Behavioral Synthesis of Asynchronous Circuits

Ph.D. thesis
by
Sune Fallgaard Nielsen

Computer Science and Technology
Informatics and Mathematical Modelling
Technical University of Denmark

This dissertation is submitted to Informatics and Mathematical Modeling at the Technical University of Denmark in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

The work has been supervised by Associate Professor Jens Sparsø and Professor Jan Madsen.

Kgs. Lyngby, December 31, 2004

Sune Fallgaard Nielsen

Resumé

Denne afhandling presenterer en metode for behavioral syntese af asynkrone kredsløb. Målet er at tilvejebringe et syntese flow, som udnytter og overfører metoder fra synkrone kredsløb til asynkrone kredsløb. Ideen er at flytte den synkrone behavioral syntese abstration ind i det asynkrone handshake domæne ved hjælp af en beregnings model, som ligner den synkrone datavej og kontrolenheds struktur, men som er fuldstændig asynkron.

Denne model indeholder muligheden for at isolerer enkelte eller alle beregningselementer ved at låse deres respektive inputs og outputs når beregningselementer er inaktivt. Dette reducerer unødvendig skifteaktivitet i de enkelte beregningselementer og derved energiforbruget af hele kredsløbet. En samling af behavioral syntese algoritmer er blevet udviklet, som tillader designeren at foretage design space exploration bestemt af både power- og udførelsestids-krav. Datavej og kontrol arkitekturen bliver derefter udtrykt i Balsa-sproget, og syntaks styret oversættelse anvendes til at konstrurere det tilhørende asynkrone handshake kredsløb (og evt. endeligt et layout).

Abstract

This thesis presents a method for behavioral synthesis of asynchronous circuits, which aims at providing a synthesis flow which uses and tranfers methods from synchronous circuits to asynchronous circuits. We move the synchronous behavioral synthesis abstraction into the asynchronous handshake domain by introducing a computation model, which resembles the synchronous datapath and control architecture, but which is completely asynchronous. The model contains the possibility for isolating some or all of the functional units by locking their respective inputs and outputs while the functional unit is idle. This reduces unnecessary switching activity in the individual functional units and therefore the energy consumption of the entire circuit. A collection of behavioral synthesis algoritms have been developed allowing the designer to perform time and power constrained design space exploration. The datapath and control architecture is then expressed in the Balsa-language, and using syntax directed compilation a corresponding handshake circuit implementation (and eventually a layout) is produced.

Acknowledgments

Many people have helped me to arrive at this point, all of whom I am grateful to.

Contents

Chapter 1

Introduction

Today, a wide range of dedicated real-time applications are emerging. Examples of these are the next generation of mobile phones, smart-cards and more futureristic applications as e-identification, e-payment, e-key systems etc. For such portable wire-less applications power is a limited resource because of restrictions in battery size or because power is extracted from the environment (light, magnetic fields or heat etc.). Furthermore, to meet the extreme size and weight requirements the entire system (input/output transducers, analog circuitry, futuristic-circuitry, power supply and the digital system, consisting of digital hardware and software) is implemented onto one single chip ("System on Chip").

The focus of this research is the hardware part of the digital system, which operates under the following difficult characteristics:

Data Processing The applications are reactive in nature with data arriving in bursts with long periods of waiting. In-between bursts ultra low-power operation is required, while during bursts heavy computation, such as encryption for secure data transmission, is required.

Response Time For some applications the time to respond to an external event is crucial as otherwise data will be irrevocably lost, requiring a close to zero transition time from sleep mode into full-speed operation.

Power Supply For battery-less applications external power is provided spuriously by the environment and stored internally on large storage capacitors leading to a very limited power supply often of poor quality.

Noise Level The presence of on-chip analog and RF-circuitry sets severe restrictions for the electric-noise and electromagnetic-emission of the digital circuit such as not to disrupt input/output-interfacing or RF-communication.

Asynchronous design offers several advantages, compared to synchronous design, for the design of these intelligent circuits. The asynchronous design methodology specifically targets low-power operation (power is only used when processing) and the self-timed nature leads to an immediate response time. Furthermore asynchronous circuits are inherently insensitive (and thus robust) to variations in temperature, process parameters and supply voltage. The latter can be used advantageously since, if the circuit has access to external power, the supply voltage can be decreased allowing for ultra low-power operation. Finally, the asynchronous nature of the switching activity causes the electromagnetic and electric noise contributions to evenly distribute across the frequency spectrum (equivalent to white noise). This reduces spikes in the spectrum down to a level which allows co-existence with analog and RF-circuitry. Typically only critical subparts (with respect to operating characteristics) of the digital system will be implemented asynchronously and the remaining part synchronously.

Currently, the lack of synthesis methods and tools which are capable of directly synthesizing a working asynchronous circuit from a high-level specification makes the design of large systems a tedious effort involving more design work than designing a corresponding synchronous circuit. The majority of existing synthesis tools in this area are low-level and dedicated to the generation of control circuitry [24, 40, 71, 86, 92]. A few high-level synthesis tools exist, among those the Tangram silicon compiler developed by Philips Research Labs and the somewhat similar public domain version BALSA from Manchester University. These tools use special asynchronous hardware description languages dedicated to asynchronous design, that does not fit well into existing VHDL/SystemC based design flows and CAD-tools. Furthermore, the supported synthesis process, syntax-directed compilation, is characterized by a one-to-one correspondence between specification and implementation.

Let us begin by looking into the current status of synthesis flows of synchronous and asynchronous circuits as illustrated by Figure 1.1. Synthesis of synchronous circuits, which is illustrated in the left column of Figure 1.1, has succeeded in raising the level of abstraction to that of specifying circuits at the behavioral level. From a behavioral description in a language like VHDL, Verilog or System-C some intermediate representation is extracted – often a control data flow graph (CDFG). From the CDFG the classic synthesis tasks [67] of scheduling, allocation, and binding is performed resulting in a RTL level circuit description which is then synthesized into gate-level circuits and eventually a layout.

Synthesis of asynchronous circuits is illustrated in the right column of Figure 1.1. It is less mature and several somewhat different approaches is being pursued. The most influential of the available synthesis tools fall in two categories: (i) synthesis of large-scale RTL level circuits based on syntax-directed compilation from CSP-like languages: Tangram [11, 100], OCCAM [17], Balsa [8], ACK [59] and TAST [85], and (ii) synthesis of small-scale sequential control circuits [26, 41]. The tools that perform syntax directed compilation target a library of so-called handshake components.

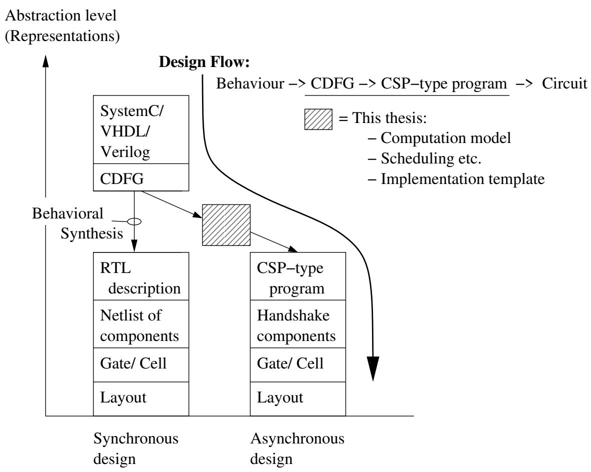


Figure 1.1: Existing synchronous and asynchronous design flows and the design flow addressed in this thesis.

The handshake components can be designed using in principle any of the sequential control circuit synthesis tools. The syntax-directed compilation approach is radically different from the behavioral synthesis flow used by designers of synchronous circuits; the compiler merely performs a one-to-one mapping of the program text into a corresponding circuit structure. Although syntax-directed compilation does allow the designer to work at a relatively high level it does not provide any optimizations; "what you program is what you get". In some situations this can be considered an advantage but in general it puts more burden on the designer: exploring alternative implementations requires actually programming these, whereas in a traditional synchronous synthesis flow, the designer can quickly and easily experiment with different constraints and goals and in this way create alternative implementations from the same program text.

It is interesting to note that the internal representation of circuit behavior used in synchronous behavioral synthesis is actually based on an asynchronous model of a control dataflow graph (CDFG), i.e., a dependency graph expressing the control- and data-flow of the application. This naturally raises the question: Is it possible to apply the transformations and optimizations used in synchronous synthesis for asynchronous design as well?

1.1 From synchronous to asynchronous behavioral synthesis

A central idea in this thesis is to construct a computation model which allows us to use the transformations and optimizations used in synchronous synthesis directly in

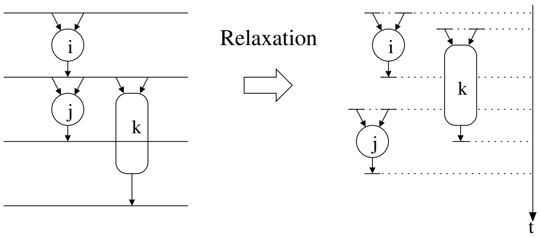


Figure 1.2: Relaxing synchronous synthesis (left) into the asynchronous handshake domain (right).

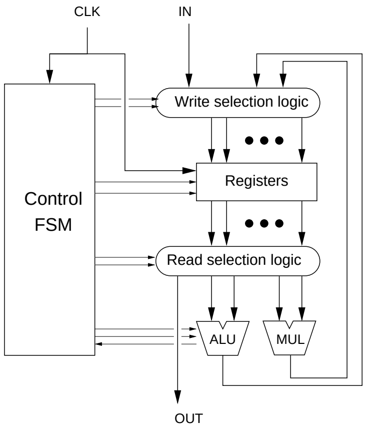


Figure 1.3: The synchronous computation model.

asynchronous design, without introducing any restrictions and at the same time use the transformations and optimizations developed for continuous time in one and the same model.

