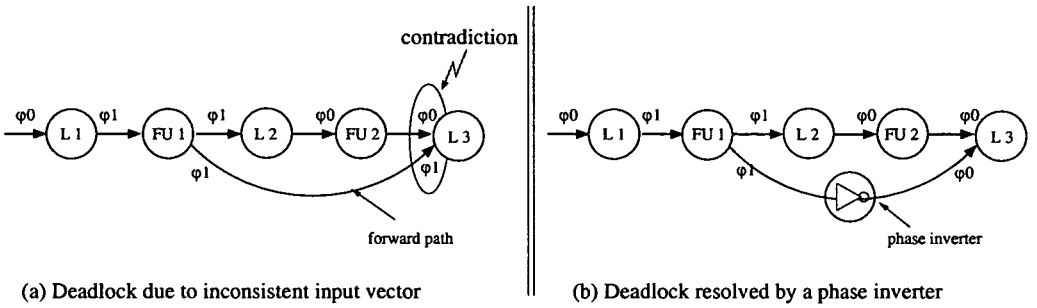


Figure 5.6. Inconsistent Input Vector due to a Forward Path

Due to the fact that latches can carry only consistent output vectors (see previous section) the forward path has to be set to the same phase as the already existing outgoing edge, namely to $\varphi 1$. Now a problem arises at node *L 3* where one input is set to $\varphi 0$, the other to $\varphi 1$ instead, which gives way to an inconsistent input vector for this node. Remember, CAL components wait until the input vector is consistent before a new output is generated. In a full initialized circuit the last node has to fire first to make room for new data at the input side. Therefore the circuit constellation in Figure 5.6a would yield a deadlock: Obviously node *L 3* cannot take over new data due to its inconsistent input vector and this procedure also blocks node *L 2*: *L 3* signalizes – through the handshake signals – that the data conveyed onto the output of node *L 2* and (transformed by *FU 1*) has not been consumed yet. Hence, *L 2* cannot fire in order to ensure that no data get lost in the circuit. The same is true for node *L 1*. As

a consequence no progress can occur and the circuit will be stalled regardless of the behavior of its environment.
The deadlock can be resolved by placing a phase inverter in the forward path as illustrated in Figure 5.6b. The phase inverter acts similar to a latch – it ensures that the phase on incoming edges and outgoing edges differs. This yields to a consistent input vector for node *L 3*, and therefore new data can be taken over. At this point it is important to highlight the fact that a phase inverter does not affect only data signals but also the related handshake signals: As described in Section 3 the *capture_done* signal of a latch indicates by its **level** the phase of the data which was consumed last. As a logical consequence, if we add a phase inverter in the data path between two latches we also have to negate the corresponding *capture_done* signal.

The abovementioned deadlock is easy to recognize since the circuit will not start working after reset. However, a missing phase inverter can cause more malicious malfunction which can be extremely difficult to identify. In Figure 5.7 the forward path is connected to the *FU 2* node, instead of *L 3*. As in the previous example we assume that the output of the function unit 2 *FU 2* carries $\varphi 0$. We recognize an inconsistent input vector, but in this example a transparent node is affected.

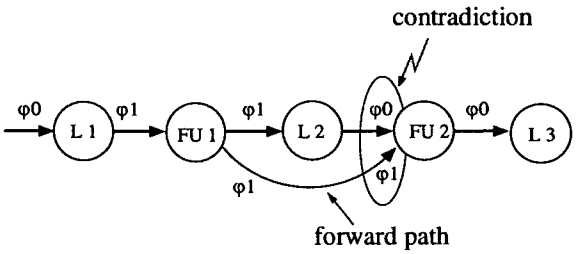


Figure 5.7. Forward Path to a Transparent Node

Due to the fact that a transparent node does not participate in the handshake procedure, the inconsistent input has a completely different impact – one will be able to view that in this circuit no deadlock occurs.

Figure 5.8 illustrates what happens: In 5.8(1) the starting point of the circuit is shown. As expected node *FU 2* has an inconsistent input vector. The input vector of node *L 3* , however, is consistent and being a sink *L 3* can consume its input as illustrated in 5.8(2). The $\varphi 0$ at the "output" side of the node indicates that data coded in $\varphi 0$ was consumed and that the next data to be used must be coded in $\varphi 1$.

Node *L 3* communicates over the handshake channels that has consumed its input data and hence it is allowed for node *L 2* to change its output 5.8(3). As soon as *L 2* has taken over new data, the input vector of *FU 2* becomes consistent, a new output can be generated and *L 1* takes over its input data 5.8(4). Immediately after that *L 1* fires due to the fact that its output data has already been consumed by *L 2* and *L 3*. The new output generated by *FU 2* enables *L 3* to fire – *FU 1* produces a new output (see Figure 5.8(5)) because *L 1* has produced a new input vector.

Note that the temporal relation between the events on *FU 1* and *L 3* does not make any difference due to the fact the *FU 2* keeps its old output as long as an inconsistent

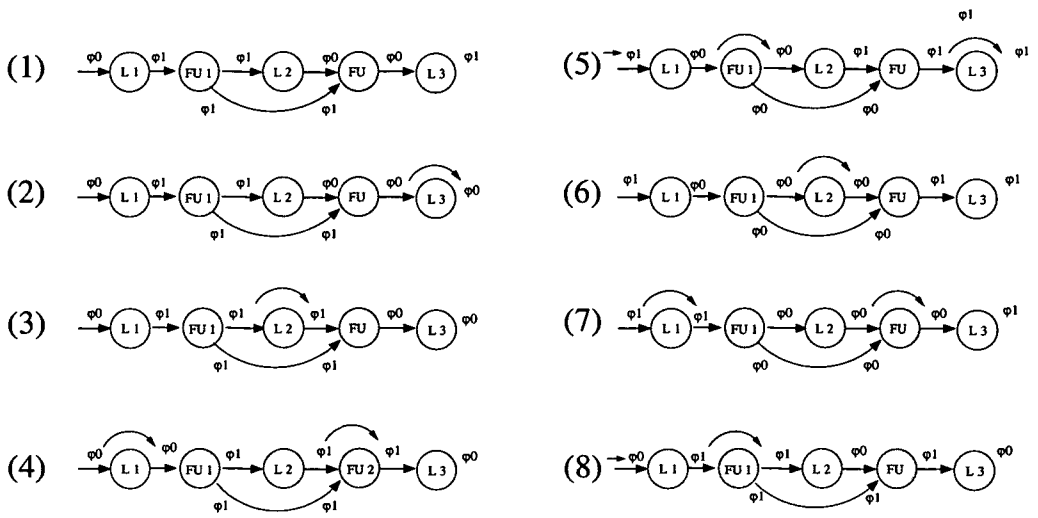


Figure 5.8. Progress of a Circuit

input vector is applied. In 5.8(6) we have the same situation as in 5.8(3) but with an inverted phase constellation. Hence node *L 2* consumes its input vector, which triggers node *L 1* on its part. In figure (8) we reach the same configuration as in the starting point. Observe that even having an inconsistent input vector no deadlock has taken place.
What went wrong or did actually anything go wrong? If we take a look at 5.8(3) and 5.8(4) we recognize the bug: *FU 2* starts its execution after *L 2* has taken over $\varphi 1$. The point is that *FU 2* was probably not intended to combine the "old" data from *FU 1* with "new" data from *L 2*. Instead *FU 2* joints the new produced output of *L 2* (coded in $\varphi 1$) with the output of *FU 1*. There are two possibilities: first the additional path is a "regular" forward path or this path is used for another purpose.

Regular forward path: If the forward path is a regular forward path, i.e. data conveyed by the forward path and data conveyed by the "regular" path (in the next iteration) is identical and *FU 2* has to select between these signals (in accordance with some control signals). Then the circuit will operate correctly. However, in this case the forward path would not be not required, since the regulation of the data flow is inherent in asynchronous circuits. Thus, if we have to transform a synchronous circuit, which has such a forward path inside, into an asynchronous CAL circuit, then we can perform a straight forward mapping of all signal paths. By doing so we are sure that we have not forgotten any useful signal path and that the "dead" forward path does not influence the correct functionality of the circuit. Or does it?
The missing phase inverter affects the timing of the circuit – it transforms the originally delay-insensitive circuit to a speed independent one: In the previous explanation a hidden timing assumption was made: In Figure 5.8(3) *L 2* takes over data coded in $\varphi 1$ and this gives way to a consistent input vector for *FU 2*. Now, we will assume that the connection between *L 2* and *FU 2* is subject to a large delay. While the data is

propagating from the output of *L 2* to the input of *FU 2*, *L 1* can take over new data (coded in $\varphi 0$) and *FU 1* can produce a new output in its part. This data, coded in $\varphi 0$ will be transmitted to *FU 2*. If this action proceeds faster than the propagation of the output of *L 2* to *FU 2* then a deadlock occurs.

Arbitrary data path: Contrariwise if the edge between *FU 1* and *FU 2* is not used for forwarding purposes, then a malfunction will occur without exception: As illustrated in Figure 5.9 the output of *FU 1* and *L 2* should be added. Furthermore we assume that the output of *L 2* carries "6", the forwarded output of *FU 1* "3" and the "normal" output of *FU 2* "2". This circuit will add the values carried by the outputs of *FU 1*, namely *2* and *3*, instead of *3* and *6*. The problem is that *L 2* must fire before the adder can perform its operation.

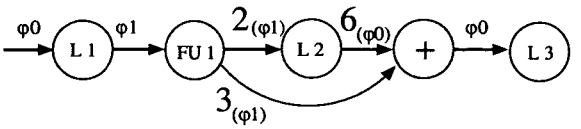


Figure 5.9. Arbitrary Data Path

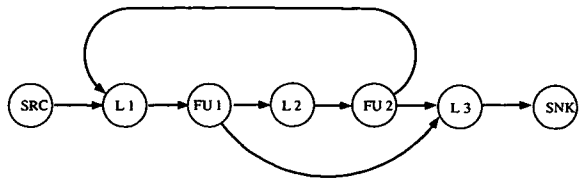
This simple example shows the complexity of forward paths in asynchronous CAL designs. On the one hand a forward path may cause a deadlock – this is relatively easy to find out and on the other hand the circuit might operate as well, but produce unintended results. Such errors are extremely difficult to seek out because they have their origin in the dynamic behavior of the circuit.

5.1.4 Dynamic Behavior

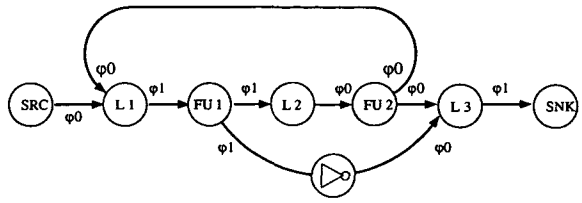
In this section we will analyze the dynamic behavior of the circuit illustrated in Figure 5.10a. In contrast to the previous example, this circuit is highly non-linear, due to the fact that it contains not only a forward but also a feedback path. If we want to initialize the circuit with alternating phases we may recognize that an additional phase inverter has to be put on the forward edge (see Figure 5.10b).

In Figure 5.10b the circuit contains only nodes which have consistent input vectors. Now we will analyze what happens when the circuit starts to operate. For this purpose we assume that the data sink node *SNK* consumes data without delay. Similarly, the source node *SRC* produces data immediately after *L 1* has demanded for it. The first node, which becomes active, is *SNK* - its input vector is consistent and therefore – as postulated above – it will consume its input data immediately (see Figure 5.11(i)). This consumption is signalized through the handshake signals to *L 3*. This node has also consistent input edges and thus it will take over its input data enabling *L 2* to fire (see Figure 5.11(ii)). Also *L 2* has consistent inputs and hence can in turn take over its input data as illustrated in Figure 5.11(iii).

As depicted in Figure 5.11(iv) The output of *L 2* causes that *FU 2* gets new input data and toggles its output to $\varphi 1$. As a consequence, the input vector of *L 1* becomes



(a) Graph example with forward and feedback path



(b) Graph with phase inverter

Figure 5.10. Highly Non-Linear Circuit Example

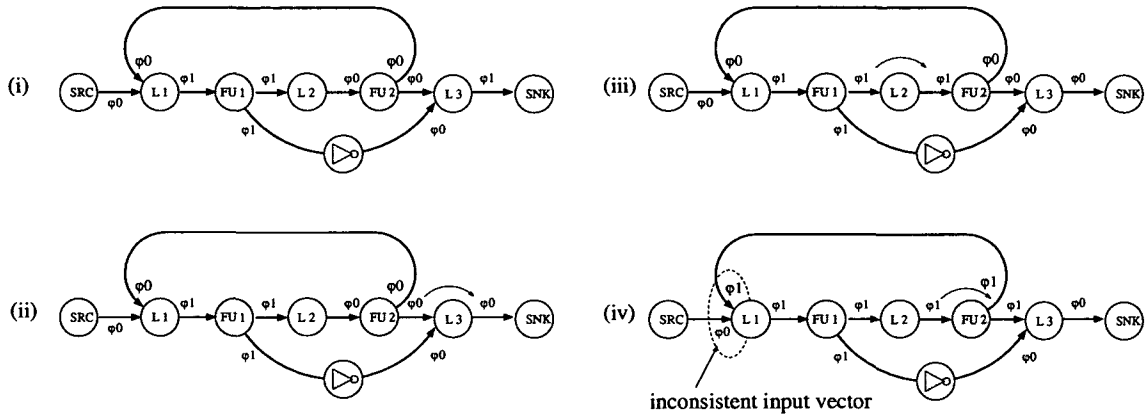


Figure 5.11. Sequence of Transitions

inconsistent. This obviously prevents *L 1* to consume its input data and a deadlock occurs. However this deadlock is not mandatory, quite on the contrary: At the moment where *L 2* has taken over the new data, *L 1* receives the information (over the handshake signals) that its output data was consumed and therefore new data can be accepted. If the handshake signals are faster than the delay of the signals from the output of *L 2* through the function unit *FU 2* to the input of *L 1* then the data can be consumed by *L 2* before its input becomes inconsistent.

Here we have the same situation as in the previous section with respect to the forward path: The data, which was accepted by *L 1* is not the data that the system designer intends to be consumed. The logical sequence of events implies that *L 1* is activated after *L 2* has taken over data and all inputs of *L 1* are in a stable state. If the feedback path goes through more than one non-transparent node then this circumstance can be viewed more clearly. Anyhow, this circuit is no more delay insensitive (on the gate

level) but speed independent.
Is it impossible to build delay-insensitive nonlinear pipelines? What is the origin of the problem? The clue is that the nodes in the circuit do not operate in a lock step manner – they do not switch concurrently, but in a sequential way. Figure 5.12 illustrates this aspect in a more apparent way.

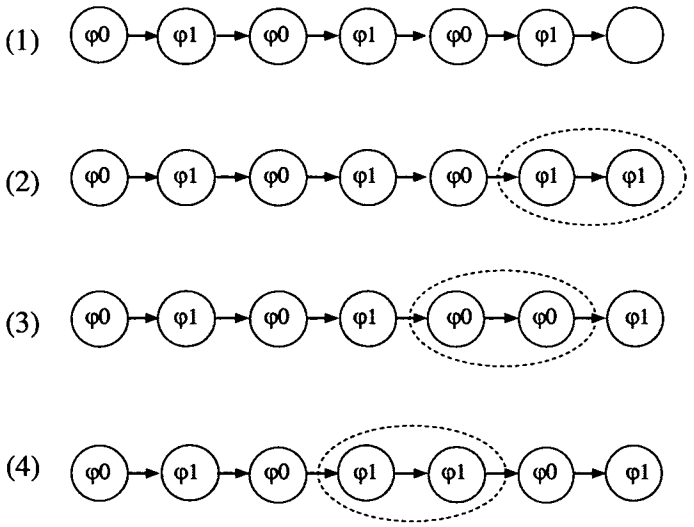


Figure 5.12. Abstract Switch Sequence

The phase indication inside the squares should indicate the phase carried by the output vector of the node. To get a more illustrative picture, we assume that all nodes are non-transparent. As already mentioned in the introduction of this chapter, we assume a full initialized pipeline (see Figure 5.12(1)). Due to this fact only the data sink can take over data – it is the last node in the chain. For a short instant the last two nodes keep the same phase (see Figure 5.12(2)): the last but one node cannot consume its input data, before the last node has acknowledged the consumption on its part. For the time, which is required to get and to process this acknowledged signal, both nodes carry the same phase on their output.
In the next step the last but one node takes over data on its part, thus reestablishing the alternating order of the phases on the circuit part on its output side. On its input side however, we have the same situation as previously, namely two adjacent non-transparent nodes which keep the same phase. As before this is only an intermediate state and when the previous node switchers, the alternating order is reestablished[3] (see Figure 5.12(3)). We may see that if a circuit makes progress such duplicated states will takes place. These intermediate states are called *bubbles* [104]. Figure 5.13 compares the circuit in state (1) and in state (4). The node which holds the bubble is marked. This node constitutes the border between the part of the circuit, which has still the

[3]Watchful readers would object, because in the meanwhile the last node in the chain is also enabled to fire again. In order to be able to concentrate this explanation on the actual point this aspect will be ignored.

same phase configuration as in the initiatory state and the part where nodes have already switched. Taking a deeper look we recognize the phases have moved one step in the sink direction and the resulting gap is filled by the bubble.

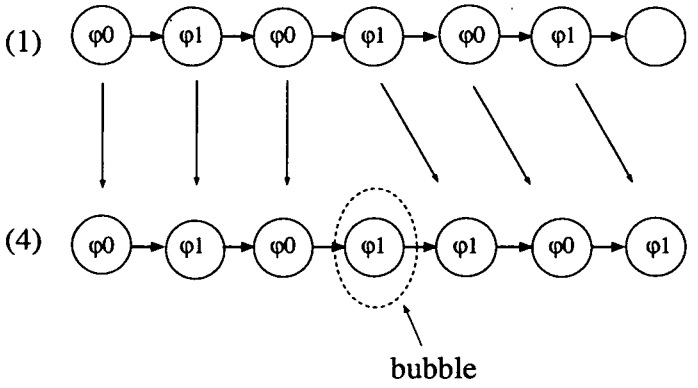


Figure 5.13. Impact of Switching Activities

What is the impact of feedback paths? In Figure 5.14 we added such an exemplary feedback path to the circuit illustrated in Figure 5.11. The source node of the feedback path carries $\varphi 1$ and the destination node requires the same phase on its input. Hence based on this static analysis no additional inverter is required for this edge.

Due to the fact that the circuit in Figure 5.14 switches in the same order, we can transfer the state information in Figure 5.11(4) directly to this circuit. Now, we recognize the problem: We determined the phase inverter, based on a steady state of the circuit. Being a full initialized circuit the last node switches first, then the last but one node and so on. Thus the feedback path carries information from a node which appears ahead of time in the logical event sequence to a node which will be activated later. As a consequence the initial state of the source node of the feedback path changes before the destination node has fired and a deadlock occurs as illustrated in Figure 5.14(4).

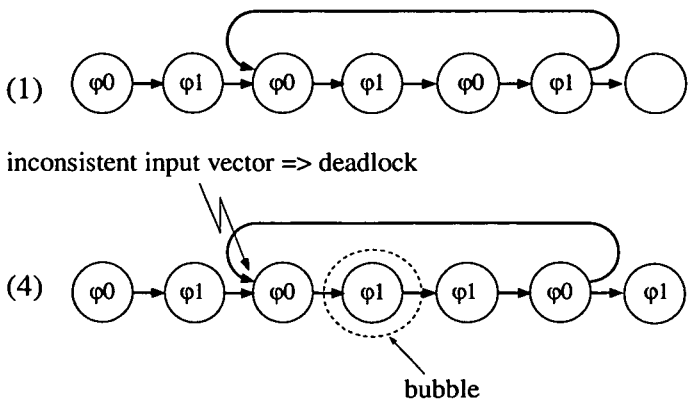


Figure 5.14. Switch Sequence with Feedback Path

The reader can easily reconstruct that a forward path is not affected by this mechanism due to the fact that the node, which receives the forwarded data is in the event

sequence prior to the source node of the forward edge.

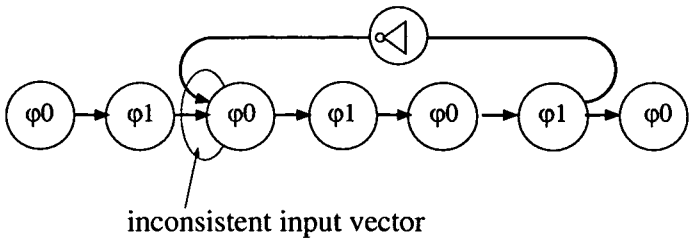


Figure 5.15. Final Circuit Constellation

Hence to solve this problem we have to identify all feedback paths in the circuit and invert their phase (again). Nevertheless, this violates the consistency requirement in the steady state as illustrated in Figure 5.15. In a running circuit, this has no additional effect, because nodes which have inconsistent inputs keep their old output value until the input data becomes consistent. In the startup phase, however, this may represent a problem for combinational nodes: Due to the fact that transparent (=conbinational) nodes are not initialized by the reset, there is no defined "previous" state that they can keep in case of inconsistent inputs. Therefore the inconsistent input yield to an **undefined** output. Anyhow, in the physical implementation this undefined state does not affect the proper operation of the circuit: On the one hand the sequential nodes that enclose combinational gates are initialized correctly and hence consume data only when it is consistent and coded in the expected phase. On the other hand, the circuit has to be initialized in a way, that enables it to start up, and hence an inconsistent input vector will become consistent sooner or later.
However, this circumstance causes problems to most of the available commercial simulation tools: Being originally intended for a use with synchronous circuits, (where such undefined signals denote a real mistake) conventional simulation tools propagate undefined states through the whole circuit making a simulation impossible. If we want to use these tools for asynchronous circuit design then we have to take this fact in account: Either we feed the reset signal to all combinational circuits, i.e. to the RS-latches embedded in all AND, OR gates, or we disable the additional inverters in the feedback path during the reset phase (for simulation only). The latter is seems to be a more reasonable solution.