The target for synchronous behavioral synthesis is a hardware architecture consisting of a datapath which is able to perform a set of operations, and a controller which controls the execution sequence of these operations in order to perform a given application, as shown in Figure 1.3 A key issue in behavioral synthesis is to reuse hardware resources for the different operations in order to minimize area, and to explore possible parallelism by executing several hardware resources concurrently in order to increase performance.

All the traditional techniques of behavioral synthesis: Scheduling, Allocation and Binding are in synchronous circuits centered around a central synchronization event, determined by the global clock. This synchronization event determines (i) the beginning for executing an operation (ii) writing the result of an operation.

If we make these synchronization events local and controlled by the controller, we can create a hardware architecture consisting of a datapath and a controller, as

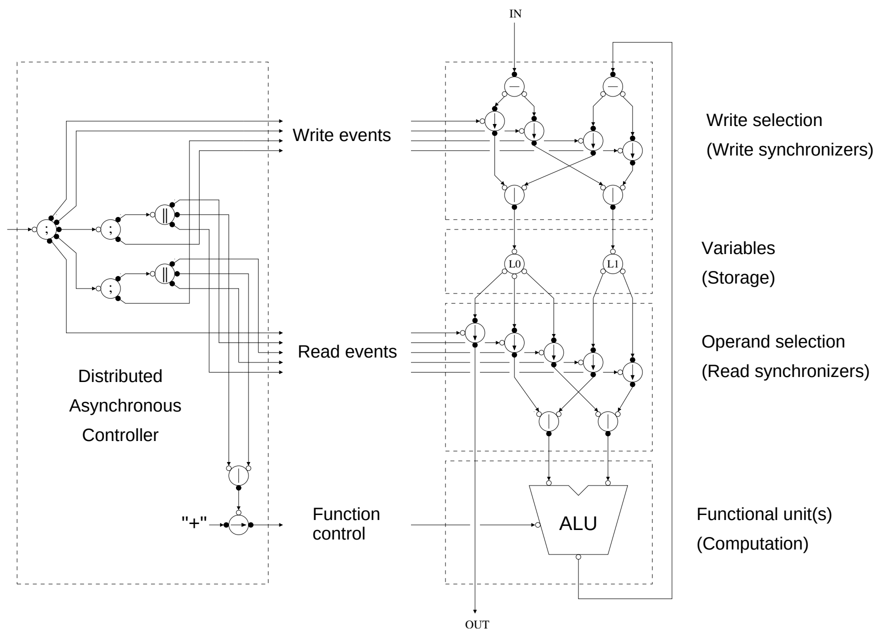


Figure 1.4: Computation model in the asynchronous handshake domain, where the labeling refers to the role the handshake components play in our model.

shown in Figure 1.4. It resembles the synchronous architecture but it is completely asynchronous. This computation model relaxes the strict ordering of the synchronous circuit and the synchronous schedule 1.2 (left) into the continuous time domain, the schedule for the asynchronous circuit 1.2 (right).

This idea allows us to use any of, but not restricted to, the many synchronous behavioral synthesis techniques to obtain a hardware architecture (datapath and controller) and then to implement this architecture using asynchronous circuit techniques.

In our work we use Balsa as a back-end. The datapath and control parts obtained from the front-end behavioral synthesis are described using a set of Balsa templates and then synthesized into handshake components and ultimately into a layout. In this way we take advantage of the fact that Balsa performs a one-to-one mapping thus allowing us to express the intended implementation at a relatively high level.

The parallism in CSP, and CSP-like languages, are centered around a parallel operator, that allows the computation to fork into parallel operations. However the construct also require all of these parallel operations to finish at the same time or have to wait until the slowest operation finishes. Therefore no new operations can begin, thus limiting the schedules that can be implemented. The implementation templates presented in this thesis is not restricted by this limitation. We utillize the CSP language constructs in an unconventional way, such that any continous schedule can be implemented.

Using this synthesis flow we have produced layouts for a couple of benchmarks and we report on the area, speed and power figures for these circuits. By building on top of syntax-directed compilation, our synthesis approach works entirely in the domain of handshake channels and handshake components. This has a number of significant implications: Firstly it enables the use of a synthesis flow which is surprisingly similar to that used in synchronous design tools, and secondly it avoids altogether the complex problem of specifying and synthesizing a controller. Our work is not in any way restricted to the use of Balsa or other syntax-directed methods, the used approach serves as a practical demonstration of how to use the developed methods and techniques.

For the behavioral synthesis part we have developed the following algorithm suite:

(i) Power aware synchronous synthesis algorithm. This algorithm is a clique heuristic algorithm operating with a time and maximum power per time constraint. This is useful for applications having a power limit e.g. given by the maximum power delivered by a solar panel.

(ii) Evolutionary synchronous synthesis algorithm and a simulated annealing synchronous synthesis algorithm. These are meta-heuristic algorithms operating with a maximum time constraint.

(iii) Simulated Annealing task level algorithm for handling the conditional parts of the CDFG. This last algorithm has not been implemented but the method is outlined.

These algorithms all operate in discrete time using time-slots. After the final schedule has been obtained it is relaxed into an asynchronous schedule, keeping the order of execution events as a relative ordering.

The contribution of this thesis is the addition of behavioral synthesis to asynchronous circuit design in the form of automatic resource sharing and constraint based design space exploration. In particular our contributions are: (1) an abstract event based computation model, (2) synthesis algorithms for scheduling, allocation and binding and (3) target implementation specifications. The thesis publications are [74, 75, 93].

1.2 Thesis outline and readers guide

This thesis is organized as follows:

Chapter 1 Introduction Introduces this work, presents our contributions and shows this outline of the thesis.

Chapter 2 Background Briefly introduces the ideas behind behavioral synthesis, CDFGs and asynchronous circuits.

Chapter 3 Related Work Gives a survey of related work.

Chapter 4 Behavioral Synthesis for Asynchronous Circuits Presents the concept which allows us to adapt the techniques from synchronous behavioral synthesis into behavioral synthesis of asynchronous design and describes details of datapath design.

Chapter 5 Implementation in Balsa The use of the Balsa-language to generate our circuits is presented in this chapter.

Chapter 6 Algorithms for Behavioral Synthesis The algorithms developed for behavioral synthesis used to generate the circuits are presented in this chapter.

Chapter 7 Results The area, speed and power figures for our layouts are presented and discussed.

Chapter 8 Conclusion contains the conclusion of the thesis and presents directions for future work.

As a reading guide, the reader who is familiar with asynchronous circuit design and behavioral synthesis and not interested in related work can skip *chapter 2 Background* and *chapter 3 Related Work*, and proceed directly to *chapters 4 Behavioral Synthesis for Asynchronous Circuits, 5 Implementation in Balsa and 6 Algorithms for Behavioral Synthesis* which presents the main contribution of this thesis. More specifically the underlying concepts of this work are introduced in *4 Behavioral Synthesis for Asynchronous Circuits*. The circuit implementation details and Balsa-templates used to design the asynchronous circuits in the result section are presented in *chapter 5 Implementation in Balsa*. For the reader with an algorithmic interest *chapter 6 Algorithms for Behavioral Synthesis* presents the behavioral synthesis algorithms developed in this research. Finally, the reader is encouraged to read *chapter 7 Results* which explains and discusses the results.

CHAPTER 2

Background

This thesis brings together the domains of both behavioral synthesis and asynchronous circuit design. In order to be able to better understand the work presented in this thesis, this chapter will give an introduction to some of the concepts and ideas of these domains. The reader should not consider this to be a complete reference, nor to be a tutorial.

2.1 Synthesis flow and CDFG format

A CDFG captures only the control and data dependencies that are inherent in the computation. In this way it is not biased towards a certain implementation.

In this section we introduce the CDFG format and an example CDFG which will be used throughout in this thesis to illustrate the synthesis flow. The focus of the thesis is on the synthesis of asynchronous circuitry *given* a CDFG. The process of extracting the CDFG from a behavioral specification in some hardware description language is well understood. It is an integral part of existing synchronous synthesis systems, and it is not addressed in this thesis.

To illustrate the source code for our running example we will use the Balsa-language [7, 8, 6], augmented with a multiplication operator, as the Balsa language does not yet include a multiplication operator. The aim in this thesis is not to advocate the use of Balsa, it should merely be seen as an illustration and in principle most hardware description languages could be used. For asynchronous circuit design it is convenient if the language includes channel communication primitives and statement level concurrency, and it is encouraging to see that such features are being included,

```
import [balsa.types.basic]

type word is 16 bits

procedure example(input X0,X1,X2:word;
                  output Y0,Y1:word) is
variable x0,x1,x2,y1,y0:word
constant a0= 255
constant a1= 255
constant a2= 255
constant a3= 255
begin
  loop
    X0->x0 || X1->x1 || X2->x2 ;
    y0 := (((a0+x0)+(x0*x1)) - a1 as word) ||
    if  x1>a2 then
      y1 := (a3*(x1+x2) as word)
    else
      y1:= (x1-x2 as word)
    end ;
    Y0<-y0 || Y1<-y1
  end
end
```

Figure 2.1: An example Balsa description.

or at least proposed for inclusion in, such languages as System-C and System-Verilog and an additional package for adding such features to System-C is proposed in [13].