Note, it is not required that the phase inverters are placed directly in the feedback path: From a logical point of view a feedback signal forms a loop in the circuit – hence any edge of the loop can be inverted to achieve the desired effect. However, the original initialization must not be changed due to the additional inverter. At this point it is essential to highlight that the proposed solution only works when the pipe is full initialized. In the next sections we will generalize these results.

5.1.5 Structural Regulation of the Data-flow

In the previous section we have built a correctly operating circuit by adding an inverter into the feedback path. This inverter was necessary to account for the fact that the phase of each node moved downstream one step (in the direction of the data sink). Without the additional inverter this would cause an inconsistent input vector on the node that receives the feedback signal but has not fired yet. However, the node that generates the feedback signal could fire again and hence re-produce the inconsistent input vector at the node which receives this signal. Is there something which prevents this node to switch again? The answer is yes, because non-linear structures inside a circuit cause a **structural regulation of the data flow**. As illustrated in the previous chapter handshake signals between latches steer the data flow, but only locally i.e. between two adjacent nodes. The result is an elastic pipeline: A data source can issue data until all pipe stages are full, regardless of the data sinks behavior and vice versa – the data sink can consume data as long as the pipeline is not empty disregard of the data sources behavior. The local data flow regulation between pairs of pipe registers ensures that incoming data propagates as far as possible into the pipeline and guarantees that, if data is consumed at end of the pipeline, the remaining data will moved downstream. A feedback path (the same is true for forward paths) inside such a circuit monopolizes this mechanism: The existence of a forward or a feedback signal implies that at least one signal is generated in one pipe stage and consumed in another one. This creates a concatenation of these pipe stages (which are not adjacent) and thus restricts the elasticity of the entire pipe: To ensure that no data is getting lost when the first pipe switches, the concatenated one has to switch on its part, before the first one switches again. To illustrate this more demonstratively, Figure 5.16 zooms out the environment of the starting and the end point of the feedback edge from a logical point of view.

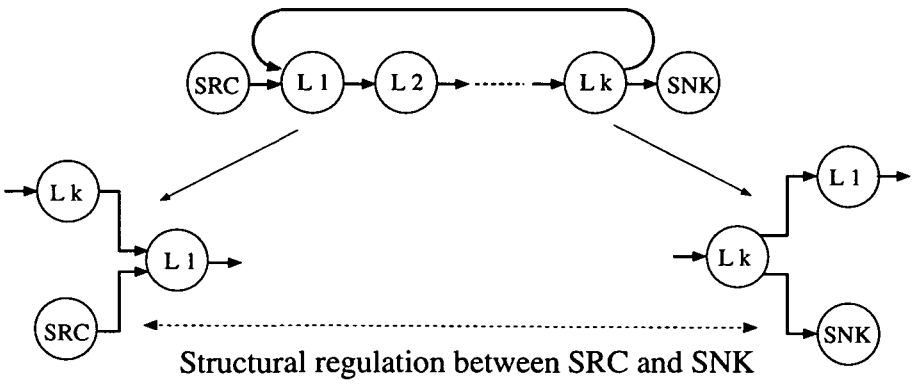


Figure 5.16. Structural Regulation

We may view that the node *L 1* has two inputs, one from *SRC* and the other one from *L k*. Hence *L 1* can fire only when both nodes has produced valid data. On the other hand *L k* acts a source for *SNK* **and** for *L 1*. This in turn requires that *L k* can fire only when both nodes have consumed their output. The consequence of this

concatenation is that the circuit delimited by *L 1* and *L k* is no longer elastic: A new data package can be issued only when *L k* has consumed a data package aforementioned – *L k* can fire again only if a new data package was consumed by *L 1* in the meantime.

This can be seen as a vital breakthrough, if we go back to the original problem, namely identify the additional inverters in the feedback path: Assuming a full initialized pipeline, we know that after *L k* has fired all nodes in the backward direction – this means *L k-1*, *L k-2*, ... *L 2* fire exactly one time before *L 1* can become active. This finally enables *L k* to fire again and the procedure starts once more. Due to the fact that this behavior is deterministic and guaranteed, all feedback paths must cause a violation of the alternating sequence of phases during the reset phase.

5.1.6 Empty Initialized Pipeline

Why have we stressed the fact up till now that the pipeline must be full? In order to find out the impact of pipeline initialization, let us consider, what happens if we initialize all latches with the same phase - which leads to an empty pipeline.

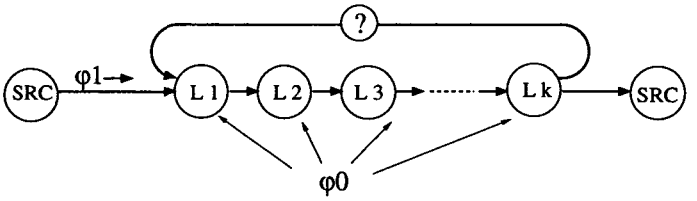


Figure 5.17. Empty Initialized Pipeline with Feedback Path

At first we will analyze the impact of the feedback paths. For this purpose we consider the circuit in Figure 5.17. Due to the fact that the pipeline is empty, neither SNK nor the nodes inside the circuit can fire. Therefore SRC has to issue data encoded in $\varphi 1$ – this would lead to an inconsistent input vector if no phase inverter is place in the feedback path. This circumstance can be generalized: **In an empty initialized pipeline we have to place a phase inverter on all feedback paths.**

Now, we will consider what happens with the issued data: As depicted in Figure 5.18(1) *SRC* provides new data, which is consumed by*L1* (see Figure 5.18(2)). Subsequently *L 2* takes over the issued data on its part as depicted in Figure 5.18(3). Although *SRC* provides the next data package *L 1* cannot fire again even due to the fact that *L k* has not fired yet. In Figure 5.18(k) *L k* fires and only now *L 1* is enabled to consume its input data (see Figure 5.18(k+1)).

We recognize a completely different switch sequence of the circuit compared to the full initialized pipeline: In the latter we always have (apart from the bubble) alternating phases on the output of adjacent nodes – in the empty initialized pipeline (with feedback path) all nodes must carry the same phase before the new data can be issued.

This also affects the placement of the phase inverters in the forward path: To find out, if phase inverters have to be placed on forward paths we consider Figure 5.18 again and add a forward path to the circuit as illustrated in Figure 5.19.

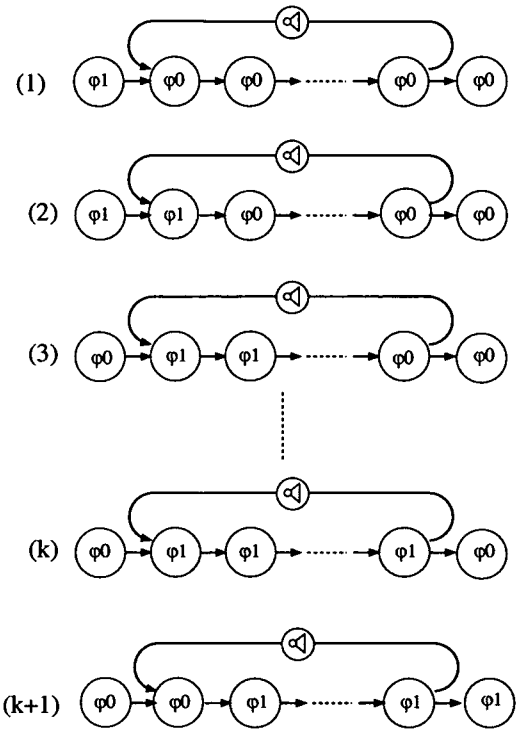


Figure 5.18. Event Sequence of an Empty Initialized Pipeline

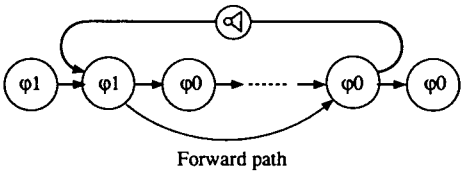


Figure 5.19. Empty Pipeline with Forward Path

We see that destination node of the feedback path becomes an inconsistent input vector, when *L 1* has fired as illustrated in Figure 5.19. However, this inconsistency will be resolved, at the moment, when the input data wave propagates through the circuit and arrives at node *L k*. Hence, **in an empty initialized pipeline no phase inverter has to be placed on forward path.**

We recognize that full and empty initialized non-linear pipelines do not only have a completely different event sequence, but they even required a different phase inverter setup in order to avoid deadlocks. In the next section we will analyze the impact of the initialization on the performance.

5.1.7 Relation Between Performance and Initialization

One consequence of the structural regulation of the data flow is that the initialization influences the throughput of the circuit. In order to show this, we will consider in the

first step only non-transparent nodes – the impact of transparent nodes will be modelled later on in this section. As in the previous sections we assume that data source and sink are faster than the pipeline circuit and thus do not constitute the bottleneck. We recognize in Figure 5.20 that the empty and the full initialized circuits are not identical - the additional inverter in the feedback path should indicate that depending on the chosen initialization different inverter configurations may be required.

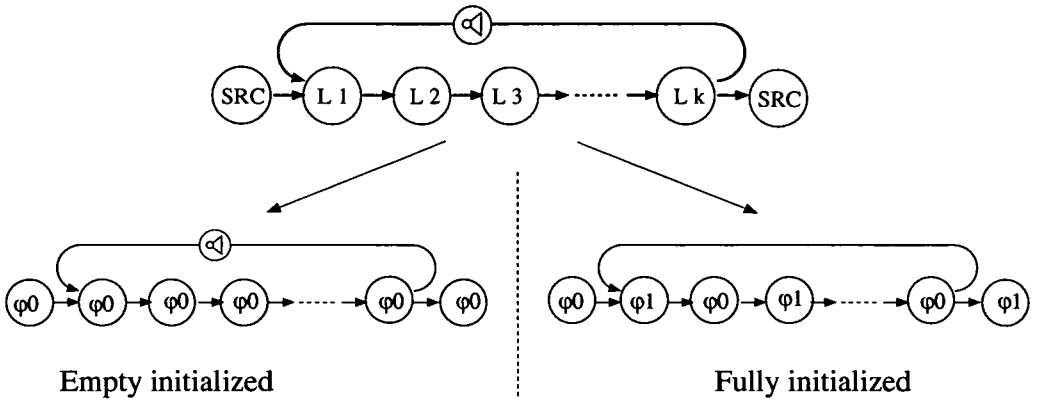


Figure 5.20. Full and Empty Initialized Circuit

In the first we will consider the empty initialized circuit. The first node, which fires, is *L 1*. As explained in the previous section *L 1* can fire again, after the first input data wave was spread through the entire circuit and consumed by *L k*.

Therefore the achievable throughput Θ can be formulated as:

$$\Theta_{Empty} = \frac{1}{\sum_{i=1}^{k} \Delta SW(i)} \tag{5.1}$$

where k is the number of edges in-between the start and the end node of the feedback edge and $\Delta SW(i)$ is the time which node $L(i)$ requires to switch.

Now, hoping to get a better result we will perform the same analysis for the full initialized pipeline. Due to the fact that the last node of the circuit fires first, we will consider the time between the consumption of consecutive data packages. While this is equivalent to the period where new data can be issued, its calculation is more illustrative. Hence *L k* fires first and changes the phase of the feedback path, which forms the input data for *L 1*. According to our assumption that the data source produces new data immediately, its output must be already set to the next phase, such that the input vector of *L 1* becomes valid as soon as the feedback from*L k* arrives. However, *L 1* cannot fire, because its output has not been consumed by *L 2* yet. The latter cannot fire, due to the fact that its output has not been taken over by *L 3* yet, and so on. We recognize that all pipe registers must fire before *L 1* can fire on its part. This in turn enables *L k* and initiates the described event sequence again. One can view from this procedure that the throughput Θ of the full initialized pipeline is defined by:

$$\Theta_{Full} = \frac{1}{\sum_{i=1}^{k} \Delta SW(i)} \tag{5.2}$$

where k is the number of edges in-between the start and the end node of the feedback edge and $\Delta SW(i)$ is the time which node L *(i)* requires to fire.

This is a notable result - even being full initialized, all nodes inside the feedback path must fire on time, before a new input can be issued and respectively a new output can be generated. We see that there is no difference between the full and the empty initialized pipeline, if we do not consider the delay of function units in-between the pipe stages.

Impact of function units on performance: Now we will consider the impact of function units between pipe registers. Do they have the same effect on the full and the empty initialized circuits, namely to slow down the progress, or is there a difference? To investigate this we consider the circuit in Figure 5.21:

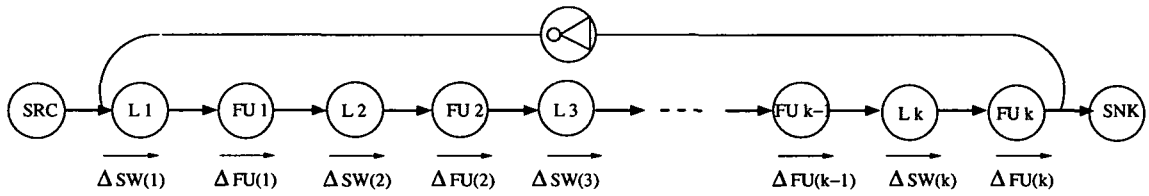


Figure 5.21. Pipeline with Function Units

We assume that the circuit is in the steady state. As previously we will analyze how much time is required until new data can be issued or consumed respectively: The throughput Θ of an empty initialized circuit can be calculated as follows:

$$\Theta_{Empty\ with\ FU} = \frac{1}{\sum_{i=1}^{k} (\Delta SW(i) + \Delta FU(i))} \tag{5.3}$$

The throughput is defined by the sum of delays of the pipe registers and the function units. This is a comprehensible result due to the fact that a data package has to pass through all components before the next one can be issued. Note, that the objective of pipe registers is to divide the circuit in sub-circuits, which operate concurrently. In an empty initialized pipeline, however, the pipe stages do not process different data waves concurrently, instead the same data wave is processed by all pipe stages in a serial manner. This stands as a contradiction to the basic principle of pipelining, since the circuit would operate faster without any pipe register inside.

Now we will consider the full initialized circuit: We assume again that the circuit is in the steady state: The last node fires first due to the fact that – as postulated previously – the data sink consumes immediately its output after becoming valid. Subsequently to it the last but one (non-transparent) node fires. In contrast to the empty pipeline, where non-transparent nodes must wait until the function units located previously to them have generated the next (valid) input data, in the full initialized pipeline this is not always true. Due to the fact that the firing sequence runs opposite to the

propagation of the data waves the output, which is generated by the function unit, is not consumed within the next firing event as illustrated in Figure 5.22.

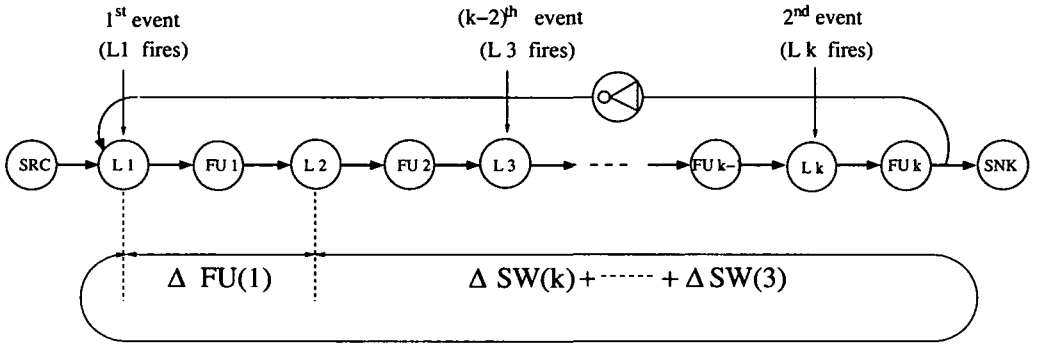


Figure 5.22. Impact of Function Units to a Full Initialized Pipeline

We recognize that the output of *FU 1* will be consumed only after whether (k-2) nodes have fired, as illustrated in Figure 5.22. Hence, we have to distinguish, if the function unit is able to generate its output in time, such that the subsequent node can consume it immediately, or not.
In the first case, we assume that the delay of the function unit *FU 1* dominates – after *FU 1* has finished its operation, the next downstream node, *L 2*, can fire and complete the "firing cycle"by enabling *L 1* again. The throughput Θ can more formally be expressed as follows:

$$\Theta_{Full\ with\ slow\ FU} = \frac{1}{\max_{1\leq i\leq k}(\Delta SW(i) + \Delta SW((i+1)\ mod\ k) + (\Delta FU(i))} \quad (5.4)$$

when $\sum_{i=1}^{k} \Delta SW(i) - (\Delta SW(j) + \Delta SW((j+1)\ mod\ k) \leq \max_{1\leq i\leq k} \Delta FU(j)$.
In contrast, if the function unit *FU 1* generates its outputs fast enough so that the subsequent node *L 2* can fire immediately, then the throughput is described as follows

$$\Theta_{Full\ with\ fast\ FU} = \frac{1}{\sum_{i=1}^{k} \Delta SW(i)} \quad (5.5)$$

when $\sum_{i=0}^{k} \Delta SW(i) - \Delta SW(j) + \Delta SW((j+1)\ mod\ k) > \max_{0\leq i\leq k} \Delta FU(j)$.
We recognize that in this case the propagation delay of the function units between pipe stages has no impact on the resulting throughput. However, compared to the empty initialized pipeline the full initialized one yields always higher throughput.

5.1.8 Nested Feedbacks/Forwards Path

Until now we have considered only one single feedback path – in real circuit implementations nested loops may also occur. Nevertheless, this does not constitute a problem – a sub-circuit, which incorporates such a nested loop, can be abstracted to

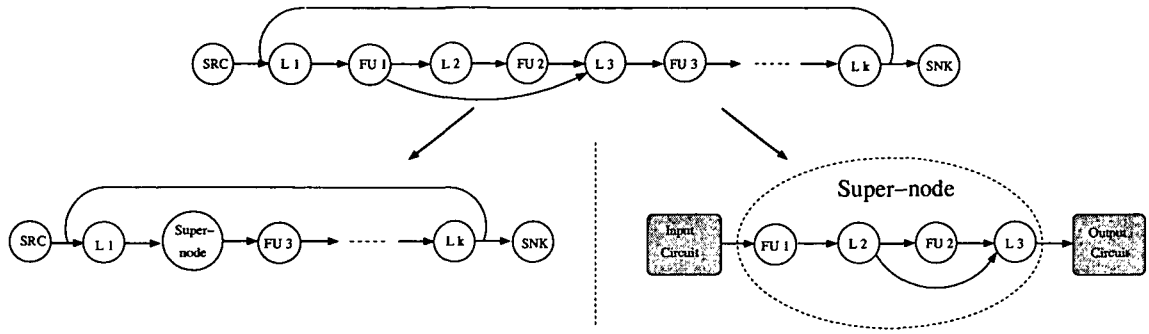


Figure 5.23. Nested Feedback Path

a super-node. Now we can place the inverters, handling the super-node such as all normal nodes (see Figure 5.23).

Inside the super-node we can apply the same algorithm to place the inverters, where the nodes outside the super node are considered as data source and data sink respectively. In this way we can recursively progress, until we reach the inner most loop. Note, it is possible to choose different initialization types for the super node and the global circuit – this has only an influence on the overall performance. Notice, that the super-node has to be modelled as transparent or non-transparent one, depending on the number and type of nodes which it incorporates.

5.1.9 Algorithm for Placing Phase Inverters

In Section 5.1.7 we have portrayed that a full initialized non-linear pipeline is preferable to an empty initialized one due to the fact that it permits higher performance. In contrast to the empty pipeline, where the phase inverter can be placed in a straight forward manner, in a full initialized circuit different aspects such as the number of non-transparent nodes, which are skipped by a nonlinear signal path as well as the dynamical behavior of the circuit, have to be considered. As a consequence, identifying the position of phase inverters becomes a too complex and error prone task. Hence, we incorporated the rule for the appropriate phase inverter placement, derived from the specific examples above, into a software algorithm: As illustrated in Figure 5.24 the placement is performed in two steps.

In the first step the steady state is considered: An arbitrary node is selected and the phases on its in- and output edges are set. Using this node as a starting point, the algorithm goes through the graph node by node considering that: (a) in- and output edges of transparent nodes have to be set to the same phase and (b) the phases of in- and outputs of non-transparent nodes have to differ. If a conflict with respect to the phase of an edge occurs, then a phase inverter has to be placed. For the **final** circuit the non-transparent nodes have to be initialized as assumed in this step.

In the second step the dynamic behavior must be considered: This requires that we place phase inverters on all feedback paths. How does the algorithm identify a feedback path?

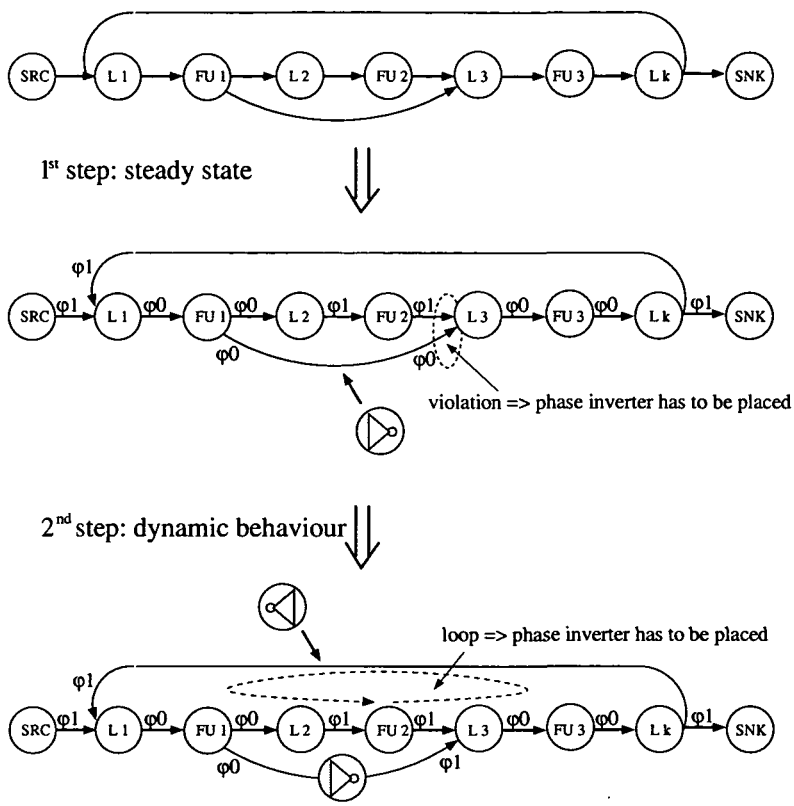


Figure 5.24. Placement of Phase Inverter

From a logical point of view, a feedback path builds a loop in the graph. Hence the algorithm identifies all loops and places the inverter on one edge of the loop. From a logical point of view, it does not matter what edge of the loop is inverted. In practice the placement of the inverter affects the start up sequence of the nodes involved by the loop: The non-transparent node arranged beforehand to this phase inverter will fire first after the reset signal is deactivated. Note that a phase inverter can be shared by several loops. This can be used for optimization purpose, but if the loops are treated independently one from each other, then some loops in the final circuit may contain (unintentionally) more than one inverter. The algorithm prevents this by considering all loops simultaneously during the phase inverter placement.

As depicted in Section 6 the result of the algorithm depends strongly on the selected starting node. Due to this fact the starting node can be selected manually and provided as a parameter to the program which implements the algorithm. A detailed description of the algorithm can be found in [91].

5.1.10 Practical Results

To substantiate the results elaborated in this section we made some simulations: For this purpose we built a simple pipeline with six pipe stages, where data source (SRC) and sink (SNK) are emulated by the testbench.

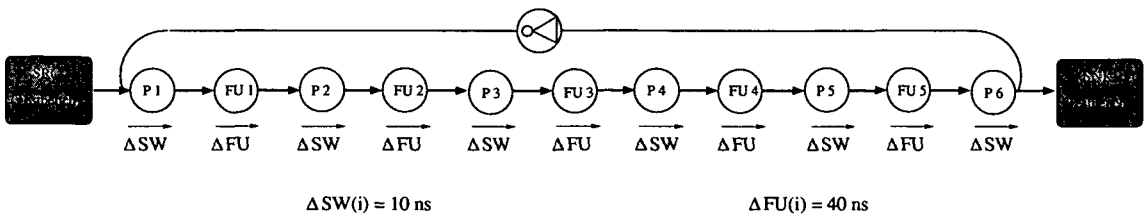


Figure 5.25. Pipeline which is Used for Simulation

We set the switch delay of all latches to 10 ns and delay of all function units to 40 ns to get more comprehensible simulation results.
To show the elasticity of linear structures in asynchronous designs, first we simulate the pipeline without a feedback.

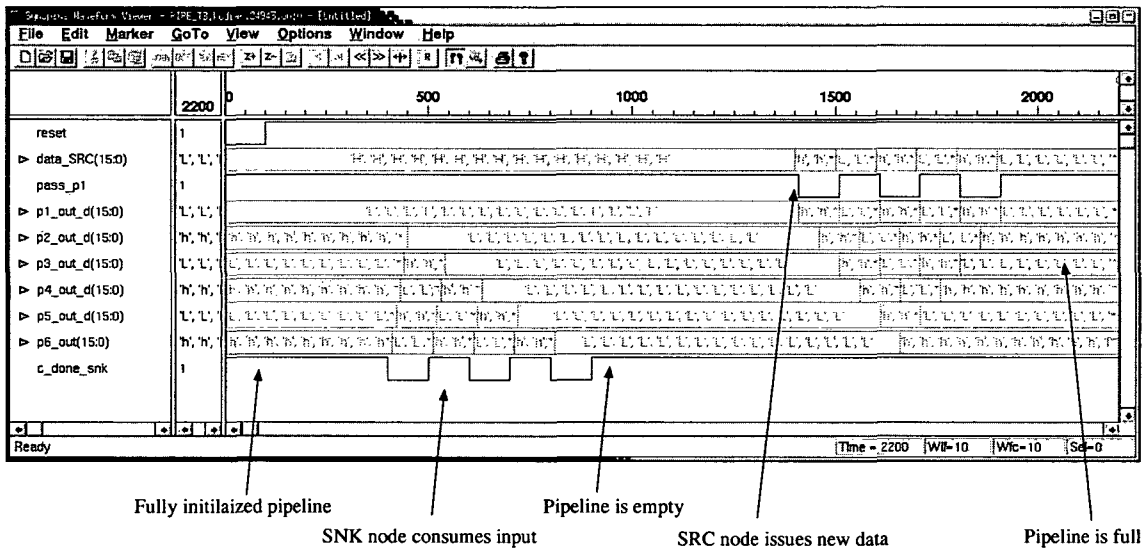


Figure 5.26. Simulation of a Linear Pipeline

As illustrated in Figure 5.26 we initialized the pipeline in such a way that all adjacent latches carry alternating phases on their outputs. We wrote the testbench in such a way that SNK becomes active first. SNK can consume all data waves inside the pipeline, regardless of the behavior of SRC – this leads to an empty pipeline. Afterwards we activated SRC – like SNK the data source can issue data independently until the pipeline is full again.

As mentioned in the previous section the propagation of data leads to "bubbles" inside the circuit, this means, that the outputs of adjacent pipe registers carry the same phase for a short period. To show this behavior, we zoomed out the data consumption of Figure 5.26.

In Figure 5.27 we recognize that the bubble propagates in the opposite direction to the data waves and compensates the "gap" originating from the propagation of the data waves as illustrated in Section 5.1.4.

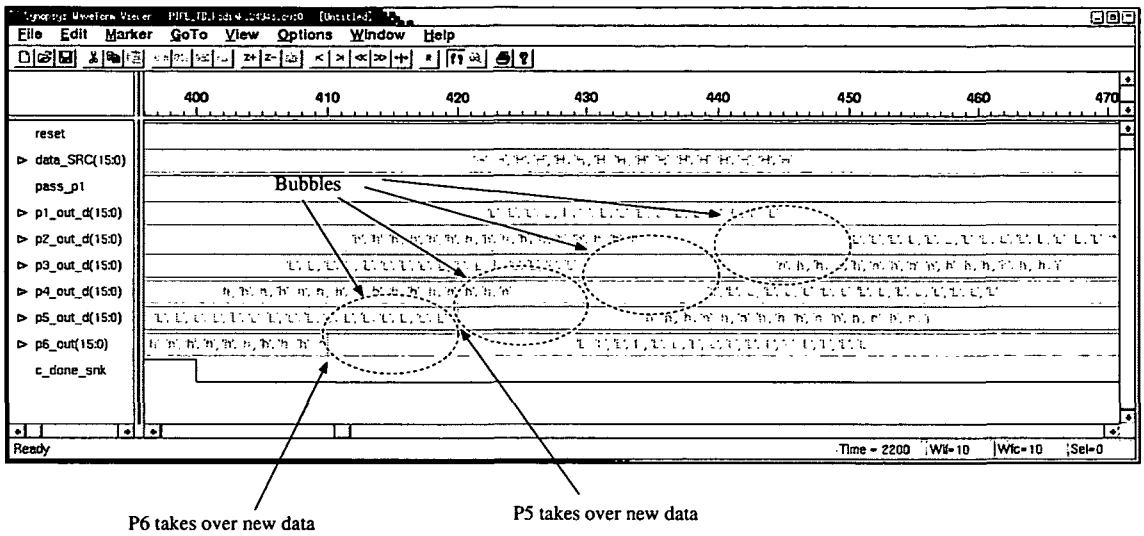


Figure 5.27. Data Propagation in Detail

In the next step we added the feedback path and applied the same stimuli to the pipeline. This simulation is shown in Figure 5.28.

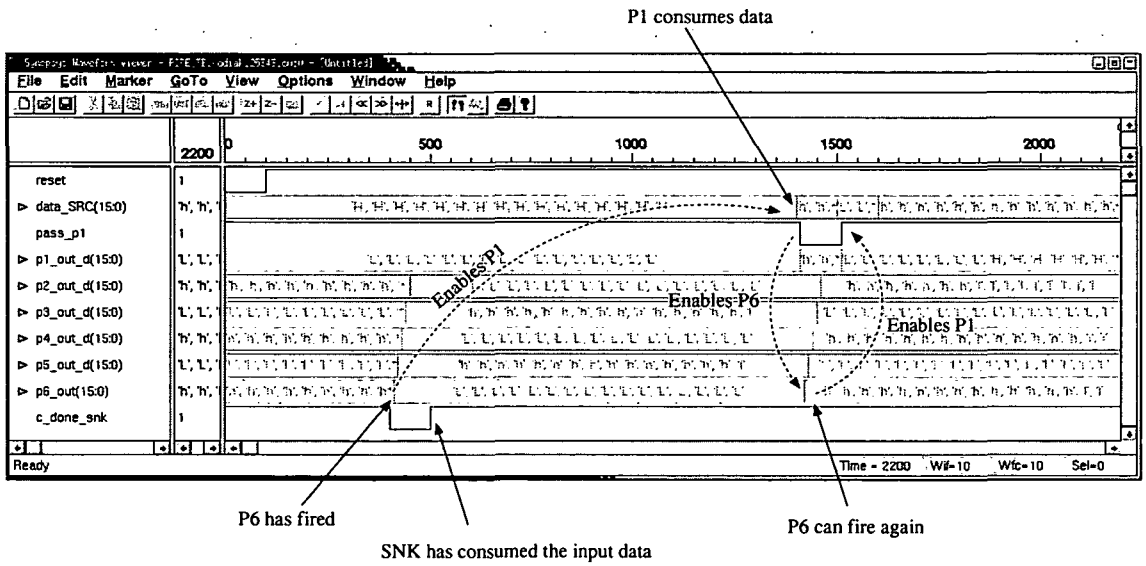


Figure 5.28. Structural Regulation of the Data Flow

We see that *P6* fires immediately after *SNK* has consumed its output, which is signalized by the falling edge of the *capture_done* signal of the sink node (*c_done_SNK*) – this enables *P1* to fire. The SNK in turn consumes the (new) output of *P6* again (see rising edge of *c_done_SNK*). However, *P6* is prevented to fire again due to the structural regulation of the data flow: Only when *SRC* issues new data and *P1* takes it over, then *P6* is enabled to fire again. The simulation confirms that the feedback

causes a structural regulation of the data flow.

The next experiments shall illustrate the impact of the initialization on the throughput. For this purpose we changed the behavior of *SRC* and *SNK* in the testbench, so that they react immediately: New data is issued promptly after *P1* has fired and the output of *P6* will be consumed, immediately after it becomes valid. Hence *SNK* and *SRC* does not constitute a bottleneck and the simulations display the highest achievable throughput of the circuit.

In Figure 5.29 the result of the simulation with an empty initialized pipeline is shown.

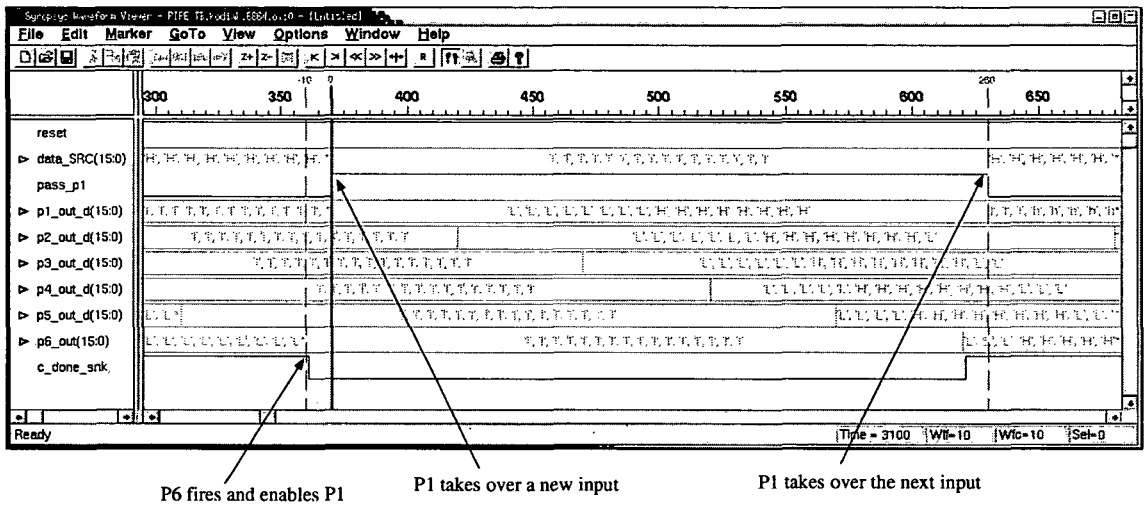


Figure 5.29. Throughput of an Empty Initialized Pipeline

First *P6* fires and enables *P1* to switch. As valid data lies on its input, this node can take over new data on its part. The delay of 10 ns originates from the switch delay of the pipe register. We may view that *P1* fires again after 260 ns – this equals to the theoretical value of the throughput defined by Equation 5.3:

$$\begin{aligned} \Delta Cycle &= \sum_{i=1}^{n}(\Delta SW(i) + \Delta FU(i)); \\ \Delta Cycle &= 6 * \Delta SW + 5 * \Delta FU; \\ \Delta Cycle &= 6 * 10ns + 5 * 40ns = 260ns; \end{aligned} \tag{5.6}$$

Note that the feedback path does not contain a *FU*– this, however, is not mandatory. An additional delay in this path has the same impact on the throughput as the delays of all other function units: it has to be added to the sum of all other delays.

Now we will consider the full initialized pipeline. We have used the same environment as with the empty initialized pipeline. Due to the fact that the condition $(k-2) * \Delta SW \geq \Delta FU$ is fulfilled the expected cycle delay is defined by the sum of the switching delays (= 60 ns). This was verified by the simulation in Figure 5.30.

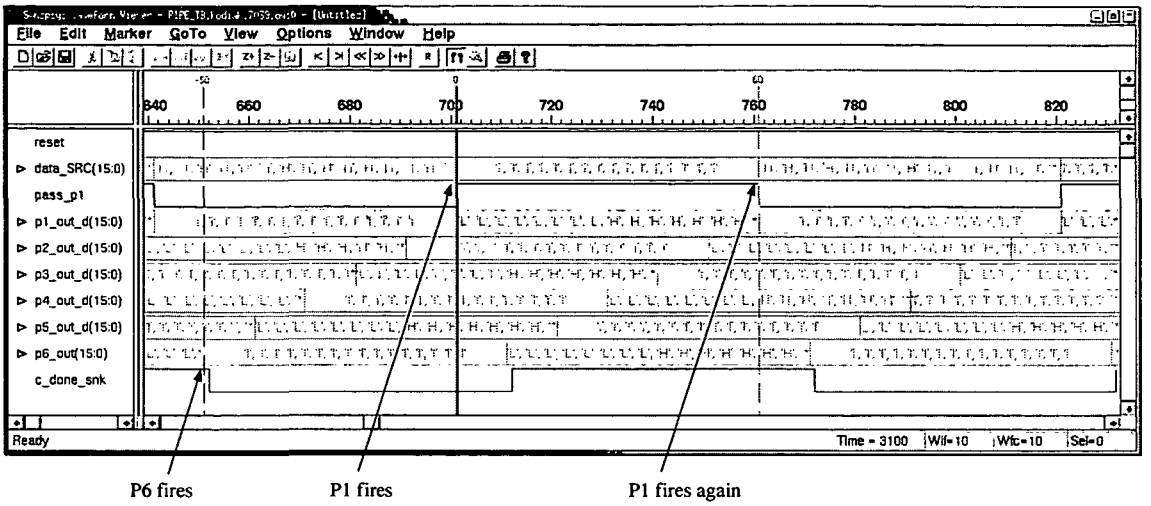


Figure 5.30. Throughput of a Full Initialized Pipeline

In contrast to the empty pipeline, where *P1* has fired 10 ns after *P6* has consumed its input, in the full initialized pipeline 50 ns elapse between those events. This has its origin in the different event sequence: in both circuits *P6* activates *P1* – in the empty initialized pipeline *P1* can fire immediately due to the fact that its output was already consumed (or is empty). In the full initialized pipeline the output of *P1* has not been consumed yet. In fact we can observe in the simulation that all nodes between *P6* and *P1* fire before *P1* can take over the input data on its part.

To investigate the case where the condition $(k-2) * \Delta SW \geq \Delta FU$ is violated we enhanced the delay of the function units *FU 5* to 70 ns. As a consequence, we have to use Equation 5.4 to calculate the cycle time.

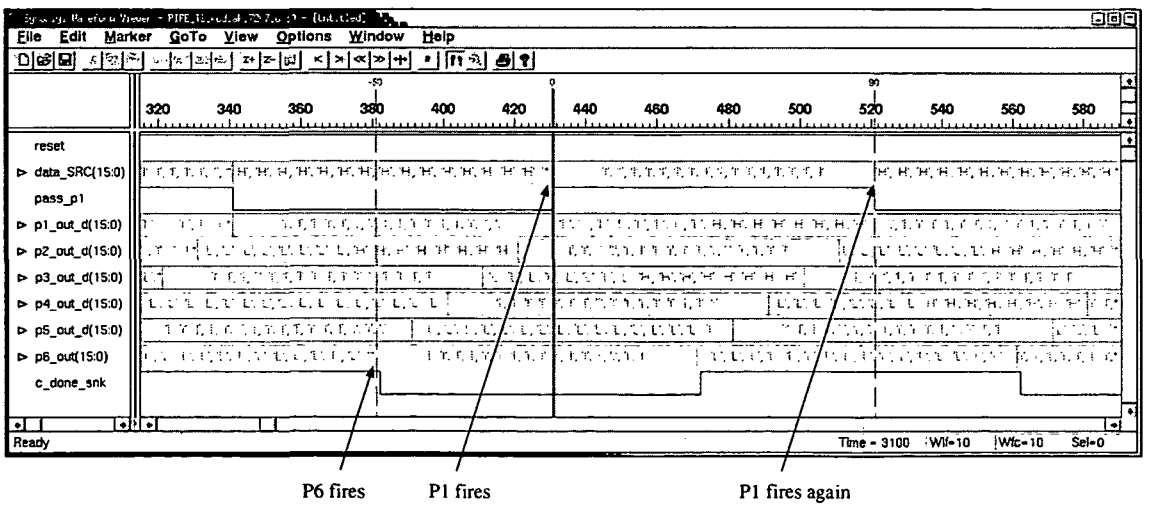


Figure 5.31. Throughput of a Full Initialized Pipeline with slow FUs

The simulation confirmed the theoretical approach: In Figure 5.31 the cycle delay

is equal to 90 ns, which is equivalent to two times the switch delay plus the delay of the slowest function unit.