The intended synthesis flow involves the following steps: From the Balsa code the CDFG is exacted. The CDFG is then subject to the synthesis steps explained in this thesis and the resulting circuit structure (datapath and control) is expressed as a Balsa program. The final step of the synthesis flow is then to compile the Balsa program into a netlist of handshake components and to produce a standard cell implementation.

Figure 2.1 shows our example asynchronous component specified in Balsa and Figure 2.2 shows the corresponding CDFG which will serve as the running example in this paper. The elements of the CDFG and the structure are explained in the following. The CDFG is a 1-bounded colored Petri net – the colors representing data values. The edges in the CDFG contain places (like in a STG) and the nodes are Petri net transitions. A node can be an operator or can represent conditional sequencing as the example CDFG shows. For a more formal definition the reader is referred to [96, 33].

The basic elements in our CDFG are shown in Figure 2.3 and are as follows:

nodes Essential nodes represent atomic computations e.g. arithmetic operations as

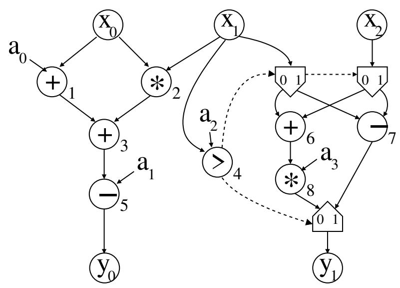


Figure 2.2: The Control Data Flow Graph for our example.

addition. For firing a node, all inputs arcs need to have a data-token present. The i designates the operation the node performs and all nodes have a numbering j. This operation could also be the loading of data in and out of the circuit, in which case the name of the input/output is written inside the node. These nodes are called input/output nodes.

arcs Represents the essential data dependencies which exist with in the computation or algorithm. The dotted arc is used to signify control arcs. There is no semantic difference between a data and control arc.

There is a set of special nodes which needs explanation:

Control nodes The mux and demux nodes are used to route data-tokens around in the CDFG. The mux node need a data-token on the control arc and then a data-token on the selected input arc to fire. The demux node only fires a data-token on the selected output.

Body The body can be replaced by another CDFG and is not a fundamental component, rather it illustrates the hierarchical nature of the CDFG format. The input and output arcs of the CDFG are required to fit with the input and output arcs to the Body node.

Using these fundamental nodes a sufficient set of algorithmic structures can be represented using the CDFGs. Figure 2.4 shows a set of basic algorithmic structures found in most languages and their corresponding CDFGs. Using these structures, the definition of the CDFGs nodes and our Balsa example in Figure 2.1 it is straightforward to arrive at the CDFG in Figure 2.2.

2.2 Behavioral synthesis

Behavioral synthesis is a refinement process in which a behavioral description of an algorithm is converted into a structural description, fulfilling a set of design constraints, and preserving the behavior of the algorithm [87, 67]. Each component

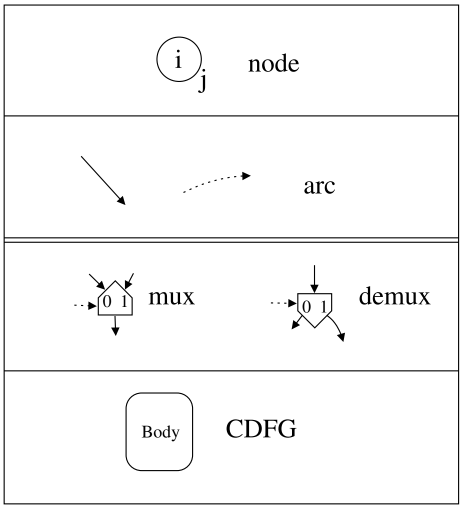


Figure 2.3: A minimum and, for most cases, sufficient set of Control Data Flow Graph elements.

in the structural description is in turn defined by its own (lower-level) behavioral description, for which a mapping to silicon hardware exists. The purpose of behavioral synthesis is two-fold: (i) Automate tedious parts of the design process and thus improve the turnaround time. (ii) To perform design space exploration.

Automating tedious parts of the design process is becoming increasingly important as designs increase in size and complexity, and the time alloted to construct the design becomes ever more tighter. Specifying the description of an algorithm at a higher level of abstraction allows a designer to focus on implementing an improved algorithm. It is well-known, that work put to use at a high-level of abstraction has a larger impact on the resulting performance characteristics, than work put to use at a lower-level of abstraction. Furthermore, the designer avoids spending time on details of the implementation e.g. transistor sizing, which of course has an impact on the performance but usually an order of magnitude less than improving the algorithm.

Design space exploration is also becoming increasingly important as modern systems are moving into System-on-Chip platforms where the design becomes part of a greater whole and thus needs to fit into certain specifications. This might mean that the maximal speed of the circuit is required if our circuit is part of the critical path of an entire system. But it might also be that requirements are low and thus there is no need to develop a large high-speed circuit.

The output from a high-level synthesis system usually consists of a datapath structure at the register-transfer level (RTL) or an equivalent description language, and a specification of a finite state machine to control the datapath. In our case we will use the Balsa language which will translate into a set of asynchronous handshake components for both the datapath and the ASFM. At the RT level or equivalent

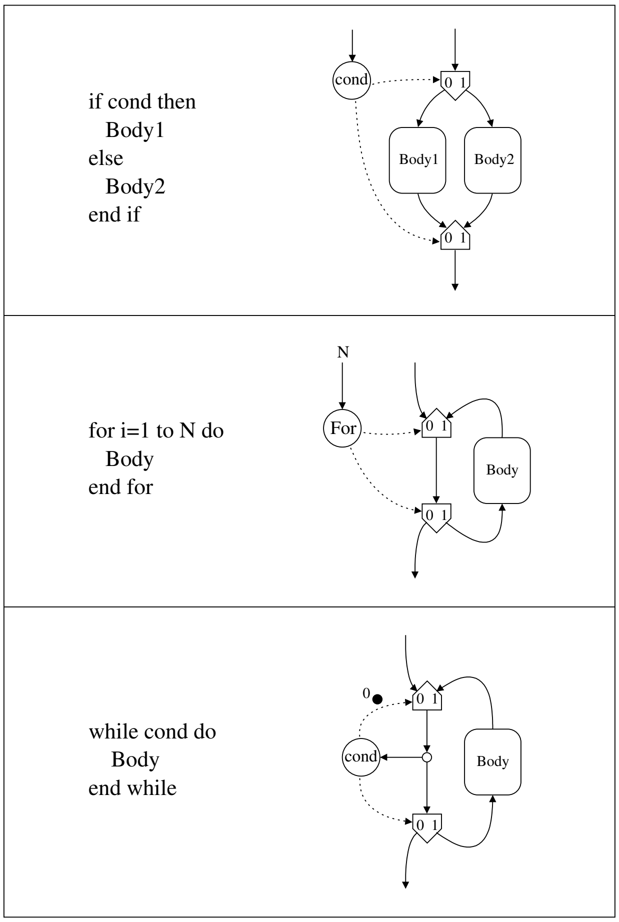


Figure 2.4: Algorithm statements and corresponding CDFG structures.

level, a datapath is composed of a computational part (functional units e.g. ALUs, multipliers, and shifters etc.), storage units (registers, latches and memories) and interconnection units (e.g. busses, multiplexors and demultiplexors).

As previously discussed the first step is to extract a CDFG from the behavioral algorithm, part of this involves a series of compiler-like optimizations as code motion, dead code elimination, constant propagation, common subexpression elimination, and loop unrolling. Following this comes the core synthesis refinement process, of which there are two classes:

Resource constrained behavioral synthesis Here the goal is to find the fastest circuit given a set of resource constraints either in the form of a maximum allowable area for the circuit or a detailed description of the maximal number and types of functional units and the maximum memory available to the circuit.

Time constrained behavioral synthesis Here the goal is to find the smallest circuit (computational area and memory) given a maximum execution time constraint.

In addition to these there is the power constraint which comes into play by adding to the two other constraints, reducing the solution space. In this thesis we will consider time and power constrained behavioral synthesis. The applications our research targets are performance-intensive parts of an algorithm which therefore require implementation in hardware, thus the constraints are often in the form of a time requirement or a dataprocessing frequency to which the smallest circuit needs to be found. However there is nothing preventing us from implementing resource and power constrained behavioral synthesis.

In general we distinguish between behavioral synthesis in continuous time and behavioral synthesis in discrete time, but in general both approaches involve the same three basic elements:

Scheduling The operations in the CDFG need a start time. For continuous time this is an absolute time or a relative ordering of operations. In discrete time this denotes the start time-slot.

Allocation A set of functional units needs to be allocated. The functional units are the machines on which the operations are executed.

Assignment The operations need to be bound to a specific machine to avoid conflicts for parallel operations.

These elements are believed to be NP-hard problems and thus in general require heuristic approaches to find solutions. These three tasks are closely interrelated and should be solved simultaneously to arrive at an optimal solution. All the behavioral synthesis algorithms presented in this thesis do this. There are in principle three approaches to solve these problems:

Integer Linear Programming (ILP) formulations which solve the problem for optimality but is only applicable for small problems.

Heuristic methods that come in two flavors: constructive approaches and iterative refinement. There are many approaches for constructive scheduling, differing with regard to the selection criteria used to schedule the next operation. Heuristic approaches run efficiently for large designs, but does not produce optimal circuits.

Meta-heuristic Algorithms which are capable of solving large ILP problems effectively, although heuristically.