For the last simulation we inserted more bubbles in the full initialized pipeline – this leads to a mixed initialized pipeline i.e. one part of the pipeline is empty and the other part full initialized. Regardless of the activities of the *SRC/SNK* these bubbles can move inside the circuit. However they cannot dissolve due to the structural regulation caused by the feedback path. This leads to an apparently unpredictable, but periodical behavior of the circuit, as illustrated in Figure 5.32

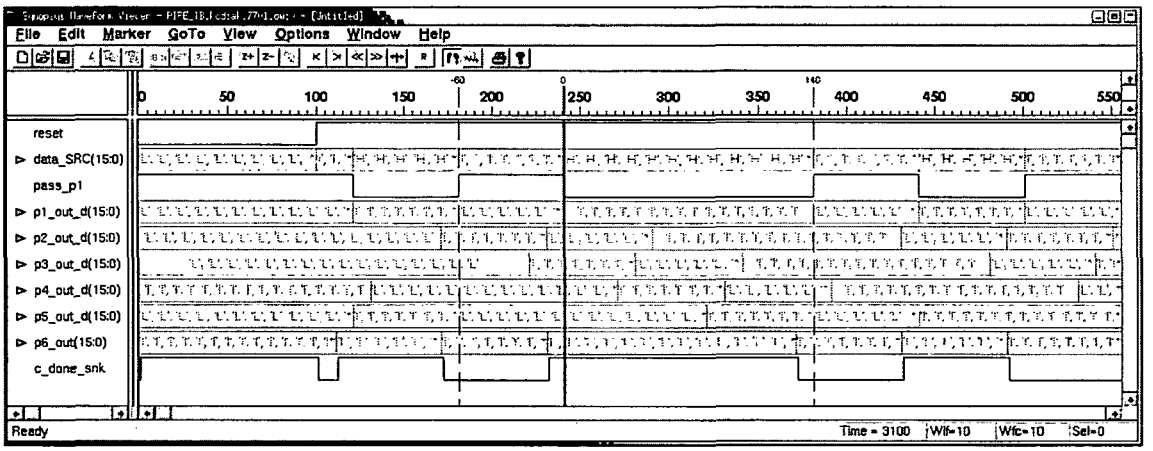


Figure 5.32. Non-Linear Pipeline with Bubbles

By means of sophisticated analysis approach the behavior can, of course, be modelled. However, this topic goes beyond the scope of this thesis and therefore will not be further considered.

5.1.11 A Short View to Other Design Styles

As illustrated feedback/forward paths may cause a deadlock due to the potentially encoding mismatch between the "regular" signals and the feedback/forwarded ones. Such a mismatch can not happen if no signal encoding is used. Hence, all asynchronous design styles, which use a single rail encoding are not directly affected by this problem – they move this issue into the time domain, by considering the feedback/forward paths during calculation of the clock period or the matching delays respectively.
As NCL uses a dual rail encoding, this design technique is subject to the same risk of deadlocks in conjunction with feedback and forward paths as CAL. However there exists no counterpart to CAL's phase inverter in NCL. Thus if "phase inverters" are required in NCL circuits complete registers have to be inserted in feeback/forward paths. This penalizes the performance and the area efficiency in comparison to the CAL solution.

5.2 Selecting Nodes

Until now we have considered only nodes that consume all their inputs and accordingly set all their outputs. Nevertheless, we have already encountered a circuit where this assumption is violated, namely Figure 5.3. Let us consider this circuit in more detail again (see Figure 5.33). In the previous section we assumed that the virtual memory nodes (*Mem_WR* and *Mem_RD*) operate independently one from the other. However, a more realistic assumption would be that *L 1* issues either a write or a read command and therefore only one node, *Mem_WR* or *Mem_RD*, will be enabled.

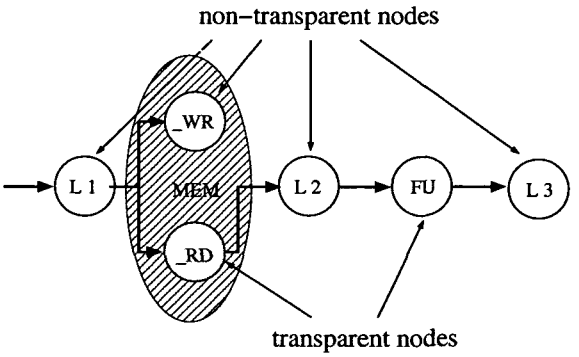


Figure 5.33. Virtual Memory Nodes

Hence *L 1* will deliver data to *Mem_WR* or (through *Mem_RD*) to *L 2* – as a consequence the alternating coded sequence of data waves at the output of *L 1* will be distributed in an arbitrary manner between *Mem_WR* and *L 2* (assuming an arbitrary distribution of write and read commands). This in turn causes the subset of data waves which are sent to *L 2*, for instance to show a random distribution with respect to their phase encoding. To operate properly *L 2* requires a strict alternating sequence of phases on its input - if we subsequently try to issue two data waves with the same phase encoding, then *L 2* will take over the first one, but not the second one. *L 2* will not recognize the second data wave as the next data wave because its phase encoding does not differ from the previous one. As a consequence, no acknowledgement will be sent back to *L 1*, which in turn causes that *L 1* cannot fire again and therefore a deadlock occurs. This problem takes place, because we do not want to read and write simultaneously. However, the mentioned deadlock has not to be confused with the deadlock that originated from feedback and forward paths - the origin of the problem is a completely different one: In contrast to the previous section, where we got inconsistent input vectors caused by signals which skip a pipe stage, in this section we will consider nodes, which require only a subset of inputs to perform their operation and/or set only a subset of their output signals accordingly. We call such nodes *selecting nodes*. The critical task in conjunction with these nodes is to ensure that the data paths, which are connected to the disregarded input signals and/or to the unselected outputs signals do not loose the synchronicity with the remaining circuit.

5.2.1 The Root of the Problem

The root of the complication with selecting nodes is the *fluctuation of validity* in CAL. To the essence of the meaning one has to return to the basics of CAL focusing our attention to the dynamic characteristic of signals with respect to their validity. In Figure 5.34(i) a node *L 1* with one input signal is illustrated[4]. We can not judge validity of the signal without considering the state of the destination node. The state of a node, however, varies over time – it changes when data is taken over. Hence, a signal, which carries the same encoded information is considered as valid at one time and as invalid in the next instant (see Figure 5.34(ii) and (iii)).

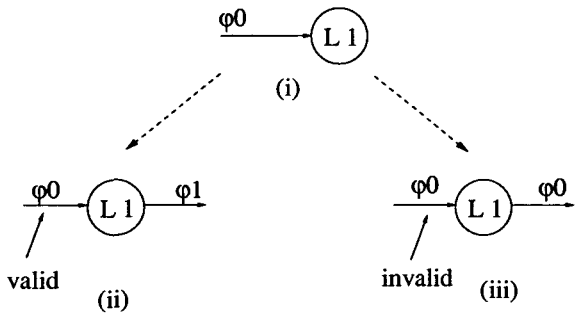


Figure 5.34. Fluctuation of Validity

In a continuous data path the fluctuation of validity is intended and used for data flow control purpose. However, there may be some nodes in a circuit that propagate data in a selective manner, such as a de-multiplexer, for instance. As already outlined such nodes cause all (alternative) downstream paths to change phases in a irregular fashion. Due to the fluctuation of validity the same output of the selecting node may be considered as valid for one node and as invalid for another one. This corrupts the control flow mechanism inherently in the CAL encoding style and inhibits the correct operation of the circuit. In the following we will differentiate between several types of nodes that may cause such irregularities in the control flow.

5.2.2 Selecting Node

Nodes that consume all input signals to perform their operation and always set all of their output signals, are easy to handle with respect to data flow control: (i) the node waits until all input signals become consistent and (ii) a new data wave is only generated when all destination nodes has consumed their input data (see Section 3). Now we will consider, what follows when only a subset of the input signals is required or when the generated output data is intended to be consumed only by a subset of all destination nodes[5].

[4]Instead of an input vector we will consider only one single signal for now – in this way consistency is always guaranteed

[5]Even mixed forms are possible, namely nodes which require only a subset of their inputs and set only a subset of their outputs. However these nodes can be represented by two logical nodes, where

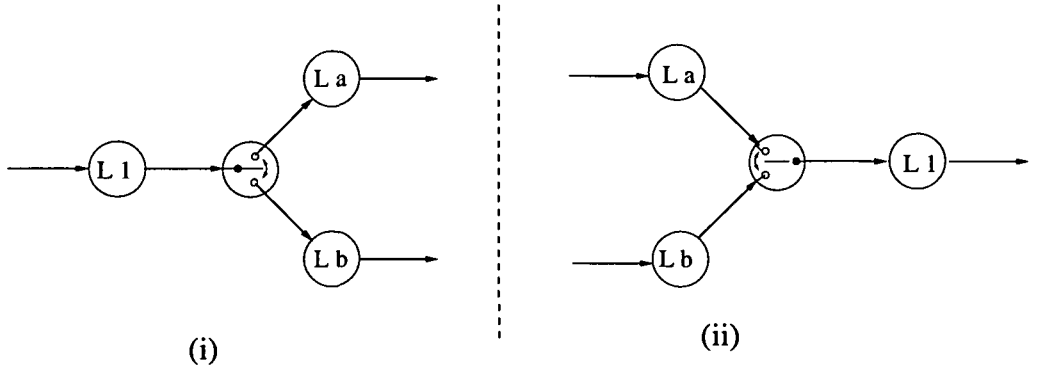


Figure 5.35. (i) Split Data Path (ii) Combined Data Path

In Figure 5.35 two circuits are shown, on the left we can view a circuit which splits a data path: The output data of *L 2* will be transmitted either to *L a* or to *L b*. This is indicated by the switch circuit inside the subsequent node to *L 1* – hence we call this node a *selecting split node*. On the right side of Figure 5.35 we see a circuit which combines two data paths to one: *L 1* receives its input data either from *L a* or form *L b* – this is indicated by the switch symbol inside the node which combines the data paths. In the following we will analyze the combination and the splitting of data paths in detail.

5.2.3 Combination of Data Paths

Data paths can be combined in two different ways: Either the selecting node puts the data from *L a* and *L b* to its output in an alternating sequence or both data packages are considered to belong to the same context, but only one of them will be passed through.

As we see in Figure 5.36(i) all data packages are passed through in the first case – the selecting node operates similar to a zipper and "serializes" the data of all input nodes. Therefore throughput of *L 1* is twice the size of *L a* or *L b*[6]. We call this operation mode *merge mode*. In contrast Figure 5.36(ii) shows a circuit, where the same number of data packages passes through all nodes. This selecting node discards data packages of non-selected the inputs and *L 1* will only acquire the selected data package. This corresponds to the functionality of a multiplexer. Thus we refer to this operation the *multiplex mode*, or *MUX mode*

Merge mode As we have already illustrated in Figure 5.36(i) the merge mode combines the data paths to a single one and no data gets lost. The first consequence is that the merge mode requires individual handshake signals for each input data stream.

---

the first one evaluates the selected input vectors and produces a "virtual output" and the second node distributes this output to the intended destination nodes. Hence we will only focus our interest on these two basic selecting nodes only.

[6]Assuming a balanced selection between the data paths

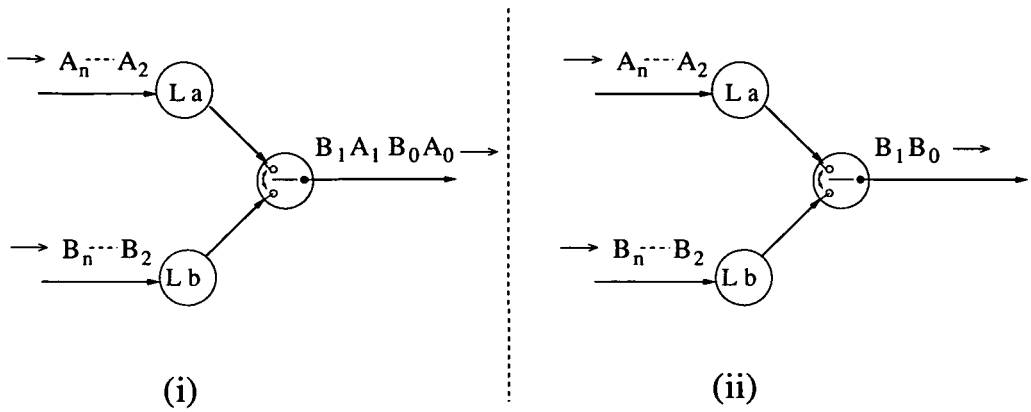


Figure 5.36. (i) Merge Mode (ii) MUX Mode

Figure 5.37 shows the principle of the merge mode again. Now we may recognize the problem of this operation mode: To operate properly both data paths must each carry data coded with alternating phases. Combining these data paths to one common data path, the alternating data encoding is violated in the resulting data path[7].

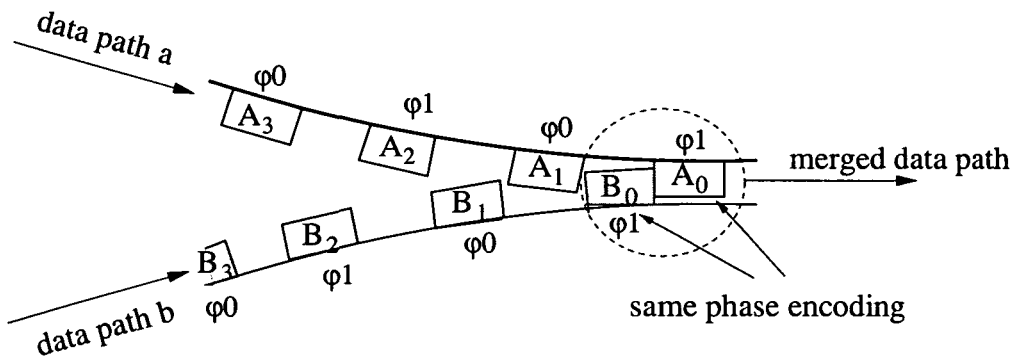


Figure 5.37. Merge Operation

We implemented the example circuit (illustrated in Figure 5.38) to explore in greater detail the behavior of the merge circuit.

The circuit was configured in such a way that *data path a* issues even numbers and *data path b* issues odd numbers only. To get a more clear picture, we set up the testbench so that the first five data packages are consumed from *data path a* and afterwards it switches to *data path b*. In Figure 5.39 we can see the result of the simulation. When data is consumed from *data path a*, then data on *input b* does not change. Contrariwise, when *input b* is selected, then *data path a* keeps a constant signal value. Thus the circuit operates correctly. In the next simulation we switched between the data paths one step earlier. As illustrated in Figure 5.40 a deadlock takes place in this case.

[7]The same is true when we have an arbitrary switch sequence, the crucial point is that no data package will be discarded

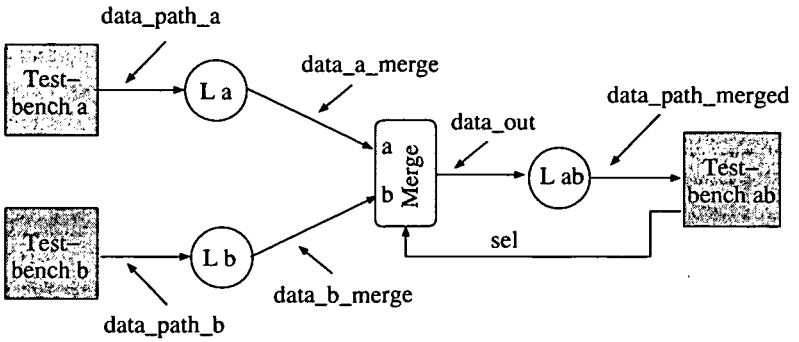


Figure 5.38. Merge Circuit

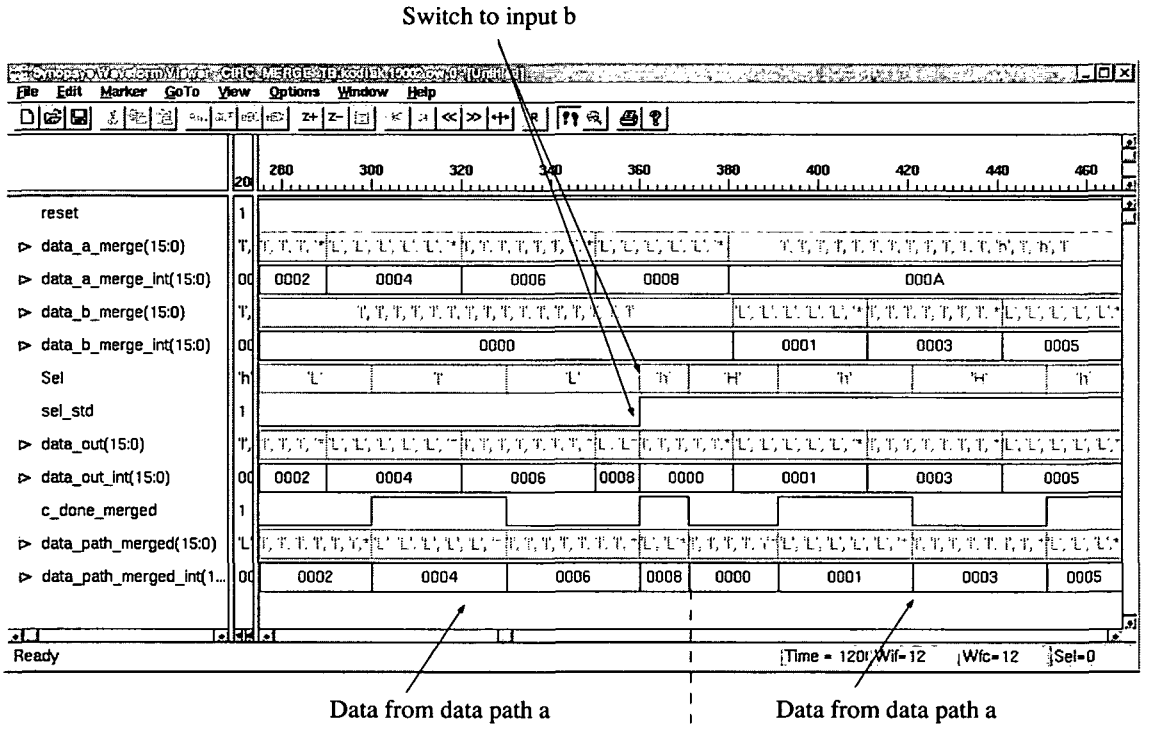


Figure 5.39. Merge Operation without Deadlock

What is the difference between these simulations? If we take a closer look at the first simulation we may recognize that the data, which lies on the *input b* of the merge element is encoded in the phase that is required next after the switch event. Hence the select signal *sel* and *data_merged_b* are consistent – the selecting merge element can fire and the data package of *data path b* can be consumed.
As aforementioned, in the second simulation we switched one data package earlier – as a consequence the encoded input data lying on *input b* does not correspond to the requested phase encoding: This leads to an inconsistent input vector and as a result the merge node cannot fire. How can we still implement a generic merge circuit?

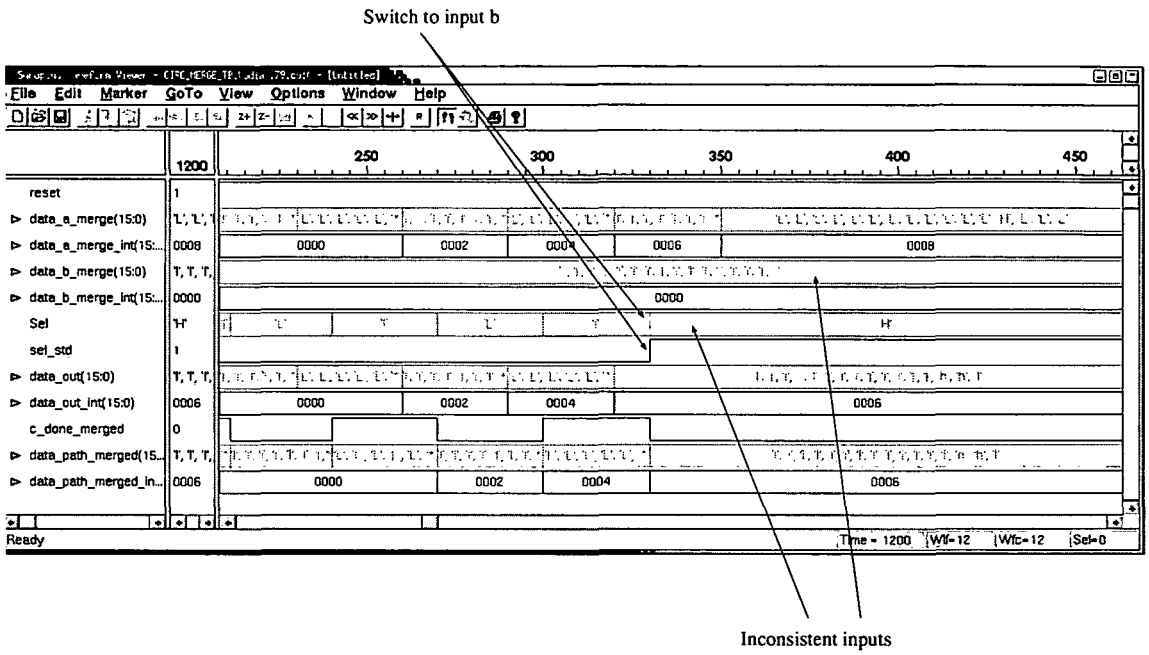


Figure 5.40. Deadlock as Consequence of a Merge Operation

The alternation of data encoding is a basic principle of CAL – therefore it must be guaranteed for all data paths. Thus the only solution is to insert a *synchronizer circuit* between those data paths, or, to be more exact, one *synchronizer* for each incoming data path. Figure 5.41 depicts the block diagram of a synchronizer circuit and shows where its has to be placed in the merge circuit.