Besides these fundamental elements of behavioral synthesis there are elements that involve finding the minimum amount of memory for the specific schedule, allocation

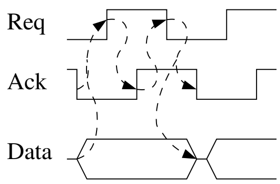


Figure 2.5: Four phase bundled data push handshake protocol.

and assignment, as well as finding the best routing (the minimal set of multiplexing) of data between the functional units. All of these elements of behavioral synthesis and datapath synthesis will be elaborated further in their respective chapters.

2.2.1 ASAP and ALAP

Now, before trying to minimize the silicon area, we first want to know if, given the CDFG and the time constraint T, a feasible schedule can be constructed at all ? (using unlimited silicon area). Fortunately, there is a polynomial algorithm, $\mathcal{O}(n^2)$, which can give us that answer:

ASAP (As Soon As Possible) Augment the CDFG with a source node which has directed arcs to all the input nodes. Set $S_{source} = 0$ for the source node. Then finding the S_i for all other nodes v_i (σ_i) becomes a matter of finding the longest path from the source to that node. (Using the fastest FU for the job).

If $S_{target} \leq T$ for the target node, it is possible to construct a feasible schedule. Furthermore S_i is the earliest time an operator σ_i can be scheduled (again allowing for unlimited silicon area). The same algorithm can be applied "backwards":

ALAP (As Late As Possible) Augment the CDFG with a sink node which has directed arcs from all the output nodes. Set $L_{target} = T$ for the target node. Then finding the L_i for all other nodes v_i (σ_i) becomes a matter of finding the longest path from that node to the target. (Using the fastest FU for the job).

And the time-interval $S_i \ldots L_i$ specifies the scheduling time interval in which the operator σ_i can be scheduled, given the time constraint T and thus bounds the solution space, in which we are going to search for the optimal solution.

2.3 Asynchronous circuit design

In this section we discuss some of the properties of the asynchronous circuit design style used in this thesis. As the word asynchronous indicates, an asynchronous circuit does not have a global synchronization event in the form of a clock, but rather is locally synchronized. In this thesis we use four-phase bundled data handshake protocol as component synchronization protocol. This means a signal contains a 1

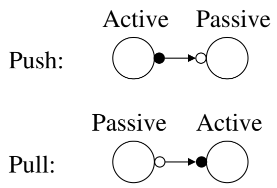


Figure 2.6: Two types of channel communications: push and pull. Data flows from left to right on the channels.

bit request and a 1 bit acknowledge wire additional to the data wires. One example of this the four-phase bundled data push early handshake protocol as illustrated by Figure 2.5. In this protocol the master controls the request and data signals and the slave controls the acknowledge, this also means data is transmitted from the master to the slave. The protocol operates by the master raising the request when the slave is ready to process data, indicated by the acknowledge being low, and the data signals are valid. The slave sees this and reads the data. When data has been read the slave acknowledges this by raising the acknowledge signal. The master then lowers the request signal, removes data and starts preparing for the next transmission. When the slave is ready for the next data the acknowledge signal is lowered. The choice of the four-phase bundled data protocol is an arbitrary choice, our method can be implemented with use of any handshake protocol.

There are two types of channels: push and pull. In a push channel data flows from master to slave and in a pull channel data flows from slave to master. In general the terms master and slave are not used, instead the terms active and passive are used to designate the controlling part of a channel communication and graphically this is illustrated by either a filled (active) or non-filled circle (passive) at the source or destination of a channel, as illustrated on Figure 2.6. The source and destination i.e. the direction of the dataflow is illustrated by the arrow on the channel line.

The asynchronous circuits designed in this thesis are built from a set of asynchronous building blocks called handshake components. As the name implies these components communicate using the handshake protocols. These components are independent components, usually designed using input/output-mode or Muller-C style [92]. All components operate using the same protocol, in this way one could consider this type of asynchronous circuit design as object oriented hardware design. Asynchronous circuits and the circuits presented in this thesis are built from handshake components which implements the equivalent RTL operations as latching data, multiplexing data, addition etc. Each of these handshake components has its own local asynchronous control to ensure proper asynchronous functionality and to handle the asynchronous handshake communication protocol [92]. Besides these asynchronous handshake components which have their equivalent RTL counter parts, there are the demerge/demux components which handle "datawire-forks".

Asynchronous handshake components where all outputs are active and all inputs are passive are push-style; components where all outputs are passive and all inputs

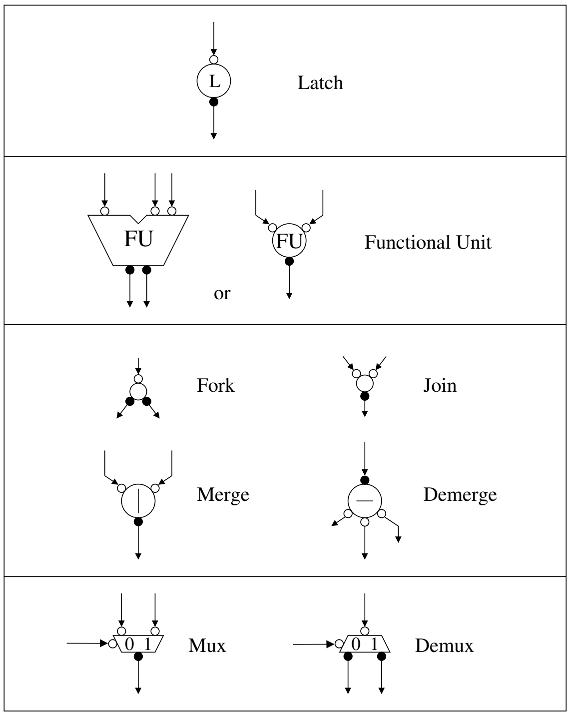


Figure 2.7: A minimum and, for most cases, sufficient set of handshake components.

are active are of pull-type; if all ports are passive the component is of passive-type; if all inputs are active the component is of active-type; others are of "mixed"-type.

The basic set of building blocks are illustrated in Figure 2.7 in their push-form, where applicable, and can be divided into four groups:

Latches Data is stored in latches and could be considered the variables of the circuit. Furthermore with one active input or output they implement the handshaking and support the token flow. In their push form a data write and data read always alternate. In their passive form they operate as the variables of the circuit where the surroundings can write and read data independently and to/from multiple sources and destinations.

Functional Units These are the asynchronous equivalents of combinatorial circuits. We will primarily use the symbol on the left, but some tools will generate the right symbol. In their push form the operation is as follows: First all inputs have to be ready, then compute the functions and distribute the results on the respective outputs. The functional units should be considered transparent from a handshaking point of view, but also versions with input/output latches will be considered.

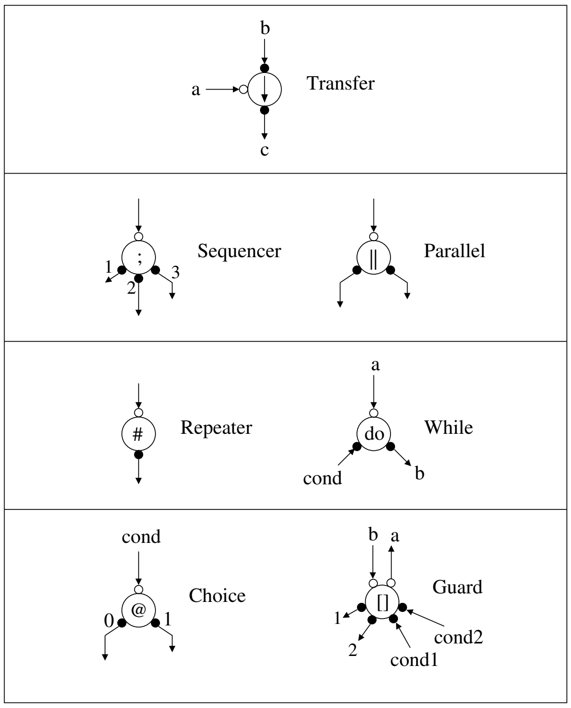


Figure 2.8: Handshake component extension.

Unconditional flow control These components are used to handle parallelism and to merge/split data streams, which are mutually exclusive. The key here is that there is no external control of the data flow. For data streams which are not mutually exclusive either the following group of components have to be used or an arbiter needs to be inserted in front of the component. The merge in shown in the push-form and the demerge is shown in pull-form, which are their only form.

Conditional flow control The MUX and DEMUX components are used to select among several inputs or routing the input to one of several outputs and thus conditionally control the dataflow in the asynchronous circuits.

The functional units in their memory form could by them selves be a network of asynchronous handshake components implementing the function, thus introducing hierarchy into the circuit.

We will need an additional set of asynchronous building blocks to build the asynchronous circuit we desire, these are shown in Figure 2.8 and are all used to build more advanced control circuitry. The groups of handshake components are:

Transfer The transfer component is an active component used to control computation. When activated on input channel a the transfer component moves data from channel b to channel c.

Unconditional control Here there are two components: The sequencer which for each activation executes a sequence, in order, of sub-operations, before completing the input handshake. The parallel executes all sub-operations in parallel and all have to complete before completing the input handshake.

Repetition Infinite repetition is handled by the repeater, which sends an infinite number of activations to its outputs and never completes its input handshake. The while component implements conditional repetition and operates in the following way: Upon activation on input a, the while component inputs condition *cond* and if true output b is activated and the while components repeats this behavior by inputing the next condition *cond*. This continues until *cond* is false then the while component completes its handshake with a.