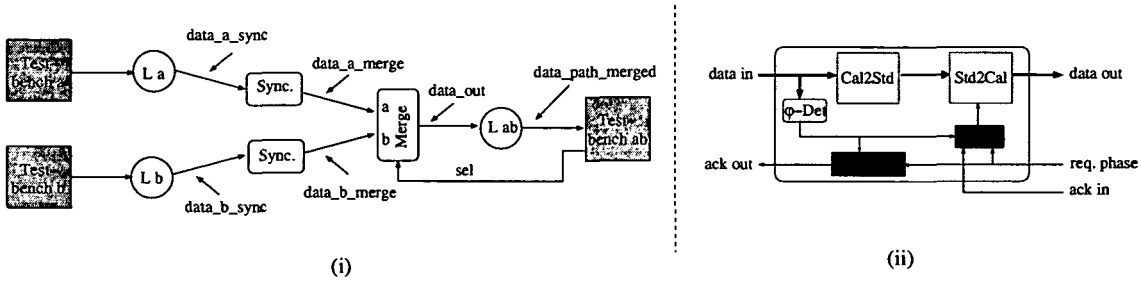


Figure 5.41. (i) Merge Structure with Synchronizer Circuit (ii)Synchronizer Circuit

The synchronizer circuit operates in two modes:
(i) If the data package on its output has already been consumed then it must be ensured that this data will not be used again: Thus the synchronizer switches its output in such a way that it is always in an invalid state.
(ii) If the input package has not been consumed yet, then the synchronizer ensures that its output carries everytime valid encoded data, in order to ensure that no deadlock occurs, when the merge element switches to its output. Obviously the synchronizer must switch between these operation modes when its output data was consumed or

when it receives a new (valid) input data.
How can the synchronizer determine the phase encoding, that leads to a valid or to an invalid data encoding? For this purpose the *c_done_merge* signal (= capture done signal of the destination node) can be used: The destination node of the *merge* element signalizes the encoding of the last consumed data – thus the synchronizer switches its output to this phase, if it wants to build invalid data and in the other phase otherwise.

We repeated the previous simulations again, and the results in Figure 5.42 and 5.43 demonstrate that now the merge circuit operates always correctly.

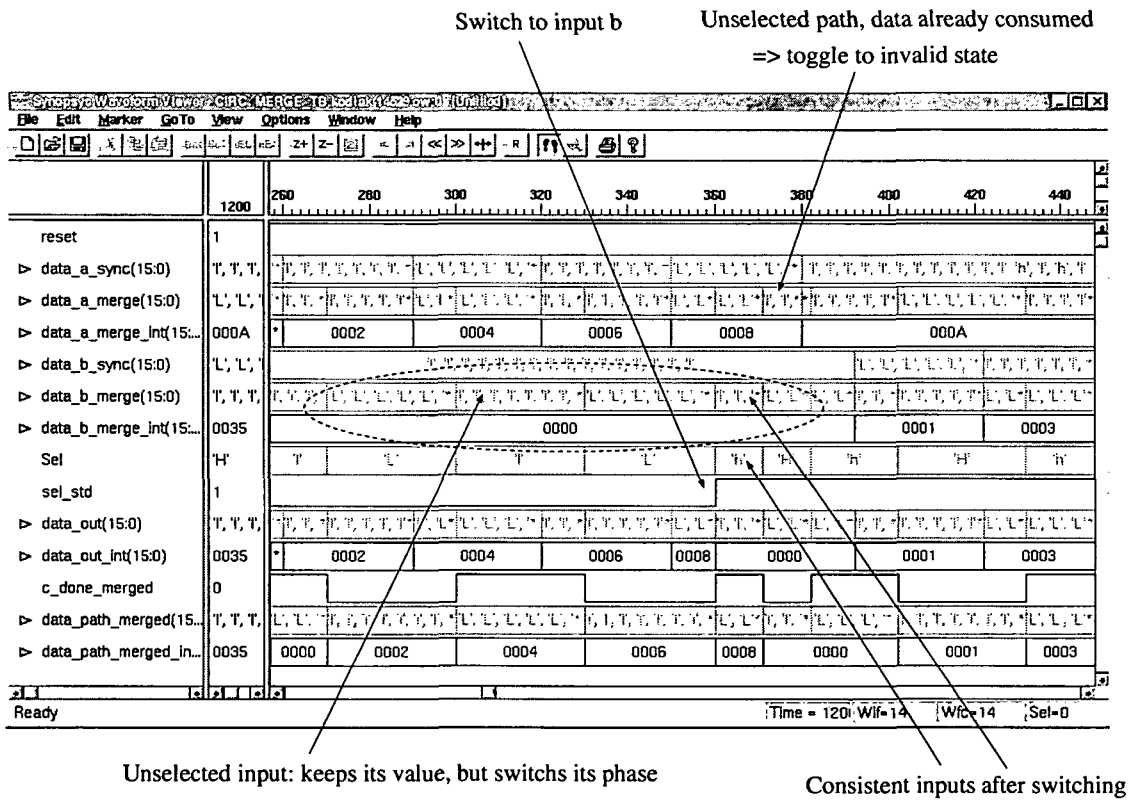


Figure 5.42. Merge Operation with Synchronizer Circuit (I)

We recognize that in both simulations the synchronizer toggles *data_b_merge* in such a way that the input vector of the *merge* element is always consistent. Furthermore we highlighted in Figure 5.42 that the synchronizer on *input a* toggles its output data to the invalid phase, due to the fact that its data path has not been selected yet, but its output data has already been consumed. However, the synchronizer separates the data and phase information – this is necessary to be able to generate the data with the requested phase encoding at the outside of the circuit. To prevent that the old data is re-encoded with the new phase encoding information, it must be guaranteed that the pure data information arrives earlier at the *Std2CAL* component than the phase information. This involves a timing assumption and therefore a synchronizer circuit cannot be delay insensitive.

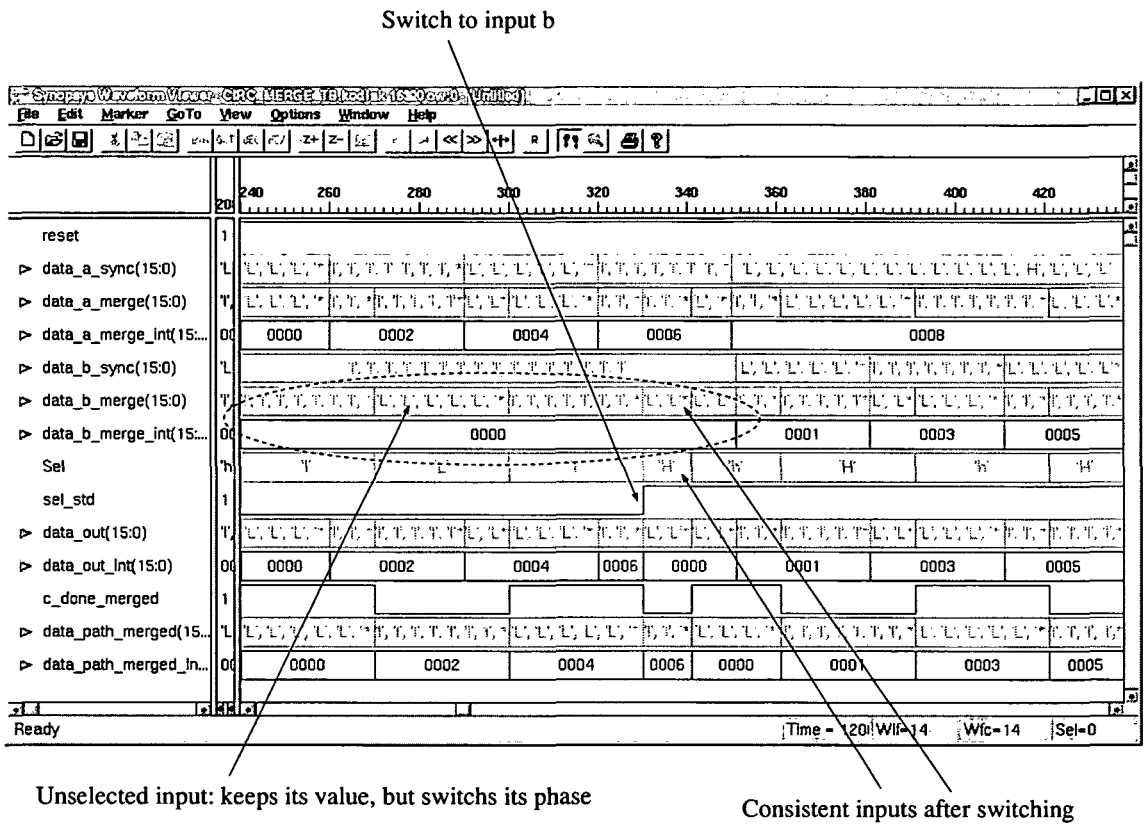


Figure 5.43. Merge Operation with Synchronizer Circuit (II)

Another (hidden) timing assumption has already been made even on the gate level: The time the synchronizer requires to switch its output from one coding style to the other one must be shorter than the delays resulting from the time required by the selecting node to switch to the other data path plus the time to consume the next data wave plus the time required to switch back to the original data path. This constraint seems to be easy to maintain – **however, from a theoretical point of view the resulting circuit is no longer delay insensitive nor speed-independent.**

MUX mode: Similar to the merge mode in the MUX operation mode only one input will be selected and passed onto the output, but data on all unselected inputs will be **discarded** instead. As a result the same number of data packages passes through all nodes. This simplifies the data flow control: A selecting MUX node operates similarly to a conventional node, which consumes all its inputs: This allows us to connect directly – this means without any additional control circuits – all acknowledged inputs of the source nodes with the request signal of the destination node.
Now the questions arises: Do we have to wait until all input vectors are valid, or can we produce the new output immediately after the selected inputs (and the related control signals) are getting consistent and valid? From a performance point of view it would be

reasonable to switch immediately. This decision is also motivated by one of the basic principles of asynchronous circuits, namely to start to working as soon as possible.
To analyze the impact of this decision we will consider the simple circuit example in Figure 5.44. This circuit is composed of a multiplexer (=selecting node) and two latches. As a result of the considerations above, we assume that the multiplexer switches when the control signal and the selected input vector are valid, we call such a multiplexer to be "eager".

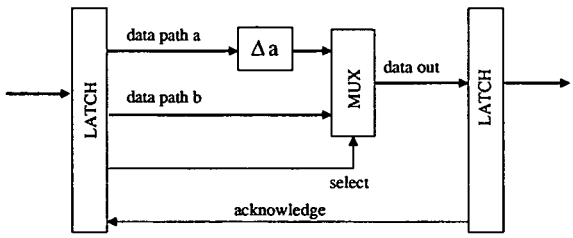


Figure 5.44. Circuit with Multiplexer

Due to the fact that we aim of delay insensitive circuits, we insert an arbitrary delay Δa into *data path a*, while *data path b* and the select signal are not delayed. To investigate how the circuit operates we assume that *data path b* is selected the first two times, while *data path a* will be selected in the third operation step. In the first step *Latch 1* issues data coded in $\varphi 0$ for instance. Due to the fact that the signals on *select* and *data path b* arrive immediately at the multiplexer, the latter is enabled to switch and *Latch 2* consumes the data. This in turn enables *Latch 1* to fire again. In the next step $\varphi 1$ will be issued and *data path b* will be selected again. After *Latch 2* has consumed this data wave, *Latch 1* will issue the third data wave coded in $\varphi 0$. As mentioned above in this third step *data path a* will be selected: Recall that on *data path b* data waves have been issued in the previous two steps as well (coded in $\varphi 0$ and $\varphi 1$, respectively), but these have not been selected. If Δa is larger than the time which was required for the previous two operations, then the first data wave (which was coded in $\varphi 0$, too) has not reached the output of the delay element yet and hence not actually been discarded although it was meant to be. When this data wave reaches the output it composes a consistent input vector with the select signal of the third data wave and the MUX will switch leading to malfunction of the circuit. Δa can be expressed more formally as follows:

$$\Delta a < 2 * (\max_{Sel|DPa} \Delta Process + \Delta MUX) + 2 * \Delta SW(1) + 2 * \Delta SW(2) \qquad (5.7)$$

$\Delta SW(i)$ is the switch delay of *Latch 1* and *Latch 2* respectively, while $\Delta Process$ is maximum of the delays of the *data path a* and ΔMUX is the time the multiplexer requires to switch.

The validity of this finding can be confirmed also by simulations: We implemented the circuit in Figure 5.44 and issued on both input of the multiplexer a counter value, which is incremented by the testbench. As a result we expect that the output would

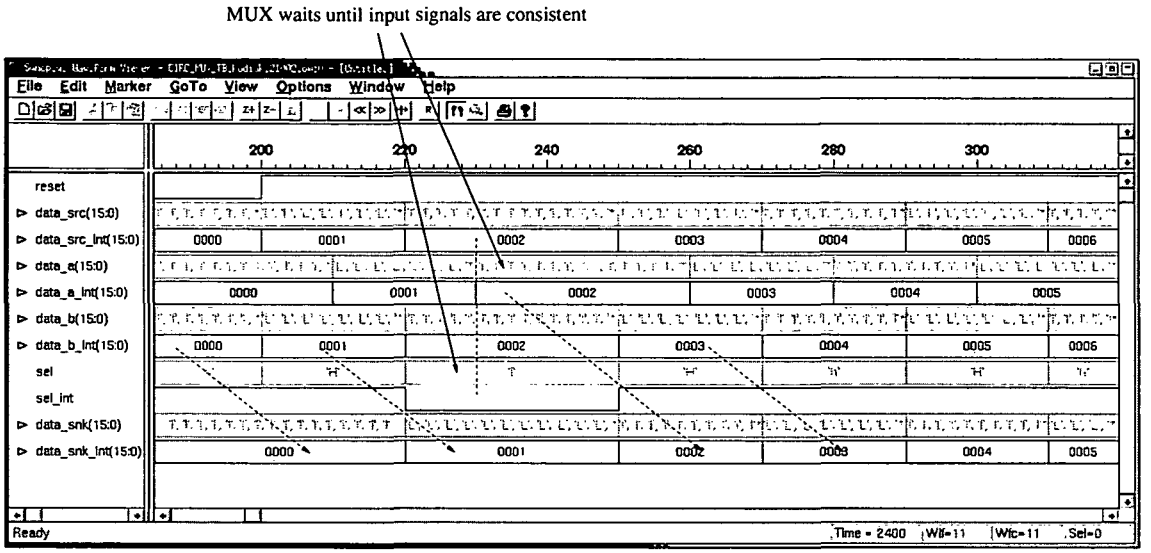


Figure 5.45. "Eager" Multiplexer Circuit with Balanced Input Delays

show a continuously incremented value. For the first simulation we set Δa and $\Delta SW(i)$ to 10 ns. ΔMUX and $\Delta Processing$ are set to zero. The simulation in Figure 5.45 shows that the circuit operates correctly.

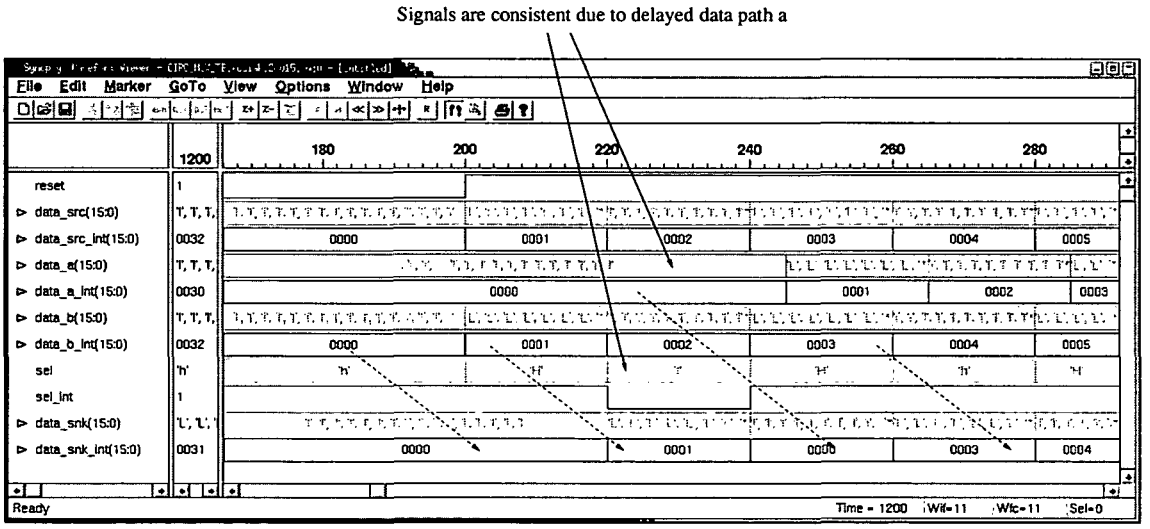


Figure 5.46. "Eager" Multiplexer Circuit with Unbalanced Input Delays

For the next simulation we incremented the delay Δa to 45 ns. We recognize in Figure 5.46 that the multiplexer takes over an old data package as follows from the above explanation.

To avoid this problem MUX has to wait until all inputs (even the non required ones) are valid and consistent before it fires.

Hence for the simulation in Figure 5.47 we used a non-eager multiplexer. We

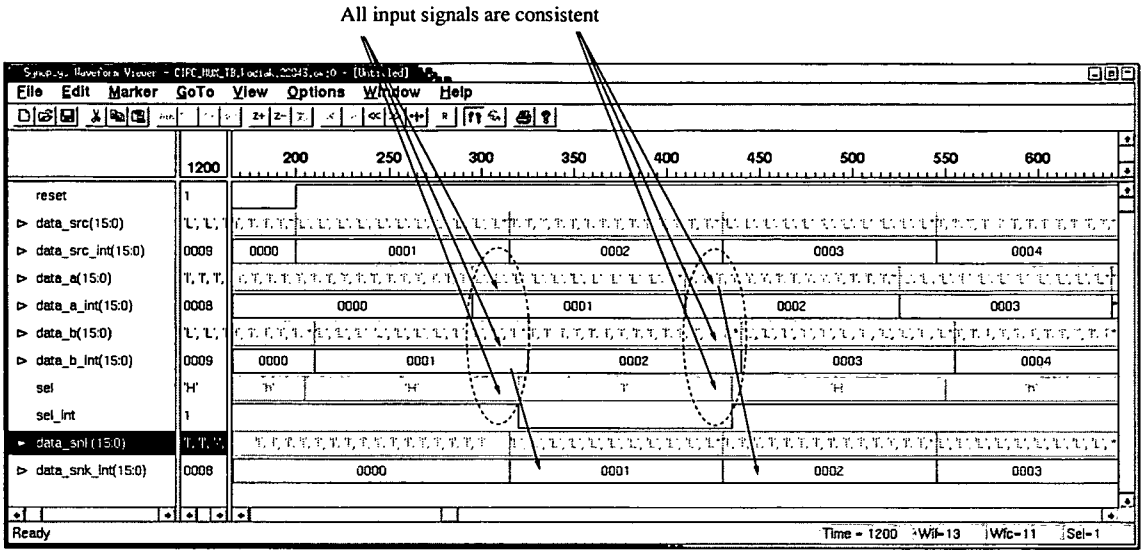


Figure 5.47. "Non-eager" Multiplexer

recognize that the multiplexer operates correctly – the price for this is performance loss. The simulations showed that the throughput of the circuit decreases due to the fact that the "non-eager" multiplexer waits always until the data, conveyed by slowest data path, becomes valid.

This circumstance can also be portrayed on a higher abstraction level: We have to take a look at the context relation between the data packages on all inputs of the selecting MUX node. The fact that all inputs are possible candidates to be passed through to the output implies that all data packages must appertain to the same context – each of them could be selected and propagated to the output of the multiplexer representing the next context for the destination node (see Figure 5.48). This in turn implies that some kind of synchronization between the data paths is required to guarantee that the order of contexts is kept. Due to the fact that we will consider a generic circuit, we can not expect that this synchronization is ensured anywhere else in the circuit – and as shown in the previous example a malfunction may occur, even if such a synchronization exists (*Latch 1* synchronizes the input data in the circuit example in Figure 5.44). Therefore the selecting MUX node has to wait until all inputs become valid before it can fire. Otherwise we have to make timing assumptions as shown in the example depicted by Figure 5.44, which results in speed-independent or timed circuits.

5.2.4 Split Data Path

A split data path is characterized by the fact that a source node has more than one destination node. The distribution of data in such a circuit structure can occur in two ways: (i) Data is transmitted to all nodes in a broadcasted manner – we will call this mode the *fork mode*. (ii) Data is only sent to the selected destination nodes – we will

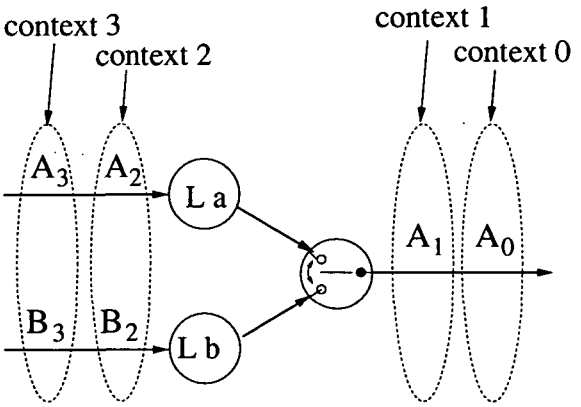


Figure 5.48. Context of Input Data

call this the *DEMUX mode*
The fork operation mode can be easily handled, since all nodes are recipients of all data waves, the source node has to wait until **all** nodes acknowledge the data package before it replaces the current data by the next one.

The DEMUX mode cannot be handled in such a straightforward manner. As depicted in Figure 5.49 the input data is divided between *data path a* and *data path b*.

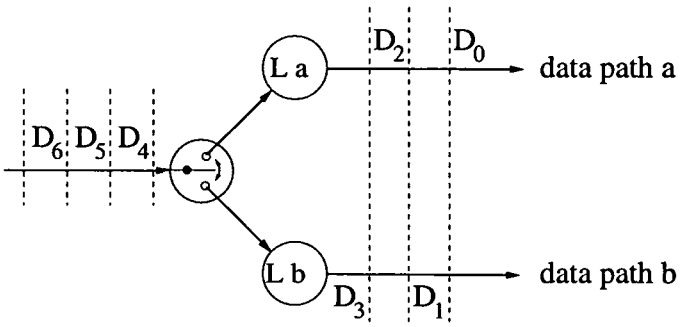


Figure 5.49. DEMUX Operation Mode

In Figure 5.49 an alternating selection between the output data paths is assumed. Hence *data path a* receives all packages with an even index and *data path b* gets all packages with an odd index. The CAL approach, however, requires adjacent data packages of the input stream D_0, D_1, ... D_n to be encoded with alternating phases. As a consequence of the switch activity of the selecting DEMUX node, *data path a* as well as *data path b* will receive data coded within the same phase only. It is easy to understand that even an arbitrary switch sequence will yield to a similar situation. There are two possibilities to solve this problem. Either we use a synchronizer circuit as explained in the previous section or we insert additional dummy data in both data paths to re-establish the phase alternation sequence.

Synchronizer circuits: As described in the previous section, can be used without modification for synchronizing splitting data paths as well. In contrast, the synchronizer has to be located at the output side of the selecting node (see Figure 5.50). The advantage of the synchronizer circuit is that both data paths are decoupled one from the other. The price which has to be paid is that the resulting circuit is no longer delay-insensitive nor speed-independent.

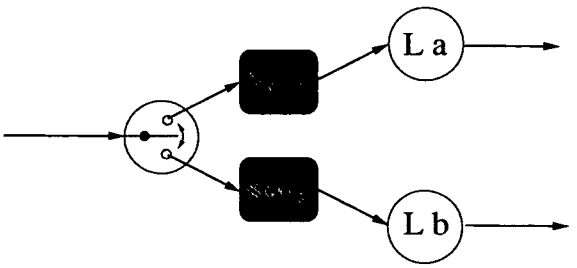


Figure 5.50. DEMUX Structure with Synchronizer Circuit

However, the synchronizer solution is not generic. It works fine if both data paths are independent from each other. If the data paths have a common node[8] in the remaining circuit, as illustrated in Figure 5.51, then a deadlock may occur:

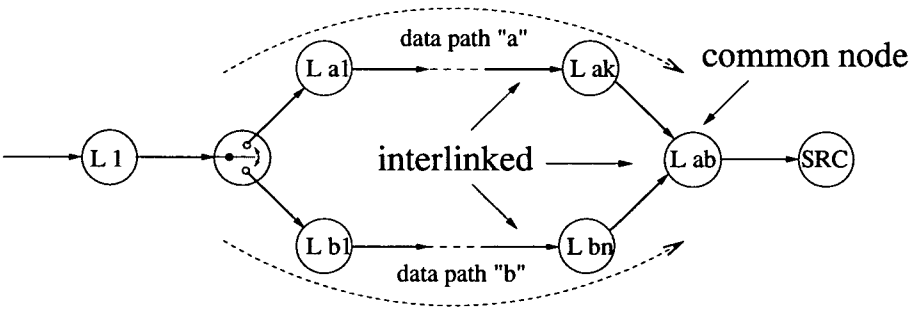


Figure 5.51. Interlinked Data Paths

If we assume an arbitrary switch sequence for the DEMUX node, then we must also permit the possibility that a data path will be selected more often in series: Let us suppose that this will happen for *data path a* in Figure 5.51. As a consequence the circuit that belongs to *data path a* will become full, while *data path b* will become empty. Consequently node *L ab* will be unable to fire and consume data on its inputs. This causes an accumulation in *data path a*. If this holdup reaches the selecting DEMUX node, then *data path b* can never receive new data and hence *L ab* will never be able to fire. As a result the holdup cannot be resolved and an deadlock occurs. Thus the synchronizer solution is suitable if we only want to connect independent circuits.

Dummy data: This approach is able to handle not only independent, but also tightly coupled data paths. The difference to the *fork mode* is that the destination nodes do

[8]We presume that the common node consumes all its input, in contrast to a MUX selecting node

not get the same data – the selected node gets the "real" data, while all other ones only receive dummy data[9]. The intention of dummy data is to keep the non-selected nodes up to date with respect to the current phase of the source node. In some sense the dummy data packages provide synchronization information (=phase) without any actual data. Figure 5.52 depicts how the dummy data is inserted into the data stream. Obviously the dummy data has to be inserted by the selecting node.

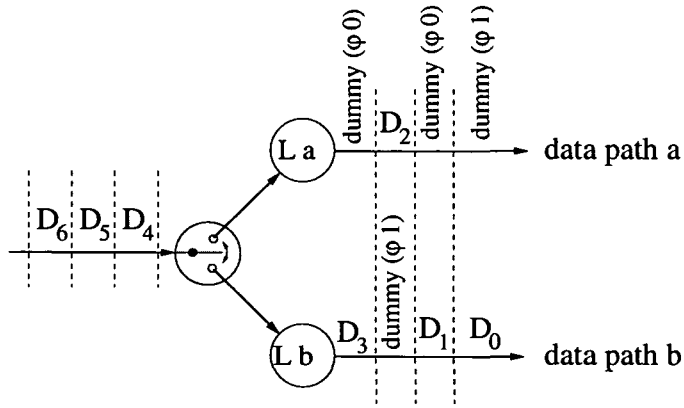


Figure 5.52. DEMUX Circuit with Dummy Data

Basically, there are two ways to generate dummy data: Either dummy data is marked explicitly by additional signals or so-called *NOP* (No OPeration) commands are used. The latter is only possible if the destination node supports such a command. Furthermore it has to be ensured that the node that gets the NOP command generates an acknowledgement. In the case of the *Mem_WR* node, mentioned at beginning of this chapter a NOP command can be easily generated by disabling the write command line. Due to the fact that in this circuit implementation all nodes consume some kind of data – even dummy data will be consumed – the source node has to wait until all nodes signalize that they have already used up their input data. This leads to a waste of performance, but the dummy approach conserves the delay insensitive character of the circuit. Furthermore, this approach can be directly applied to selecting DEMUX circuits with more than two outputs.

5.2.5 Tradeoff Between Performance and Delay-Insensitivity

We have shown that using selecting nodes in conjunction with the CAL coding style, delay insensitive circuits can be built only when fork/DEMUX nodes wait until all destination nodes have consumed its current output data and when MUX nodes waits until all inputs have issued new data before they performed their operation. (Merge node are an exception – they always yield to self-timed circuits). Waiting for an unselected data to arrive leads to a waste of performance, which is avoidable, if we

[9]Dummy data has not to be confused with the NULL-wave in NCL: dummy data is inserted when it is needed – the NULL wave is an integral part of the NCL coding style. Dummy data can rather be compared with default values of registers, which are set when no write access takes place

move away from the dogma of delay insensitivity. Let us recall the circuit example of Figure 5.44: Having multiplexers in such small circuits, it is simple to estimate the timing of all data paths and hence allow the MUX to switch before all inputs are valid without the risk of a malfunctions. In fact in the literature functions, which perform their operation before all input are valid, are already considered and referred to as "eager functions" [85]. However the decision in favor of performance or delay insensitivity may depend on several other aspects and has to be performed individually for each circuit.

5.2.6 Short View to Other Design Styles

The selecting node problem does not affect all those design techniques which use single rail encoding style: The validity of signals has to be provided explicitly by these design techniques. Therefore the validity of data does not depend on the current state of nodes as it is the case in CAL. In this way it is easy to define a generic invalid state and hence avoid all the problems, which arises in conjunction with selecting nodes.
Similar to CAL NCL uses a dual rail encoding. Hence in this design technique the validity of data fluctuates as it does in CAL. However there is a crucial difference: NCL has a NULL wave which explicitly marks an intermediate state between two data packages. This intermediate state can be used for synchronization purposes. Hence an NCL circuit does not require *synchronizer circuits* to deal with the multi-source and multi-destination problem. The "eager" function problems, however still exists.

5.3 Summary

In this chapter we illustrated the impact of non-linear structures to the CAL design style. Two types of non-linearity were distinguished: forward/feedback paths and selecting nodes.
Due to the coding scheme of CAL, forward and feedback paths may cause a phase mismatch at the input of the destination component of the forward/feedback paths. This problem can be solved by selective placement of phase inverters. We show that to find out the position of the phase inverter we have to consider the initialization of the circuit as well as its dynamic behavior. Furthermore we discovered that forward/feedback paths yield to a structural regulation of the data flow and as a consequence the performance of the circuit depends heavily on its initialization.
The second type of non linearity is constituted by selecting nodes – these are nodes that requires only a subset of their input/output to perform their operation. We find out that there exist two solutions to deal with selecting nodes: One solution is to use so-called "non-eager" selecting nodes. These nodes have to wait until all input vectors (even the non-required ones) are valid or all output vectors are consumed respectively, before the new output can be generated. This penalizes the performance of the circuit, but yields to a delay-insensitive circuit. The second solution is to use synchronizer circuits and allow selecting nodes to become active immediately after the required input signals only become valid or when the selected output channel only was consumed. The synchronizer circuits guarantee that none of the involved data path loose its (phase-) synchronization. However, this implies timing assumptions which in turn yield to self timed circuits. The system designer has to decide whether the gain of performance justifies the loss of delay-insensitivity.

Chapter 6

ASPEAR - Asynchronous SPEAR

In this section we will show how we built the asynchronous processor core *ASPEAR* starting from the synchronous reference processor *SPEAR* described in Section 4. Using the design flow described in Section 3, the transformation from conventional designs to CAL circuits can be performed by (i) renaming all signals from *std_logic* to *cal_logic* and (ii) replacing processes, which build registers with *CAL_latch* instances. Obviously the handshake signals of the CAL latches have to be connected correctly.
However, the *SPEAR* processor core required a couple of additional modifications to permit us to apply the CAL design style. As illustrated in Chapter 5 we have to pay special attention to non linear structures.

6.1 Synchronous Reference Processor

6.1.1 Structural Adaptation of SPEAR

One of the key features of SPEAR is its scalability – there exist many different variants of the processor core. As we will validate the concepts presented in this work we have chosen the basic version of the processor core, without any additional extensions apart from the *Processor Control Unit*. However, a few modifications on the processor core itself had to be made, in order to fit the architecture for the CAL design approach:

Writeback bus: It is modelled in the original SPEAR processor as a tri-state bus: This allows us to connect an arbitrary number of additional *Extension Modules* without any further modification of the processor core. As depicted in Section 5.2 tri-state buses cause problems in conjunction with CAL – we recognized that the high impedance state on the bus leads to undefined and/or invalid input vectors for destination nodes. As a consequence the data flow mechanism is corrupted and a deadlock occurs. We have two possibilities to deal with the *Writeback bus*: either we replace the bus by a multiplexer or we ensure that the bus is always driven.

The latter requires a dummy module, which drives the bus when all other modules switch their outputs to high impedance. This leads to a costly "bus-keeper"

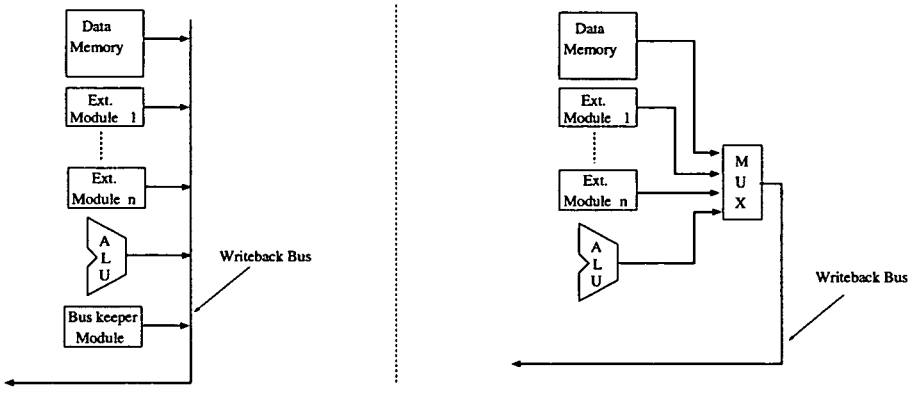


Figure 6.1. Tri-state Bus

module in terms of hardware since not only the entire hardware configuration must be known by the dummy module, but also the associated control signals have to be evaluated. The multiplexer solution has the disadvantage that a separate input port for each extension module must be provided. This procedure restricts the scalability of SPEAR. Furthermore a central control unit is required to generate the control signals which select the active input.
Note, simple pull-up resistors cannot be used, because the *Writeback Bus* must carry valid "dummy" data even in the idle state, which will be illustrated later on in this chapter. We chose the multiplexer solution for our purpose – the first prototype of ASPEAR is intended to validate the concepts elaborated in the previous chapter – hence scalability is not a crucial factor and for a small number of extension modules the multiplexer solution is less complex than the *bus keeper* module solution.

Incrementer: In ASPEAR, the program counter cannot generate the next address by incrementing the output of the *PC* and feeding it back (using a phase inverter) to its input. Such a feedback constitutes a problem for CAL latches – as illustrated in Figure 6.2 a "direct" feedback causes two competing data paths from the output of the latch to its input, which are the incremented output of the register and the output signal of the Φ – *Detector* which freezes the latches.

If $\Delta 1$ is less $\Delta 2$ then the incremented output will be consumed again before the latch is frozen. Thus a direct feedback affects the delay-insensitivity of the circuit[1] To avoid this problem we exploit the redundancy of a pipelined processor core and use the program counter value stored in *Pipe Register 1* as input for the incrementer.

Instruction ROM: To simplify the design we waived the program download module and replaced the *Instruction RAM* and *Boot-ROM* by a simple *ROM.*

[1]There are alternative solutions for this problem. These will be addressed in [117]

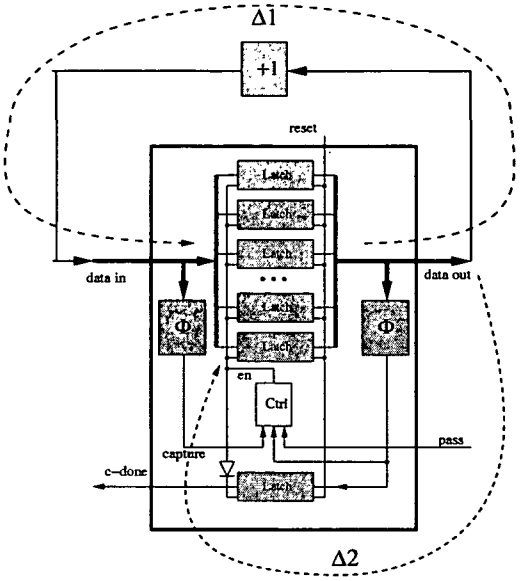


Figure 6.2. Incrementer Module

Figure 6.3 illustrates the processor configuration which served as the starting point for ASPEAR processor core.

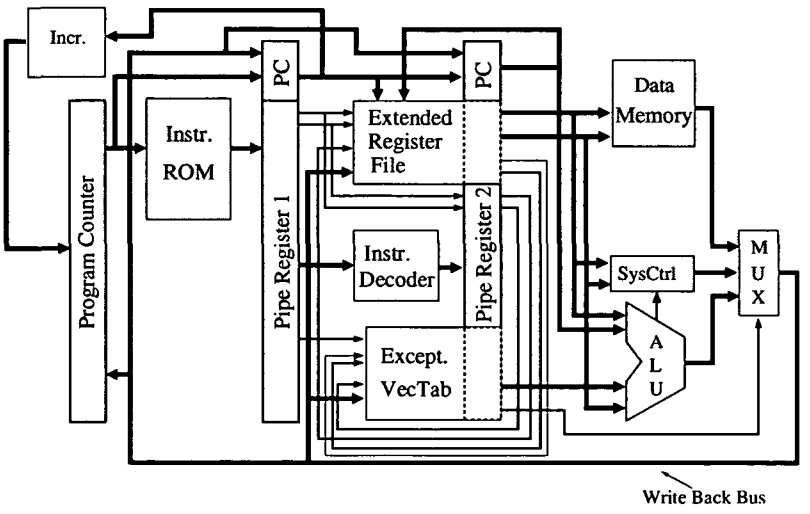


Figure 6.3. Reference Processor Core

6.1.2 Memory Implementation

Memories are considered to be asynchronous by their nature. However, in this context the expression "asynchronous" refers to memories that do not require a clock signal – it does not mean that validity and consistency of data are directly visible at the output of these memories.

Figure 6.4 depicts a read access. We assume that the read address changes exactly at

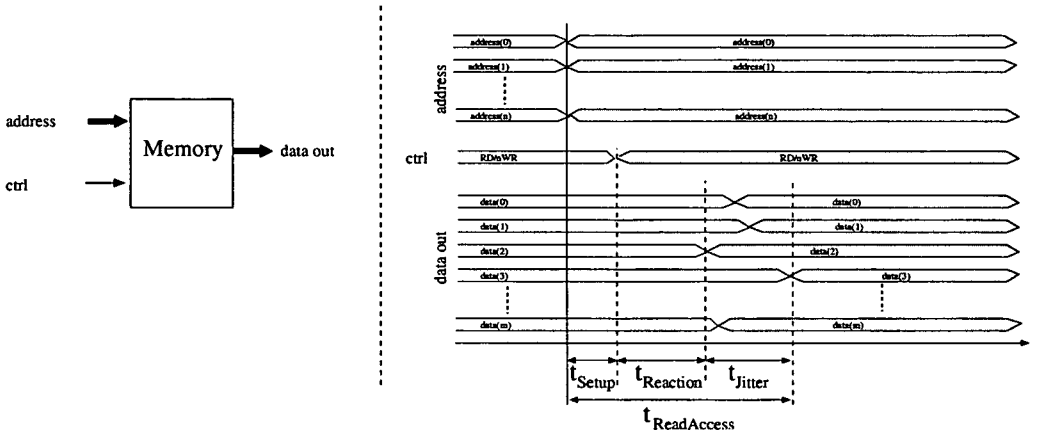


Figure 6.4. Read Access Timing

the same time – this does not imply that the related output data will also change in an ideal manner. One may view that quite the opposite is the case: As illustrated in Figure 6.4 the access time $t_{ReadAccess}$ is composed of three parts: (i) The setup time t_{Setup} which defines how long the read address has to be stable before a read command can be applied. (ii) The reaction time $t_{Reaction}$ which defines the period needed by the memory until the first output changes and (iii) the jitter time t_{Jitter} which expresses how long it takes until all remaining outputs are in the (steady) final state. These parameters, even if defined in the data sheets, are completely useless for delay-insensitive implementations of memories. A lot of research activities are ongoing in this field and some solutions were proposed: [100] proposes a current sensing approach to determine whether a memory operation is completed or not. In [101] a static SRAM design is presented: For the read access it exploits the fact that each SRAM cell is connected to two bit lines, which carry a complementary value during the read access and the same state otherwise. Thus, to determine the termination of read operations a dual-rail voltage sensing on these bit lines is used To signalize the termination of the write access, delay elements are utilized: Depending on the position of the memory, where the write operation takes place, different (matched) delays are used. In this way the average (write-) performance of the memory could be improved. In [50] a full-customized register bank is presented which is able to signalize the termination of read/write operations without any delay elements.
Due to the fact, that our target device is an FPGA we have no possibilities to change the embedded memory blocks. Therefore we have to implement a work around as illustrated in Figure 6.5.