Conditional control The choice component implements a binary choice by selecting on the input "cond" if equal to zero the "0" channel is activated otherwise the "1" channel is activated. The Guard components is used for implementing multiple selections or guards. Here the component have two selections and operates as follows: when a is activated the Guard component inputs all its conditions, here *cond*1 and *cond*2. The conditions have to be mutually exclusive. If any of the conditions where true the number is returned on a otherwise zero is returned. When b is activated with a positive data value, it is used to activate the operations, here either 1 or 2. The Guard component can have as many selections as required.

Of these components the transfer plays is most important for this research, as it plays the role of event synchronizer; controlling the computation and is the component connecting the control dominant part of the asynchronous handshake network with the data dominant part of the asynchronous handshake network. Transfer components degenerate to simple wire connections containing no logic.

As mentioned in the introduction, there is an apparent resemblance between a circuit designed by a network of handshake protocols and the CDFG describing the behavior of the same circuit. This suggests a simple one-to-one synthesis approach where the CDFG is directly mapped into an asynchronous circuit, as shown in Figure 2.9. Such an approach was more extensively pursued in [73] and is further discussed in the following chapter.

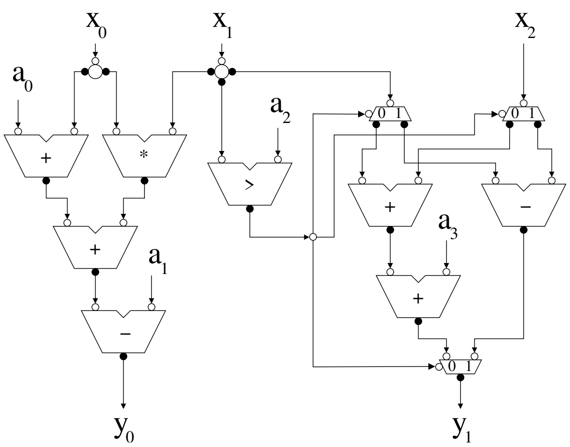


Figure 2.9: Our example designed as an asynchronous circuit using handshake components.

Chapter 3

Related Work

This chapter has two purposes: (i) To present an overview of recent advances in research in behavioral synthesis of low-power synchronous circuits and (ii) to present and compare related work with respect to behavioral synthesis of asynchronous circuits. In doing so, the desirable abilities and requirements for an asynchronous behavioral synthesis approach are uncovered.

3.1 Low power behavioral synthesis, an overview

In CMOS circuits, there are two primary sources of power dissipation [72]: (i) Static dissipation originating from leakage current. (ii) Dynamic dissipation originating from switching transient (short-circuit) current and from charging of load capacitance. The total power dissipation becomes:

$$P_{avg} = P_{switching} + P_{short-circuit} + P_{leakage} \tag{3.1}$$

Of these components the first is the most dominant and is given by:

$$P_{switching} = \langle \alpha_{0\to 1} \rangle_t C_l V_{dd}^2 \tag{3.2}$$

Where V_{dd} is the supply voltage and $\langle \alpha_{0\to 1} \rangle_t$ is the average number of switching per time unit, that a node with capacitance C_l will make a power consuming transition $(0 \to 1)$. For a synchronous circuit $\langle \alpha_{0\to 1} \rangle_t = \alpha_{0\to 1} f_{clk}$, where $\alpha_{0\to 1}$ is the average number of times the node switches per clock cycle and f_{clk} is the clock frequency.

It is well-known that resource sharing destroys correlation between inputs and the computation and therefore increases the power consumption of the circuit. Furthermore, there is usually an overhead associated with resource sharing which will lead to a larger power dissipation. On the other-hand reducing the area of the circuit leads to a reduction of C_l which reduces the power consumption. For future deep submicron technologies leakage power will become more dominant. Therefore as leakage current is proportional to area, resource sharing has the potential to reduce leakage power dissipation. But as resource sharing also have an impact on on-off times for functional units and therefore leads to longer activation times which counters this effect.

There are three dominant approaches for behavioral synthesis targeting reduced dynamic power dissipation:

- Low-power behavioral synthesis [44, 19, 57, 61, 94, 69, 70, 84, 45, 89] through arranging the computation such that the internal switching activity is minimized: $P \sim \langle\alpha_{0\to1}\rangle_t$. The design goal is to find $\min(\langle\alpha_{0\to1}\rangle_t)$.

- Low power behavioral synthesis through voltage scaling [55, 27, 10, 80]. Usually low-power designs operate at voltage-levels just above $2|V_t|$, thus the benefit from voltage scaling lies in speeding up a few critical computations at a power penalty, which is then more than canceled by choosing slower low power functional units at non-critical places in the circuit.

- Power aware behavioral synthesis [102, 5, 1] characterizes methods which targets the generation of a specific power profile of the circuit. The goal is usually a uniform flat power profile below a certain power maximum which corresponds to a hard constraint (e.g. maximum power delivered by a solar-panel). The majority of these algorithms are either based on meta-heuristic algorithms, or two-step algorithms, where in step one a traditional time constrained schedule is constructed and in step two the schedule is made "power-aware".

Usually there is an area penalty associated with these low-power techniques compared to non-low-power techniques and the different methods have different tradeoffs between area and power.

In the following sections we focus on the first of these approaches. There are many ways to minimize $\langle\alpha_{0\to1}\rangle_t$, but the most dominant are those methods which exploit correlations in input-data as well as in the computation. This body of work can be divided into five groups which we will present in the following. The first group focuses on providing accurate lower bounds on power consumption for use in synthesis. The second group focuses on scheduling, allocation and assignment reducing the switching activity of the functional units, which is the largest contributor to power dissipation. The third group focuses on reducing switching activity at the CDFG level. The fourth group focuses on proper register allocation for low power. And finally the last group of papers focuses on reducing the power consumption of the interconnect binding functional units and registers together and the impact this has on scheduling, allocation and assignment. In the following we will present a non-exhaustive list of synthesis methods.

3.1.1 Lower bounds on switching activity

In order to find optimal solutions through exhaustive search based methods as branch and bound, it is necessary to bound the solution space using a polynomial approach. This is also useful for measuring optimality of heuristic approaches as the optimal solution is bounded by the heuristic solution and the lower bound. A branch and bound algorithm traces a decision tree whose leafs represent all possible solutions. Given a best solution found during execution of the branch and bound algorithm, a subtree can be pruned if a lower bound estimate of the best solution from the sub-tree yields a larger cost.

In [57, 94] the switching activity metric is defined as the Hamming distance of consecutive input vectors to functional units. Let w_{ij} define the power cost for the variables i and for each operation type j present in the DFG. This is computed based on a representative set of input vectors to the circuit. The central idea is to formulate the low power binding problem with resource constraints as a graph problem by defining an arc-labeled directed graph. The optimization problem is then to cover all nodes with exactly m (node disjoint) cycles with minimum total cost under the constraint that each cycle contains exactly one backward arc. The total cost is the sum of the arc weights of all cycles. Each cycle of a solution to this problem represents one resource, while the nodes of a cycle are the operations bound to it. The authors prove that the following ILP problem provides a lower bound on the low power binding problem with m resources:

$$z = min \sum_{i,j=1}^{n} w_{ij} x_{ij} \tag{3.3}$$

subject to

$$\begin{array}{ll} \sum_{j=1}^{n} x_i j = 1 & i = 1, ..., n \\ \sum_{i=1}^{n} x_i j = 1 & j = 1, ..., n \\ \sum_{i \geq j} x_i j = m & \end{array} \tag{3.4}$$

with $x_i j$ integer. In this formulation it is not guaranteed that precedence constraints, specifying operation a has to start after operation b, are fulfilled, hence a solution of the ILP problem delivers only a lower bound on the switching activity. Furthermore, the problem is a relaxation of the optimization problem as there are no constraints forcing each cycle to have exactly one backward arc. Instead of solving the ILP problem, a polynomial time bounded approach is proposed which approximates the ILP problem based on Lagrangian relaxation.

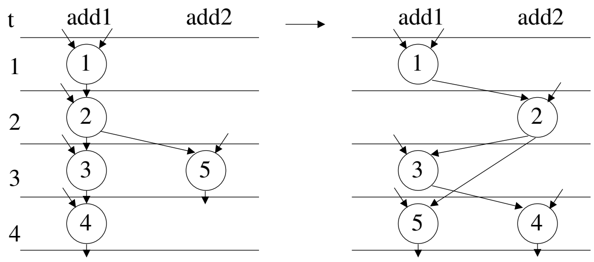


Figure 3.1: Optimizing schedule (from left to right) for reuse of input variables and reduction of switched capacitance. Operation 3 and 5 uses the same result from 2 in successive steps.

3.1.2 Reducing switching activity of functional units

The reduction of switching activity of functional units can be accomplished by scheduling operations such as to increase the correlation of the data presented to the functional unit. The first step in this direction is to observe that the average switching activity of any functional unit is significantly reduced if one of the operands remains unchanged [69, 70]. As operands are usually reused more than once in computations on the same type of functional unit, there is a basis for grouping operands together in the scheduling and binding process. The central idea is to group reusable operands together on the same particular functional unit and to execute these in successive time-slots/operation-groups. The idea is shown in Figure 3.1. In [69, 70] this is accomplished by extending the List-scheduling [67] to a Low Power List-scheduling by adding more heuristics. The traditional List-scheduling operates by having a priority queue of all ready operations determined by urgency, more precisely the difference the ASAP-ALAP interval. In the Low Power List-scheduling operation that share operands are grouped into operand-sharing sets. Once an operation has been scheduled, the other operations in the group are moved up to top priority and are scheduled successively, until an operation outside the set gets urgency zero, which is then set for immediate execution.

The next step is to generalize this observation into scheduling operations such as to increase the correlation between consecutive inputs to a functional unit [89, 45].

Again the list-scheduling heuristic can be modified to include this data correlation [89] and to operate by, besides the set of operations $U_k$where all predecessors have been scheduled, maintaining the set of most lately scheduled operations for each functional unit L_k. At any point the algorithm tries to schedule the operations that consume less power. By scheduling operations in this way there are more candidates in the ready set when power hungry operations are scheduled. For evaluation of the priority for the scheduling a power metric is used. Multiplexer power is no considered in this scheme. Let c_j be the switched capacitance from scheduling operation j on functional unit k where operation i was executed previously $i \in L_k$. If the operation is commutative, then operand swapping is tried to find the smallest switched capacitance. This information is stored for register binding. c_j is normalized with respect to the total switched capacitance of all operators in U_k of same type. The cost of the candidates are set to:

$$priority = \omega c_j + (1 - \omega) t_j^L \tag{3.5}$$

where t_j^L is the ALAP time of operation j relative to the average ALAP time of candidates in U_k of the same type. Parameter ω is the weight given to relate power importance to meet time-deadline importance.

The Force-Directed scheduling method can also be modified for low power synthesis [45]. The algorithm models the switched capacitance of an sequence of two consecutive operands to a functional unit as the spring constant k and the probability of selecting the corresponding sequence is modeled as the displacement x, in the force equation $F = kx$. Thus, a force is associated with each feasible combination of forces which is used to make a power-optimal scheduling decision. This metric is then used in the Force-Directed scheduling method [77] to solve the behavioral synthesis problem for low power digital circuits.

The low power binding problem for a finite set of functional units having a single instance type/single-architecture can be formulated as a min-cost flow problem [31]. This problem is solvable, unlike the generalized low power binding problem functional units having multiple architectures which is an ILP problem. In [31] two polynomial algorithms are presented to heuristicly solve the ILP problem. The first graph-based method iteratively utilizes the single-architecture flow formulation for architecture and then chooses the least power consuming assignment from the set of candidates. Afterwards, the possible unassigned operations are assigned through a node coverage algorithms that follows another flow formulation. The node coverage algorithm runs iteratively until all operations are covered. The second technique assigns the operations to the functional units of multiple architectures in incremental steps similar to the left-edge algorithm.

There are many other methods for addressing the low power synthesis problem [84, 61, 44] these methods involve specifying the problem as auction based non-cooperative finite game, iterative optimizations and constraint logic programming.

3.1.3 Reducing switching activity at CDFG level

A different more radical approach is to design complex custom low-power functional units such as FFTs and filters and use these as buildings blocks for the circuit in addition to simple functional units as adders and multipliers [60]. This requires for the synthesis approach to be able to map groups of operators on these custom functional units, as shown in Figure 3.2. The method also provides techniques for resynthesis of the functional units to match the constraints and techniques for mapping multiple behaviors onto the same complex functional unit. The meta-heuristic approach used for the design space exploration is based on finding a sequence of incremental moves where only the last move has to generate an improvement in the cost function (the intermediate steps are allowed to move to unoptimal state-space solutions). The sets

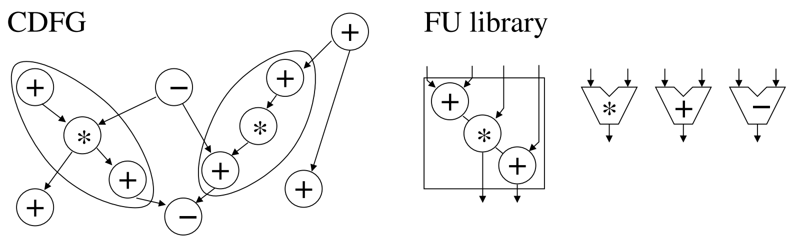


Figure 3.2: Finding groups of operations in the CDFG to match the low-power functional units in the library.

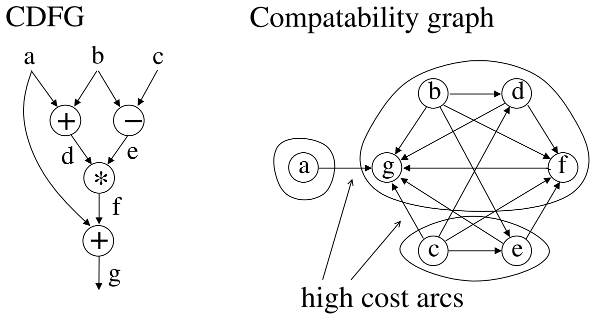


Figure 3.3: Generating the compatibility graph and performing a minimum cost clique-partitioning, asumming the shown arcs have a high switching capacitance cost.

of moves are: i) Simple and complex functional units are replaced by new modules from the library. ii) Complex modules are resynthesized. iii) Simple operations are combined into complex operations. vi) operations are split in to two separate operations. A Tabu-search [43] mechanism ensures solutions are not repeatedly traversed, this method is know as the variable depth search.

Addressing the low power synthesis problem directly at the CDFG level has the potential for large power savings [81, 82]. The proposed CDFG-transformation techniques involve: i) Reducing the total number of operations to be performed by common sub-expressions elimination, loop merging and distributivity. ii) Reduction of spurious switching transitions due to finite propagation delays from one logic block to the next (dynamic hazards). These extra transitions are a complex function of logic depth, input pattern and skew. To minimize these unwanted transitions, signal path balancing and logic depth reduction is handled. The sequence of optimization moves are handled by the use of a heuristic/probabilistic search algorithm.

3.1.4 Memory allocation for low-power

The goal here is to find the appropriate number of registers and the associated binding to minimize power consumption in the registers.

The register binding problem can be formulated as a minimum cost clique covering problem [19]. The power consumption is computed based on statistical information derived from assumptions on probabilistic input distributions. The power dissipation model is based on the Hamming distance and the capacitance of the registers are modeled as a fixed load for a given library. The paper [19] defines the compatibility graph $G(V, A)$ as the graph where the nodes are variable intervals and the directed arcs A between two variables if their variable life times are non-overlapping and end-life-time of the source variable is less than the start-life-time of the target. Each arc represents a possible assignment and carries the switched capacitance difference between the two variables. The register assignment problem is then formulated as a minimum cost clique partitioning problem of that graph. They show that the unoriented compatibility graph for the data values in a scheduled dataflow graph without cycles and branches (a DFG fragment) is a comparability graph (or transitively orientable graph) which is a perfect graph. This is a useful property as clique partitioning problems can be solved in polynomial time for perfect graphs, through a max-cost flow reformulation of the problem, giving the minimum total power consumption on the registers in the circuit.

The memory allocation for low-power problem can also be formulated as a network flow problem [18]. This work focuses on solving the problem of rapidly resolving the problem to optimality for an incremental change of the schedule for use in low power scheduling methods. This is a two-step process: i) A max-flow computation involving a valid flow solution while retaining the previous solution as much as possible and ii) a min-cost computation which incrementally refines the found flow solution, using the concept of finding a negative cost cycle in the residual graph for the flow.

3.1.5 Interconnect design for low-power

One way to reduce the switching activity in the interconnect connecting registers to the functional units is to isolate/signal guard parts of the interconnect [110]. For interconnect, in this case built by a multiplexing network, it is not justifiable to insert latches through-out the routing network, when compared to the power overhead introduced by such a method. In addition to make use of data-correlations, it is proposed to freeze the inputs of the multiplexors to a fixed (hardwired) value, denoted the filler value. The probabilities for the different switching characteristics are computed by simulating the CDFG in which the binding and scheduling information is back-annotated. The algorithm for computing the filler values is a simple polynomial algorithm running through computing the most probable value. The power reduction of the interconnect is then built into an iterative behavioral synthesis algorithm for scheduling and binding to find the optimal low-power circuit. The meta-heuristic approach used for this is based on finding a sequence of moves where only the last move has to generate an improvement in the cost function (the intermediate steps are allowed to move to unoptimal state-space solutions), a tabu-search mechanism ensures solutions are not repeatedly traversed.

For bus-based micro architectures, reduction of switching activity can be accomplished in two ways [29, 30, 28]: (i) Through multiplexing the signals onto the buses in the correct order. (ii) And choosing the optimal set of busses and their connection between functional units and registers. For design of the buses, the average signal switching activity for all nodes in, and inputs to, the CDFG are computed by repeated simulation using a representative set of input vectors. Using this data the switching activity matrix SA^k_{ij}, for successive data transmissions $i \rightarrow j$ for bus k, for a given bus configuration is computed and the lowest energy is selected. Simulated annealing is used to handle the complete synthesis process including bus configuration design.

3.2 Asynchronous behavioral synthesis, an overview

Synthesis of asynchronous circuits falls mainly in two categories: (i) synthesis of small-scale sequential control circuits [26, 41, 106] and, (ii) synthesis of large-scale circuits based on syntax-directed compilation from CSP-like languages: Tangram [11, 100], OCCAM [17], Balsa [8, 36] and ACK [59]. Several tools exist (in the public domain) in these areas, and these tools have been used to design industrial scale circuits.

Synthesis methods for generating small-scale sequential control circuits are low-level logic synthesis methods for the design of asynchronous logic, the asynchronous equivalent to synchronous control logic synthesis. syntax-directed synthesis is a line of high-level synthesis where there is a one-to-one correspondence between the high-level programming language specifying the circuit and the circuit itself.

Besides those two main lines of research there are a number of other attempts. One of the most promising is desynchronization [14, 25] which relies on synchronous behavioral synthesis and then in the low-level logic synthesis phase substitutes the clock and the synchronization with asynchronous handshaking and control.