Since we do not know in which phase data will be requested, we cannot store the *CAL* encoded signals directly in the memory. This forces us to separate the data- and the phase- information and to store data only – as a result such a memory implementation can neither be delay insensitive nor speed-independent. The conversion of the CAL input signals to *std_logic* is performed by the *CAL2STD* component. Note, that as

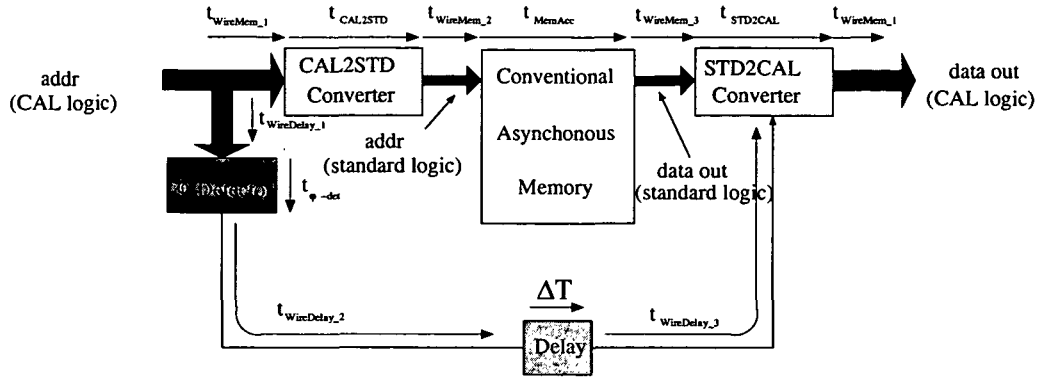


Figure 6.5. Read Access to a CAL Memory

illustrated in Section 3 this transformation is very simple due to the fact that *line a* of CAL encoded signals always carries the data information.
The output of the memory has to be re-transformed to a CAL_logic type. As the read access is transparent, input and output of the CAL memory have to carry the same phase. Thus the input phase is scanned by φ-*Detector* and used to build the CAL coded output signals. *STD2CAL* must receive the data read from the conventional memory before the (new) phase information is provided by the φ – *Detector*. Otherwise the old data would be encoded with the new phase information - subsequent components would consider this signal as the new data wave and consume it. To ensure that this does not occur ΔT has to be adapted so that the following equation is fulfilled:

$$\Delta T + t_{\varphi-det} + \sum_{i=0}^{3} t_{WireDelay(i)} \geq t_{CAL2STD} + t_{MemAcc} + t_{STD2CAL} + \sum_{i=0}^{3} t_{WireMem(i)} \quad (6.1)$$

In contrast to the read access where, the (CAL-)read control signal can be directly connected to the conventional memory the *WrEna* signal requires a dedicated treatment: We have to ensure that data and address are valid before the write signal becomes active. This can be easily achieved by using the output of the φ-Detector – it changes its state only when all input signals are valid and consistent. The *WR_CTRL* unit can use this signal in conjunction with the CAL write/read control signal to generate the *WrEna* signal for the conventional asynchronous memory. We still have to assume that the delay of the input data through the *CAL2STD* unit is less than the delay of the *WrENA* generated by the *WR_CTRL* unit. Thus the write access cannot be delay insensitive.

Figure 6.6 illustrates the final implementation of our CAL-memories. We are aware that this is only a less-than-ideal solution - in this work, however, our focus was to verify the concepts presented in Section 5 and hence we consider the CAL memory elements as basic blocks such as the other basic gates presented in Chapter 3.

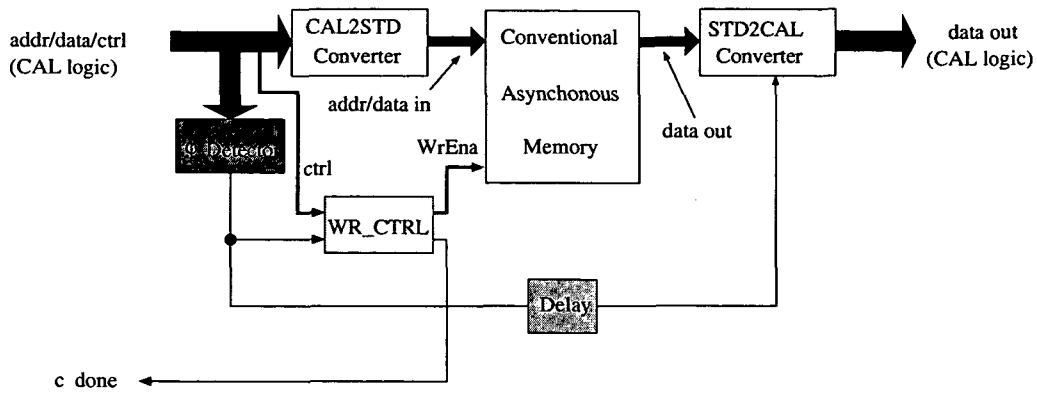


Figure 6.6. CAL Memory Block

6.2 Feedback and Forward Paths

Even the basic SPEAR version exhibits a lot of forward and feedback paths. The fact that we have only a short pipeline further complicates the placement of the inverter, because the entire non-linearity is concentrated in this small area. To get a first simplification, we will build a graphical representation of the processor core, where signals that have the same source and destination nodes are combined to a single edge.

6.2.1 Graphical Representation

To construct the graphical representation we have to identify what nodes are transparent and which are non-transparent: ALU, decoder and incrementer are combinational nodes and hence transparent. The pipe registers and the program counter regulate the data flow, thus they are non-transparent. All other nodes are memory components and therefore they have to be split into virtual transparent and non-transparent parts. These components are hatched in Figure 6.7.

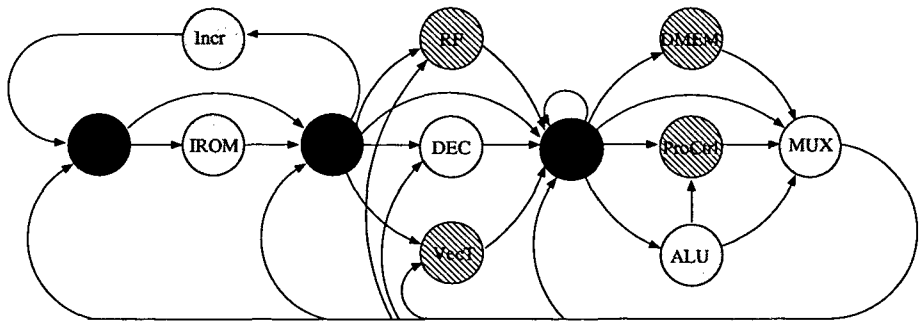


Figure 6.7. Direct Mapping from Components to Nodes

In the representation of Figure 6.8 we have split the memory components into transparent and non-transparent nodes and re-arranged the graph to evidence the logical

structure of the processor core. Note that the virtual nodes of the register file as well as those of the vector table are located in different pipe stages.

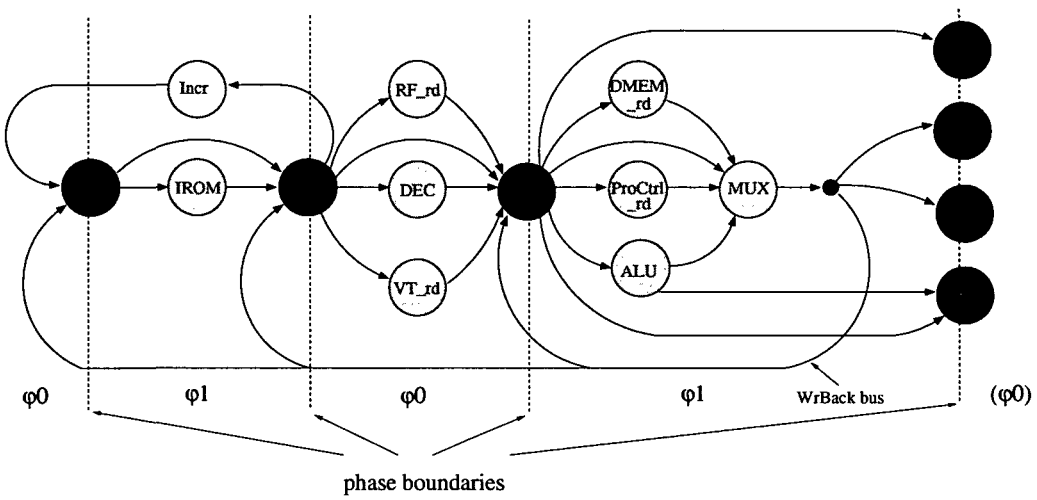


Figure 6.8. Graphical Representation of ASPEAR

We recognize that *RF_wr*, *VT_wr*, *DMem_wr* and *ProcCtrl_wr* act as data sinks for the processor. In a processor core the instruction memory can be considered as data source. However in the graphical representation of the processor core a dedicated source node can not by identified: The instruction memory is triggered by the program counter, which receives is next value from pipe register 1. The latter in turn obtains its input data from the memory and the program counter – this constitutes a self triggered loop (i.e. a loop that needs no external stimulation), which provides the input data of the processor core. As we can view in Figure 6.8 this loop is not the only non-linear structure – the processor core contains a series of other feedback and forward paths. Therefore we will identify the phase inverters required to accommodate these non-linear structures in the next step.

6.2.2 Phase Inverter Placement

As a consequence of the throughput considerations in Section 5.1.10 we decided to initialize the pipeline of the ASPEAR as full.

To identify the position of the phase inverters we used the algorithm described in Section 5.1.9. This program requires as input a textual description of the graph, which has the following format:

```
Sourcenode_type    Edge_weight    Destinationnode_type
```

The *type* of a node can be either transparent(T) or non-transparent(N), the *weight* of an edge defines the number of signals which are associated to it.

Having the a graphical representation such as in Figure 6.8 it is quite simple to derive the required textual description. We applied the algorithm several times by switching the starting node: The best constellation required only 92 inverters, the worst one 242 instead.

6.2.3 Impact of Structural Regulation

One of the fundamentals of asynchronous logic is, that each destination node has to acknowledge consumed data to the related source node. At the ASPEAR example we can see how the structural regulation of the data flow breaks down this rule: Let us consider *PC*, *P1*, and *P2* (see Figure 6.8). Both nodes, *PC* and *P1*, obtain data from *P2* over the *Writeback Bus* path. While an acknowledge from *PC* to *P2* is mandatory, the data consumed by *P1* needs not to be acknowledged explicitly: *PC* can fire only if *P1* has consumed its output. Therefore the acknowledgement from *PC* to *P2* includes the information that *P1* has already consumed the data on the feedback path.

6.2.4 Forward Mechanism

The SPEAR core contains several forward mechanisms – in this section we will consider one of them in detail: the condition-flag forwarding.
The instruction set of the processor core provides *conditional instructions.* These are instructions where their execution depends on the fact if the condition flag of the *Processor Status Register* is set or not. The *Processor Status Register* is situated in the *Processor control unit.* The *condition flag* is set by the ALU as a result of a dedicated arithmetic operation which defines the condition. A typical instruction sequence using conditional instructions looks as follows:

```
...
CMPI r1, 2;  /* Compare register 1 with two */
JMP_CT addr; /* Condition true:jump to addr else do nothing */
...
```

The compare immediate instruction (CMPI r1,2) causes that the ALU subtracts the constant "2" from *r1* – if the result is zero then the condition flag will be set in the *processor control register* as illustrated in Figure 6.9.

The *Pipe Register 2* uses this flag to determine if a conditional instruction should be executed or replaced by a NOP. In the synchronous processor the condition flag generated by the ALU is stored only with the next active clock edge in the *Processor Status Register.* As a consequence the condition flag is available to pipe register 2 only one clock cycle later. Thus if a conditional instruction follows a compare instruction the old condition flag is going to be consumed by pipe register 2. To ensure a correct operation we have to forward the condition flag output of the ALU to the pipe register 2: Depending on the preceding instruction (compare instruction or not) the condition flag generated by the ALU or by *Processor Status Register* has to be used for evaluation purpose.

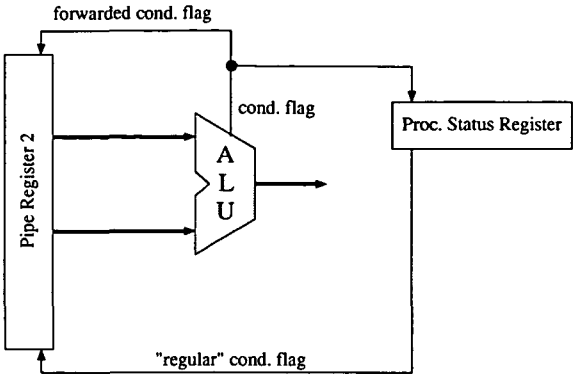


Figure 6.9. Forwarding of the Condition Flag

The asynchronous counterpart of SPEAR does not need this forwarding mechanism: The *Processor Control Unit* is a destination node of *Pipe Register 2*. Due to the control flow mechanism, the source node can fire again only if the destination node has consumed its output. In this way the result of the compare instruction (=condition flag) must be consumed first by the *Processor Control Unit* and hence the correct condition flag is available before *Pipe Register 2* can fire again.

6.3 Selecting Nodes

Due to the fact that a micro-controller can execute different instructions (the (A)SPEAR instruction set comprises 80 instruction) on the same hardware platform, one can imagine that many selecting nodes are required inside such a circuit. In this section we will focus our attention on the execution stage of the processor core, where we can identify a combination of data paths as well as a split data path.

6.3.1 MUX Structure

At the beginning of this section we motivated the replacement of the tri-state bus (= *Writeback Bus*) by a multiplexer. This is a typical example of a MUX structure which may cause a malfunction if an "eager"multiplexer is used: One input of the multiplexer is generated by *ProcCtrl_rd*. Remember that the extension module interface comprises eight registers – having only such a small number of registers the address decoding is done very quickly. Another input is generated by the ALU, which performs arithmetic operations that are much slower than a read access of a small register file. Therefore this multiplexer structure is equivalent to the example in Figure 5.44 in Section 5.2.
Obviously, we can assume that the execution of an arithmetic operation is faster than the propagation of two data waves through the execute stage of the processor core. However, as we will see in the next section an FPGA implementation of a CAL circuit increases the size of the circuit by an order of magnitude: In such a huge circuit the interconnect delay will become dominant (see [35]). Hence it is reasonable to renounce

of "eager"multiplexers at the cost of lower average performance, in order to get a delay-insensitive circuit.

6.3.2 DEMUX Structure

The entire processor core can be considered as a single big de-multiplexer: We recognize that the output of *P2* is consumed by all non-transparent nodes – however this is not required for all instructions. Quite on the contrary, there is no instruction that requires the data to be issued to **all** non-transparent nodes. As a consequence these nodes are address in a selective manner. In Section 5.2 we have shown that such a structure can be handled either by using a *synchronizer circuit* or by issuing dummy data. For tightly coupled data paths such as those found in a processor core, the dummy data approach is the favorable solution: Apart from the CAL data flow control mechanism, which can be compared with the clock signal in synchronous designs, the processor core has an "execution"control mechanism on a higher abstraction level, which is regulated by the instructions. Therefore each node recognizes due to the "execution"flow mechanism whether it is selected or not. Thus we can broadcast the output of *P2* to all nodes – if the data is not intended for a specific node, then it considers the received data as dummy data. Note that in both cases (dummy data or not) an acknowledgement must be generated.
Now we understand why *P2* has to wait until the *PC* has consumed a (dummy) jump address even if the current instruction is an arithmetic operation and the result has already been written into the register file.

6.4 Implementation Results

Having taken into account all the consideration of this section we developed the first prototype of ASPEAR. As we have already mentioned the target platform constitutes ALTERA's APEX 20KC FPGA (see Section 4). In Figure 6.10 we see the summary of the compilation process performed by *Quartus II*, the Place&Route frontend for the APEX 20KC. Compared to its synchronous counterpart, which requires ca. 2000 logic elements, ASPEAR needs ten-times as much logic elements as the synchronous SPEAR.

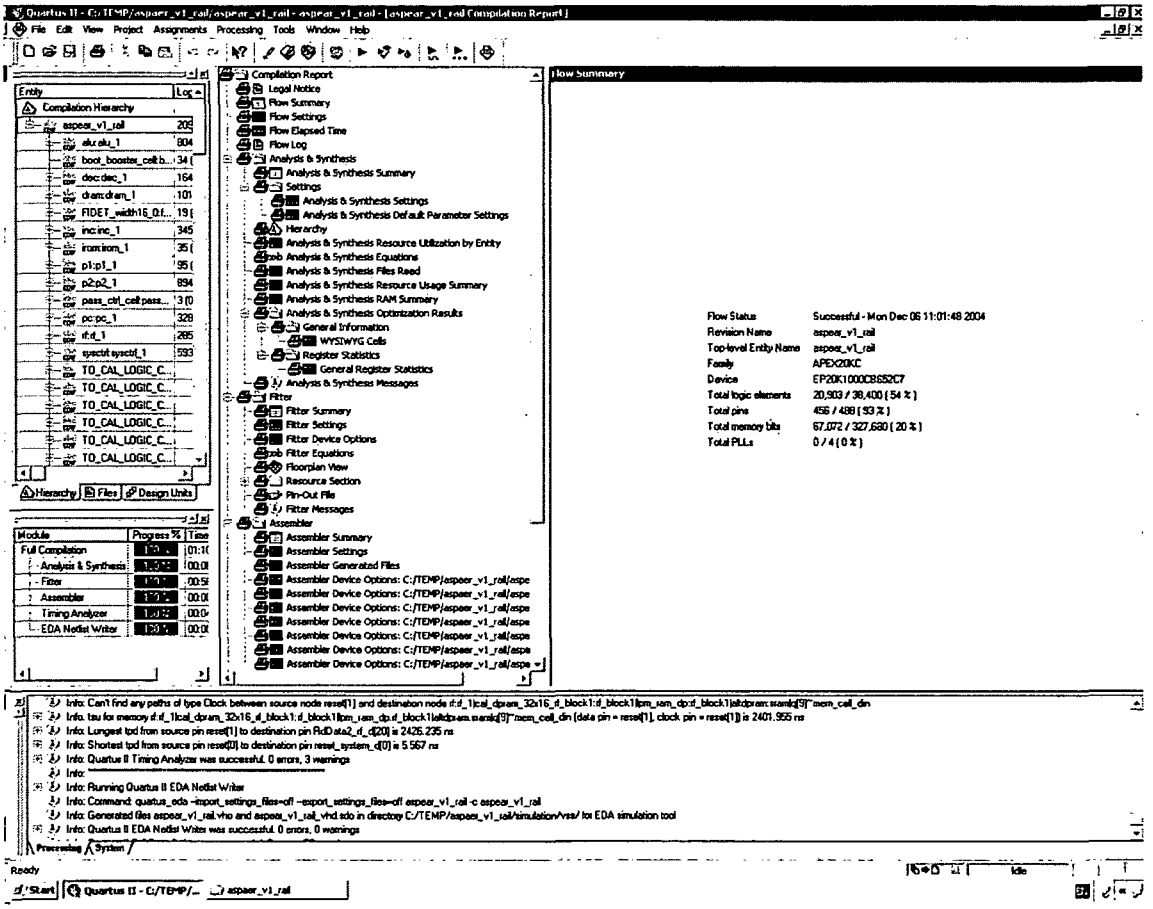


Figure 6.10. Compilation Report of ASPEAR

Figure 6.11 reports the post-layout simulation of the processor core. We recognize the switch activity of the programm counter and the pipe registers: as a consequence of the full initialization, *pipe register 2* fires first, subsequently *pipe register 1* switches, which in turn activates the program counter.
Furthermore we can determine the DEMUX behavior of *pipe register 2* as aforementioned: *pass_p2* changes its value only when all destination node of *pipe register 2* have activated their *c_done* signals.

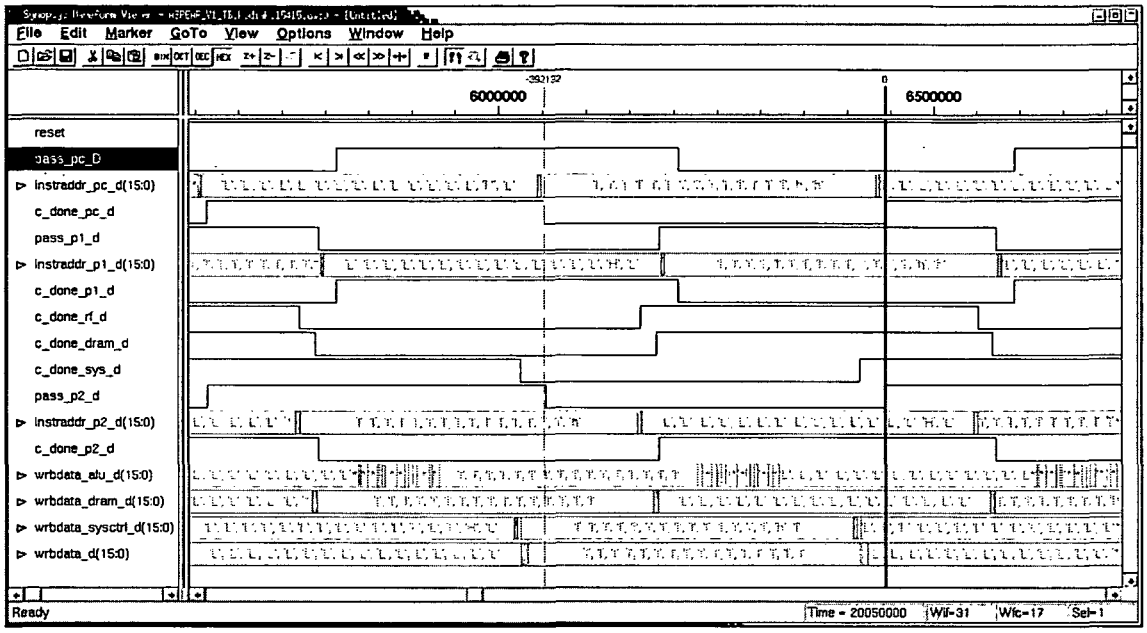


Figure 6.11. Simulation Report of ASPEAR

We can observe that the processor works with a speed comparable to a synchronous clock with 2,5 MHz. Building an asynchronous processor that is larger and still slower than its synchronous equivalent appears to be disappointing at first glance. It should be considered , however, that no optimization has taken place on the given platform (see Chapter 4). Given these suboptimal conditions, we are proud that ASPEAR works correctly, and view this as a convincing sign for the robustness of the used asynchronous approach.

Chapter 7

Conclusion and Outlook

CAL is a promising asynchronous design style, due to the fact that it can be implemented using standard design tools on the one hand and it leads to delay-insensitive circuits on gate level on the other hand. Furthermore the alternating encoding style of subsequent data wave allows to associate each bit in the circuit to a specific context, which simplifies the debugging procedure of CAL circuits. Due to the fact that the CAL approach uses a signal encoding non-linear structures such as forward/feeback path as well as selecting nodes face some problems: Either a deadlock occurs due to a permanently inconsistent input vector or unselected components loose their (phase-) synchronization because they only receive a subset all incoming data waves.
In this thesis we analyzed in detail these topics in order to contextualize the complex interactions between data flow control, which is inherent in CAL, and non-linear circuit structures. We explained how phase inverters have to be placed in the circuit to avoid deadlocks and how selecting nodes must be adopted to ensure a correctly operating circuit. In addition we have portrayed that feedback/forward paths cause a structural regulation of the data flow and thus the performance of the circuit depends on its initialization. Furthermore we illustrated that, using CAL, merge structures cannot be implemented in a delay insensitive manner.
All these findings were considered during implementation of the ASPEAR processor core. Although the asynchronous processor core cannot keep up with its synchronous counterpart, neither in terms of processing speed nor in terms of gate count, this first prototype demonstrates that validity of the presented concepts.

However, ASPEAR is intended to be used for research purpose and in this sense it stands as an ideal starting point for further optimizations and research activities:

CAL-memories: The weakest point of this prototype is constituted by the CAL-memories. These are the only parts of the processor core which are not delay insensitive. Basically, two strategies can be taken into consideration: (i) Enhanced conventional memories by storing a signature in addition to the data, which can be used to determine the validity of the current output vector. (ii) Build a full customized "CAL-memory"as presented in [50].

Basic gates: The key to improve speed and reduce the gate count are more efficient basic gate implementations – this requires a transistor level as effectuated in [21]. Another aspect which has to be investigated is if all basic gates require a memory element or if it suffice that only pipe registers check consistency?

Structural optimizations: ASPEAR is based on a direct mapping from its synchronous counterpart. Some other implementations are imaginable: all pipe registers could be removed from processor core, for instance. On the one hand this would result in longer data paths, on the other hand the additional delay resulting from the communication protocol, as well as the switching delay of the registers would be eliminated. In this way it is possible to investigate the optimal relation between the number of pipe registers and the length of data paths.

There is still a lot of work to do, however, I hope that this thesis will fire some further research activities concerning CAL at the department.

Bibliography

[1] Altera homepage. http://www.altera.com/products/devices/dev-index.jsp.

[2] Brainy encyclopedia. http://www.brainyencyclopedia.com/encyclopedia/g/gl/glossary.htm.

[3] *IEEE Standards Navigation Bar IEEE Std 1364-1995 IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language.*

[4] *ASIC System Design with VHDL: A Paradigm.* Kluwer Academic Publishers, 1990.

[5] *IEEE standard multivalue logic system for VHDL model interoperability (std_logic_1164).* 1993. IEEE Std 1164-1993.

[6] Digilab megAPEX manual - apex 20k high-end prototyping system. http://www.elca.de/Downloads/Manual Digilab megAPEX.pdf, 01 2003.

[7] Apex 20kc programmable logic device data sheet. http://www.altera.com/literature/ds/ds_apex20kc.pdf, 02 2004.

[8] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, April 1997.

[9] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming and silicon compilation. In *Proc. International Conf. Computer Design (ICCD)*, pages 150–166, Rye Brook, New York, 1988. IEEE Computer Society Press.

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

[11] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[12] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag, 1995.

[13] J.A. Brzozowski and S. Singh. Definite asynchronous sequential circuits. *IEEE Transactions on Computers*, C-17(1):18–26, January 1968.

[14] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits.* Springer-Verlag, 1995.

[15] Chris Caldwell. Graph theory glossary. http://www.utm.edu/departments/math/graph/glossary.html.

[16] Tam-Anh Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *Proc. International Conf. Computer Design (ICCD)*, pages 220–223. IEEE Computer Society Press, 1987.

[17] Ilana David, Ran Ginosar, and Michael Yoeli. An efficient implementation of boolean functions as self-timed circuits. *IEEE Transactions on Computers*, 41(1):2–11, January 1992.

[18] Ilana David, Ran Ginosar, and Michael Yoeli. Implementing sequential machines as self-timed circuits. *IEEE Transactions on Computers*, 41(1):12–17, January 1992.

[19] A. Davis and S. M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, University of Utah, Department of Computer Science, 1997.

[20] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Dept. of Computer Science, University of Utah, September 1997.

[21] Mark Dean, Ted Williams, and David Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In Carlo H. Séquin, editor, *Advanced Research in VLSI*, pages 55–70. MIT Press, 1991.

[22] M. Delvai, W. Huber, B. Rahbaran, and A. Steininger. *SPEAR - Design-Entscheidungen fr den Scalable Processor for Embedded Applications in Real-Time Environments*, 2001.

[23] Martin Delvai. Handbuch fr spear (scalable processor for embedded applications in real-time environments). Research Report 70/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[24] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor Support for Temporal Predictability - The SPEAR Design Example. In *Proc. 15th Euromicro International Conference on Real-Time Systems, Porto, Portugal*, 2003.

[25] AMD Advanced Micro Devices. www.amd.com.

[26] AMD Advanced Micro Devices. Amd powernow technology. http://www.amd.com/us-en/assets/content_type/ DownloadableAssets/Power_Now2.pdf, 2002.

[27] Webmaster Dictionary. Moore's law. http://www.webster-dictionary.org/definition/Moore's20Law.

[28] R. Dobkin, R. Ginosar, and C. P. Sotiriou. Data synchronization issues in GALS SoCs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 170–179. IEEE Computer Society Press, April 2004.

[29] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9:90–99, March 1965.

[30] Karl M. Fant and Scott A. Brandt. Null convention logic system. US patent Nr. 5,305,463, April 1994.

[31] Karl M. Fant and Scott A. Brandt. Null convention logic: A complete and consistent logic for asynchronous digital circuit synthesis. In *Proc. International Conference on Application Specific Systems, Architectures and Processors*, pages 261–273, august 1996.

[32] Farlex. The free dictionary. http://www.intel.com/products/processor/index.htm.

[33] C. Foley. Characterizing metastability. In *Advanced Research in Asynchronous Circuits and Systems*, pages 175 – 184, March 1996. Proceedings., Second International Symposium on.

[34] A. D. Friedman and P. R. Menon. Synthesis of asynchronous sequential circuits with multiple-input changes. *IEEE Transactions on Computers*, C-17(6):559–566, June 1968.

[35] Gottfried Fuchs. A superscalar 16 bit microcontroller for real-time applications. Master's thesis, Technische Universität Wien, 2003.

[36] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, and J. V. Woods. The design and evaluation of an asynchronous microprocessor. In *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, October 1994.

[37] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, April 1997.

[38] Stephen B. Furber, James D. Garside, Peter Riocreux, Steven Temple, Paul Day, Jianwei Liu, and Nigel C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, February 1999.

[39] Jim Garside. The Asynchronous Logic Homepage. `http://www.cs.man.ac.uk/amulet/async/`.

[40] Mark R. Greenstreet and Brian de Alwis. How to achieve worst-case performance. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 206–216. IEEE Computer Society Press, March 2001.

[41] K. Roy Hai Li S.Bhunia Y.Chen T.N.Vijaykumar. Deterministic clock gating for microprocessor power reduction. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003,HPCA-9 2003*, pages 113– 122. IEEE Computer Society Press, February 2003.

[42] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[43] Scott Hauck, Steven Burns, Geatano Borriello, and Carl Ebeling. An FPGA for implementing asynchronous circuits. *IEEE Design & Test of Computers*, 11(3):60–69, 1994.

[44] John L. Hennessy and David A. Patterson. *Computer Organization and Design.* Morgan Kaufmann Publisher, Inc., 1994.

[45] M. Hevery. Asynchronous circuit completion detection by current sensing. In *Twelfth Annual IEEE International ASIC/SOC Conference*, pages 322–326, 1999.

[46] Wolfgang Huber. Spezifikation der Schnittstelle zwischen Extension-Modulen und SPEAR. Technical report, Institute for Technical Computer Science, VLSI - Design, Vienna, 2001.

[47] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, March/April 1954.

[48] Intel. www.intel.com.

[49] Intel. Mobile intel pentium iii processors featuring intel speedstep technology. http://www.intel.com/mobile/resources/downloads/pdf/P3P_fn.pdf, 2001.

[50] D. L. Jackson, R. Kelly, and L. E. M. Brackenbury. Differential register bank design for self-timed differential bipolar technology. *IEE Proceedings, Circuits, Devices and Systems*, 144(5), October 1997.

[51] Martin Jankela, Wolfgang Puffitsch, and Wolfgang Huber. Towards a rapid prototyping framework for architecture exploration in embedded systems. In *Proc. Workshop on Intelligent Solutions in Embedded Systems*, pages 117–128, Granz, Austria, June 2004.

[52] Mark B. Josephs, Steven M. Nowick, and C. H. (Kees) van Berkel. Modeling and design of asynchronous circuits. *Proceedings of the IEEE*, 87(2):234–242, February 1999.

[53] Yun Y. Kenneth. Recent advances in asynchronous design methodologies. In *Asia and South Pacific Design Automation Conference 1999 (ASP-DAC'99)*, pages 253–259, jan 1999.

[54] Joep Kessels and Paul Marston. Designing asynchronous standby circuits for a low-power pager. *Proceedings of the IEEE*, 87(2):257–267, February 1999.

[55] Miloš Krstić and Eckhard Grass. New GALS technique for datapath architectures. In Jorge Juan Chico and Enrico Macii, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, volume 2799 of *Lecture Notes in Computer Science*, pages 161–170, September 2003.

[56] Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, and Alex Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 114–125. IEEE Computer Society Press, April 2000.

[57] D. W. Lloyd and J. D. Garside. A practical comparison of asynchronous design styles. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 36–45. IEEE Computer Society Press, March 2001.

[58] G. Magó. Realization methods for asynchronous sequential circuits. *IEEE Transactions on Computers*, C-20(3):290–297, March 1971.

[59] K. Maheswaran. Implementing self-timed circuits in field programmable gate arrays. Master's thesis, University of California, Davis, 1994.

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

[61] Alain J. Martin. Formal program transformations for VLSI circuit synthesis. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, UT Year of Programming Series, pages 59–80. Addison-Wesley, 1989.

[62] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[63] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278. MIT Press, 1990.

[64] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The first asynchronous microprocessor: the test results. *Computer Architecture News*, 17(4):95–110, June 1989.

[65] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Pénzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[66] Alain J. Martin, Mika Nyström, Paul Pénzes, and Catherine Wong. Speed and energy performance of an asynchronous MIPS R3000 microprocessor. Technical Report CSTR:2001.012, California Institute of Technology, 2001.

[67] Doug Matzke. Will physical scalability sabotage performance gains? *Computer*, 30(9):37–39, 1997.

[68] Anthony J. McAuley. Dynamic asynchronous logic for high-speed CMOS systems. *IEEE Journal of Solid-State Circuits*, 27(3):382–388, March 1992.

[69] Anthony J. McAuley. Four state asynchronous architectures. *IEEE Transactions on Computers*, 41(2):129–142, February 1992.

[70] John McCardle and Dr. David Chester. Measuring an asynchronous processor's power and noise. In *Synopsys Users Group Boston*, 2001.

[71] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[72] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[73] G.E. Moore. No exponential is forever: but "forever"can be delayed! [semiconductor industry]. In *Solid-State Circuits Conference, 2003*, volume 1, pages 20–23, 2003.

[74] Gordon E. Moore. The experts look ahead: Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[75] David E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1962.

[76] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[77] Chris Myers. *Asynchronous Circuit Design.* John Wiley & Sons, 2001.

[78] Chris J. Myers, Wendy Belluomini, Kip Killpack, Eric Mercer, Eric Peskin, and Hao Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, February 2001.

[79] Christian Dalsgaard Nielsen, Jørgen Staunstrup, and Simon Jones. A delay-insensitive neural network engine. In Will R. Moore, editor, *Proceedings of the Workshop on VLSI for Neural Networks*, pages 367–376, September 1991.

[80] L. S. Nielsen, C. Niessen, J. Sparsø, and C. H. van Berkel. Low-power operation using self-timed and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391–397, December 1994.

[81] L. S. Nielsen and J. Sparsø. An 85μW asynchronous filter-bank for a digital hearing aid. In *International Solid State Circuits Conference*, February 1998.

[82] Lars S. Nielsen and Jens Sparsø. Designing asynchronous circuits for low-power: An IFIR filter bank for a digital hearing aid. *Proceedings of the IEEE*, 87(2):268–281, February 1999.

[83] M. Olivieri. Translating occam constructs into delay-insensitive circuits: a trace theory-based proof. Technical Report CPSI92-1, Dept. of Biophys. and Electronic Eng., Univ. of Genoa, Italy, 1992.

[84] M. Olivieri. Design of synchronous and asynchronous variable-latency pipelined multipliers. *IEEE Transactions on VLSI Systems*, 9(2), May 2001.

[85] Recep O. Ozdag, Montek Singh, Peter A. Beerel, and Steven M. Nowick. High-speed non-linear asynchronous pipelines. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1000–10007, March 2002.

[86] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low-noise configurable self-timed DSP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 32–42, 1998.

[87] R. E. Payne. Self-timed FPGA systems. In W. Moore and W. Luk, editors, *Fifth International workshop on Field Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 21–35, 1995.

[88] Ad Peeters. The 'Asynchronous' Bibliography Homepage. http://www.win.tue.nl/async-bib/async.html.

[89] Ad Peeters. The 'Asynchronous' Bibliography (BibTeX) database file `async.bib`. http://www.win.tue.nl/async-bib/doc/async.bib. Corresponding e-mail address: `async-bib@win.tue.nl`.

[90] Christian Piguet. Logic synthesis of race-free asynchronous CMOS circuits. *IEEE Journal of Solid-State Circuits*, 26(3):371–380, March 1991.

[91] Christian Pucher. Algorithmen auf graphen. Master's thesis, Technische Universität Wien, 2004.

[92] Wolfgang Puffitsch and Wolfgang Huber. Porting the GNU Compiler Collection to the SPEAR microprocessor. Research Report 24/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[93] P. Puschner and A. Burns. Writing temporally predictable code. In *Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems*, San Diego, California, USA, Jan. 2002.

[94] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Tech. Rep. 120, Massachusetts Inst. of Tech., February 1974.

[95] R.B. Reese and S.B. Sikandar-Gani. Control versus compute power within a LEDR-style self-timed multiplier with bypass path. In *Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on*, volume 2, pages II–302–II–305, 2002.

[96] Robert B. Reese, Mitch A. Thornton, and Cherrice Traver. Arithmetic logic circuits using self-timed bit level dataflow and early evaluation. In *Proc. International Conf. Computer Design (ICCD)*, pages 18–23, November 2001.

[97] Charles L. Seitz. Asynchronous machines exhibiting concurrency, 1970. Record of the Project MAC Concurrent Parallel Computation.

[98] International SEMATECH. International technology roadmap for semiconductors, 2003 edition. http://public.itrs.net/Files/2003ITRS/Home2003.htm, 2003.

[99] N. Shintel and M. Yoeli. Synthesis of modular networks from Petri-net specifications. Technical Report 743, Dept. Comp. Science, Technion, Haifa, Israel, 1992.

[100] V. W. Y. Sit, C. S. Choy, and C. F. Chan. Use of current sensing technique in designing asynchronous static RAM for self-timed systems. *Electronics Letters*, 33(8):667–668, 1997.

[101] Vincent Wing-Yun Sit, Chiu-Sing Choy, and Cheong-Fat Chan. A four-phase hand-shaking asynchronous static RAM design for self-timed systems. *IEEE Journal of Solid-State Circuits*, 34(1):90–96, January 1999.

[102] R. L. Sites. *Alpha Architecture Reference Manual.* Digital Equipment Corporation, 1992.

[103] Michael John Sebastian Smith. *Application-specific integrated circuits.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[104] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Kluwer Academic Publishers, 2001.

[105] K. Stevens. *Private communication*, September 2000.

[106] Marco Storto and Roberto Saletti. Time-multiplexed dual-rail protocol for low-power delay-insensitive asynchronous communication. In Anne-Marie Trullemans-Anckaert and Jens Sparsø, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 127–136, October 1998.

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

[108] Hiroaki Terada, Souichi Miyata, and Makoto Iwata. DDMP's: Self-timed super-pipelined data-driven multimedia processors. *Proceedings of the IEEE*, 87(2):282–296, February 1999.

[109] G. K. Theodoropoulos, G. K. Tsakogiannis, and J. V. Woods. Occam: an asynchronous hardware description language? In *Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology*, pages 249–256, September 1997.

[110] A.K. Uht. Going beyond worst-case specs with teatime. *Computer*, 37:51–56, March 2003.

[111] Stephen H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, 20(12):1437–1444, December 1971.

[112] Hans van Gageldonk, Kees van Berkel, Ad Peeters, Daniel Baumann, Daniel Gloor, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. In *4th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '98)*, 1998.

[113] Victor I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[114] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.

[115] Tom Verhoeff. Characterizations of delay-insensitive communication protocols. Computing Science Notes 89/06, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1989.

[116] P. Vingron. Coherent design of asynchronous circuits. *IEE Proceedings, Computers and Digital Techniques*, 130(6):190–202, 1983.

[117] W.Huber. *Design of an Asynchronous Processor Based on Code Alternation Logic – Explorations of Delay Insensitivity.* PhD thesis, Vienna University of Technology, 2005.

[118] Wikipedia. The free encyclopedia. http://en.wikipedia.org/wiki/Gray_coding.

[119] Ted Williams, Niteen Patkar, and Gene Shen. SPARC64: A 64-b 64-active-instruction out-of-order-execution MCM processor. *IEEE Journal of Solid-State Circuits*, 30(11):1215–1226, November 1995.

[120] Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.

[121] J.V. Woods, P.Day, S. B. Furber, J.D. Garside, N. C. Paver, and S. Temple. Amulet1: A micropipelined arm. In *IEEE Computer Conference*, pages 476–485, 1994.

[122] Sheng-Fu Wu and P. David Fisher. Automating the design of asynchronous sequential logic circuits. *IEEE Journal of Solid-State Circuits*, 26(3):364–370, March 1991.

[123] F. Xia, A. Yakovlev, D. Shang, A. Bystrov, A. Koelmans, and D. J. Kinniment. Asynchronous communication mechanisms using self-timed circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 150–159. IEEE Computer Society Press, April 2000.

[124] Michael Yoeli. Examples of LOTOS-based verification of asynchronous circuits. Technical Report CS-2001-08, Dept. Comp. Science, Technion, Haifa, Israel, 2001.

[125] M.D.Ercegovac Zhijun Huang. Effect of wire delay on the design of prefix adders in deep-submicron technology. In *Conference on Signals, Systems and Computers, 2000*, okt-nov 2000.

Curriculum Vitae

Martin Delvai
Theodor Kramer Straße 8/2/81
1220 Vienna

Personal Data

Date of Birth:	February, 15, 1974
Place of Birth:	Bozen
Citizenship:	Italy

1980 – 1985	Volksschule *(elementary school)* Seis am Schlern
1985 – 1988	Mittelschule "Leo Santifaller" *(secondary school)* Kastelruth
1988 – 1993	Gewerbeoberschule "Max Valier" Fachrichtung Elektronik *(polytechnic - Electrical Engineering Department)* Bozen
1993 – 2000	Technische Universität Wien – Elektrotechnik *(Vienna University of Technology – Electrical Engineering)* Academic degree: Diplomingenieur *(comparable to Master of Science)*
1997 – 1998	Auslandsjahr an der TU Darmstadt *Exchange student: Darmstadt University of Technology*
June – October 1998	Summer job at Fa. Telnet: Software Developer
1999 – 2000	Working student at Fa. Widder: Network Support
2001 – 2004	Research assistant at TU Vienna Embedded Computing Systems Group