We illustrate the design flows of the different synthesis methods currently developed for asynchronous circuit design and indicate the different levels of abstraction in the synthesis process. The position inside each level is unimportant and does not signify any further degree of abstraction. The levels of abstraction are:

Abstract This is the level where the behavior is expressed only by essential operations and their essential dependencies.

Behavior The level where the behavior is specified in the form of a programming language and as such may contain restrictions in expression form, which may correspond to non-essential behavior.

Architecture In this level the behavior is specified by architectural information consisting of larger-scale components implementing a predefined behavior.

Gate/Logic At this level the behavior is expressed in the form of an architectural design built by logic gates.

Physical This level represents behavior in physical form either as a layout or as a physical model of a layout.

Not all details will be indicated in the figures describing the different synthesis flows, only those which are of special nature or original to the method in question.

In the following we present a non-exhaustive list of synthesis methods, grouped together as to how their synthesis flows relates to each other.

3.3 Asynchronous logic synthesis

Asynchronous logic synthesis is the building method behind asynchronous synthesis as these methods are used to generate the asynchronous logic. This area has been and still is, the focus of a majority of the research in asynchronous circuit synthesis. Asynchronous logic synthesis can largely be divided into two groups: (i) Synthesis of small-scale sequential input/output-mode control circuits or handshake components [26, 41, 107, 108]. This is usually done through tools like Petrify [24, 26]. The behavior of the asynchronous circuit together with its environment is specified using a 1-bounded 1 color petri-net called a Signal Transition Graph (STG). The approach is limed by the NP-hardness of the synthesis problem with several improvements implemented through: Reducing the search space using heuristics [76]. Series of local graph transformations [91]. Furthermore the problem contains the important subproblem of consistent state coding (CSC), which is also the subject for extensive research [63, 65]. The design of GasP circuits [35, 97] fall under the same category of logic synthesis but employ a different handshake protocol.

The other group is synthesis of larger-scale controllers operating in fundamental mode/Burst mode [40, 41, 107, 108, 109]. These are race-free asynchronous combinatorial circuits with restrictions on both type of operation and the timing of how the environment interacts with the circuit. This synthesis problem is likewise an NP-hard problem which limits the size of the controllers possible to synthesize, but usually larger circuits than for the input/output-mode circuits can be synthesized. Again heuristics are employed to improve on the method [9, 98].

Theseus logic has developed a Synopsys back-end. Here the low-level logic synthesis of control and datapath is implemented using a NCL logic-synthesis leading to an asynchronous circuit. The tool is integrated into Synopsys through the use of special libraries and compile commands [38, 90].

3.4 Asynchronous behavioral synthesis

A number of papers have presented work on behavioral synthesis of asynchronous circuits from DFG or CDFG representations, but they are surprisingly few and they have a different and/or more limited scope [3, 4, 22, 23, 54]. The first paper limits itself to DFGs and focus mostly on a synthesis algorithm and its runtime. The remaining papers address synthesis from a CDFG representation and they target solutions where a centralized controller or a distributed structure of controllers are specified at the level of individual signal transitions (in the form of signal transition graphs or burst-mode state graphs).

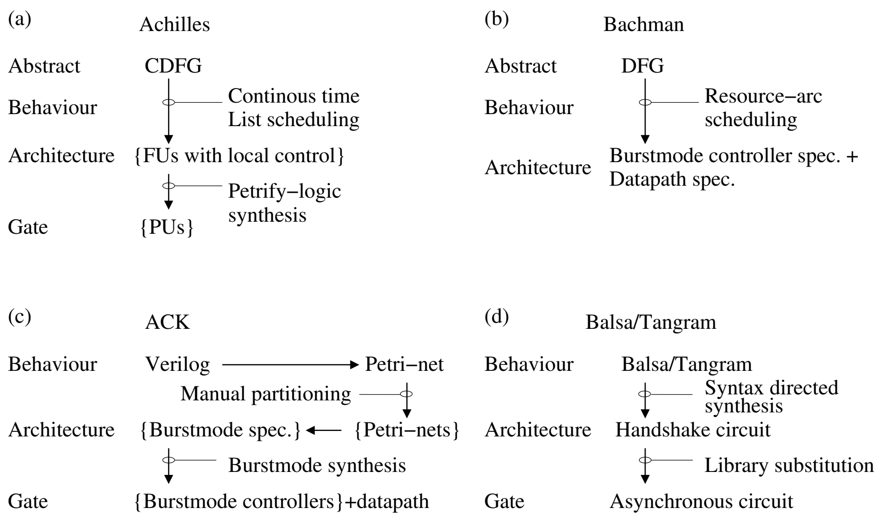


Figure 3.4: Synthesis flow for Achilles, the Bachman approach, ACK and Balsa/Tangram

The synthesis tool Achilles [4, 22, 23] and the synthesis tool by Bachman et al. [3] both represent "pure" asynchronous behavioral synthesis.

Achilles starts from a control data flow graph and uses a modified list-scheduling to generate a schedule in continuous time. The target architecture is a set of independent machines corresponding to each of the functional units in the circuit, as illustrated on Figure 3.5. Each independent FU then implements the appropriate part of the schedule, has its own memory and handles communication with the other FUs. Using this method, there is a possible communication overhead and memory overhead when comparing to a method using a single controller and datapath. The controller of each FU is specified as a Petri-net and synthesized using Petrify. The complete synthesis flow is illustrated in Figure 3.4 (a).

The synthesis tool by Bachman, utilizes a method designated as resource-edge scheduling, which is a form of scheduling where the additional ordering imposed by scheduling is represented as additional graph-dependencies added to the data flow graph, as illustrated in Figure 3.6. It is unclear from their work whether the starting point is a DFG or if they have included DFG extraction from VHDL/Verilog. The synthesis flow is illustrated in Figure 3.4 (b). The focus in their work is on architectural scheduling and series of algorithms have been developed, including scheduling and a continuous left-edge algorithm with the target architecture being a central controller and datapath. They primarily address the runtime and complexity of the developed algorithms.

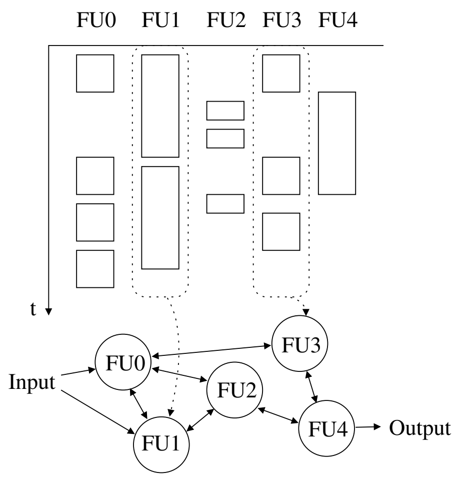


Figure 3.5: Connection between the continuous schedule and the assignment to the asynchronous architecture for Achilles.

3.4.1 Partitioned controllers

Asynchronous Circuit Kompiler (ACK) [48, 59] is a high-level synthesis system, which is based upon a traditional circuit design style; consisting of a datapath and a centralized controller. The starting point is a CDFG from which a datapath (functional unit allocation) and a Petri-net describing the control of the datapath is extracted. No behavioral synthesis is involved in this extraction, except the source code could contain pragmas for e.g. sharing a common subexpression. The synthesis process could therefor be characterized as syntax-directed.

The size of the Petri-net prevents direct synthesis of the controller, as this is an NP-hard problem. Instead, it is proposed to divide the controller into a small set of controllers and methods are described for letting multiple controllers jointly control a single functional unit in the datapath, through a boundary layer, also responsible for sending data from the datapath to appropriate controller, as illustrated in Figure 3.7. Unlike Achilles, there is not a one-to-one correspondence between the FU and the controller partitioning. The partitioning of the Petri-net is left to the designer and no automated methods are presented in the work.

The set of manually partitioned Petri-nets are then automatically converted into a set of burst-mode specifications, which is then synthesized into burst-mode controllers. The synthesis of the datapath is handled through Synopsys. The complete synthesis flow is illustrated on Figure 3.4 (c).

Several other approaches employ similar techniques with shared controllers and look into automated methods for partitioning controller into manageable sizes [54, 104, 105].

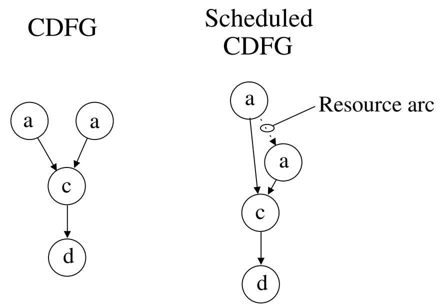


Figure 3.6: Behavioral synthesis mechanism for the synthesis tool developed developed by Bachman.

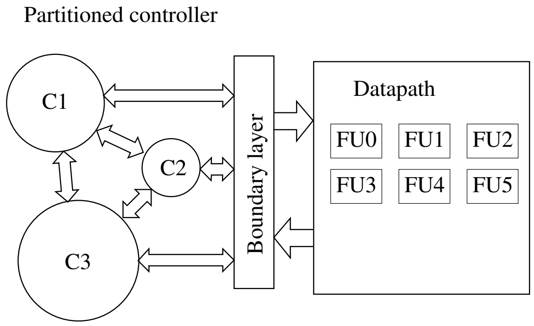


Figure 3.7: Control and Datapath architecture for ACK.

3.4.2 syntax-directed synthesis

Balsa [7, 8, 36], Tangram [11, 12, 100, 101] and OCCAM [17] are CSP type languages specifically designed for synthesis of large scale asynchronous circuits. They employ syntax-directed synthesis into a set of predefined asynchronous handshake components. Both tools are well developed, supported and have been used to design industry scale circuits. The controller consists of a distributed net of handshake components and likewise for the datapath. The flow is illustrated in Figure 3.4 (d).

The syntax-directed compilation approach is radically different from the behavioral synthesis flow used by designers of synchronous circuits. Firstly, syntax-directed compilation is based on a non-standard language, and secondly, and more important, the compiler merely performs a one-to-one mapping of the program text into a corresponding circuit. Although syntax-directed compilation does allow the designer to work at a relatively high level it does not provide any optimizations; "what you program is what you get". In some situations this can be considered an advantage but it also puts more burden on the designer: exploring alternative implementations requires actually programming these, whereas in a traditional synchronous synthesis flow, the designer can quickly and easily experiment with different constraints and

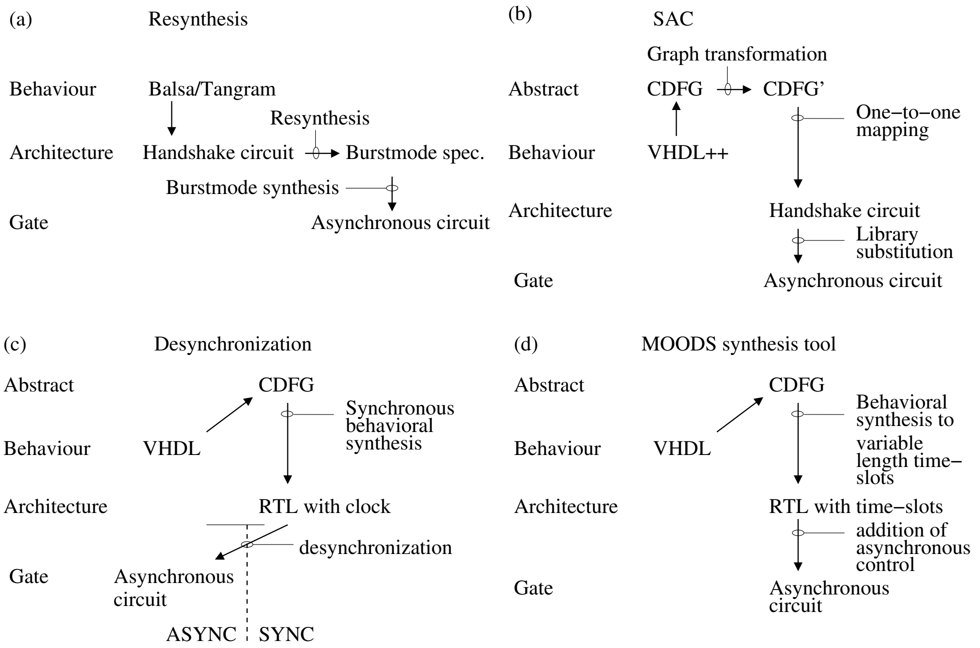


Figure 3.8: Synthesis flow for Resynthesis, SAC, Desynchronization and the MOODS tool.

goals and in this way create alternative implementations from the same program text.

The tools support logic optimization to some degree i.e. in the form of Peep-hole optimizations. These are optimizations where groups of handshake components when placed together in a certain way are replaced by one larger handshake component thus reducing the control logic.

To further improve on this, the resynthesis [20] approach is pushing this even further by grouping parts of the components related to operators in the datapath and re-synthesize the control logic using burst-mode circuits. The flow follows the balsa-flow until the point where the circuit is described by a set of handshake components, these are then resynthesized. The flow is illustrated in Figure 3.8 (a).

The TAST tool [85] is pursuing the same direction but is instead synthesizing the controller from the specification, avoiding the handshake components completely and using a traditional control/datapath architecture. Advances in STG to asynchronous circuit synthesis has allowed this to be used for larger circuits and thus becomes more attractive. The starting point is a VHDL description, from which the Petri-net-specification and datapath is derived. The TAST tool is currently not available in the public domain.

Blunno [15] targets the generation of micro-pipelines directly from a Verilog specification and [62] generates delay insensitive circuits from graph-theoretic specifications, but again there is a one-to-one correspondence between a specification and the resulting circuit.

3.4.3 Synthesis of Asynchronous Circuits

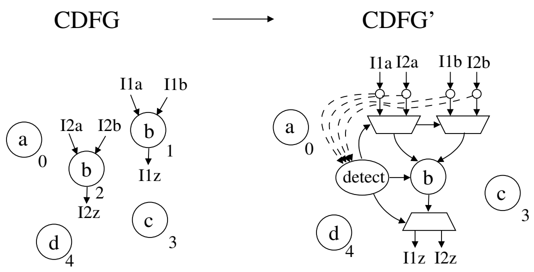


Figure 3.9: Graph theoretic transformations supported by SAC: Disjunctive.

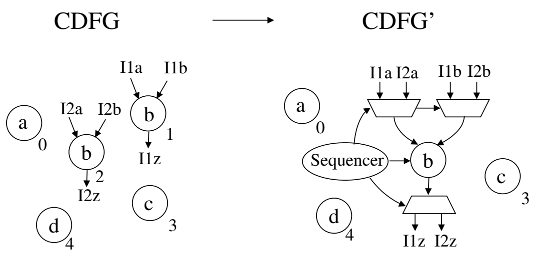


Figure 3.10: Graph theoretic transformations supported by SAC: Deterministic.

In SAC [46, 50, 73] behavioral synthesis is handled at the CDFG level. The tool can synthesize a single VHDL process (assuming inputs and outputs to be handshake channels) into a standard cell circuit implementation. Two types of synthesis methods are supported: Non-performance degrading resource sharing and performance degrading resource sharing. The synthesis flow begins by extracting a control data flow graph from the specification (a single VHDL process).

The CDFG is analyzed and resource sharing and operation scheduling, in the form of graph transformations, are performed. Two types of graph transformations are supported:

- Disjunctive resource sharing. Operators that have a disjunctive relation, i.e. from a graph theoretical perspective never can execute at the same time, could be resource-shared to the same operator. The order of execution is handled by a detect component which detects which operation is ready to execute. The method is illustrated in Figure 3.9.

- Deterministic resource sharing. Operators that have a deterministic relation i.e. a fixed order of execution can be established, can be resource-shared to the same operator. The execution order is then controlled by a sequencer component. The method is illustrated on Figure 3.10.

After these optimizations a corresponding circuit implementation is generated. The method utilizes the fact that there is a close correspondence between a CDFG [33, 67, 96] and an asynchronous circuit: The edges in a CDFG can be seen as handshake channels and the nodes in a CDFG can be seen as handshake components – components that are quite similar to the handshake components used in syntax-directed compilation. In this way a simple one-to-one mapping of the CDFG to a network of asynchronous handshake components is performed.

The graph transformations makes this different from the syntax-directed compilation of large-scale asynchronous circuits from non-standard languages. The flow is illustrated in Figure 3.8 (b).

This work represents our initial effort to implement asynchronous behavioral synthesis. The method was discontinued as we found there was a power overhead associated with this method of synthesis. Research into a non-one-to-one correspondence between a CDFG and a handshake circuit might alleviate this.

3.4.4 Desynchronization

Common for these methods is the use of existing synchronous methods and tools as part of the process for generating an asynchronous circuit. In some way these methods represent the opposite of the "pure" asynchronous behavioral synthesis, as all these methods use synchronous behavioral synthesis to perform architectural synthesis before employing asynchronous logic synthesis to generate the final circuit.

Desynchronization [14, 16, 25] makes use of existing synchronous methods and tools to synthesize a synchronous circuit down to gate-level and then replace the synchronous control logic and the clock by asynchronous control logic and asynchronous handshaking. The synthesis flow is illustrated on Figure 3.8 (c). Two directions exist for generating the asynchronous control logic:

Synthesis [25] Infer the overall behavior from the synchronous behavior, this involves construction of a STG description or a burst-mode description and then synthesizing the central controller. This approach is limited to smaller-size control circuits, limited by logic synthesis capabilities.

Substitution [14] Systematically replace synchronous components by local handshake components through a transparent one-to-one correspondence. This approach generates less optimal solutions than the former, but can be used for larger-scale synthesis.

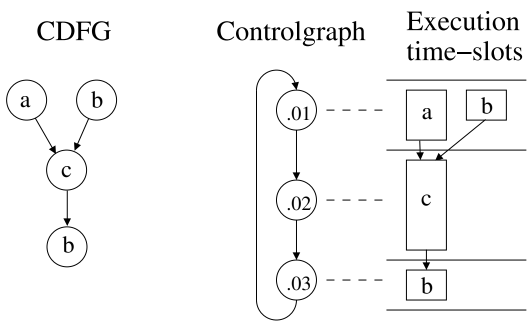


Figure 3.11: The behavioral synthesis mechanism of the MOODS synthesis tool.

3.4.5 Variable length time-slot behavioral synthesis

Sacker [88] proposes a method which resembles the synchronous behavioral synthesis flow but where the target operation group time-slots are of variable length. Borrowing from compiler technology and synchronous synthesis the group has extended their existing synchronous behavioral synthesis MOODS to handle asynchronous circuits. The target is a single control sequence of operation-groups, where each operation-groups can consist of several operations in parallel and have the execution time of the slowest operation in the group. Multi-cycle operations are not supported, but chaining is. However chaining implies data is fed directly between two FUs with-out being stored in registers and therefore no resource sharing of the FUs involved in chaining is allowed. There has to be a sufficient number of FUs such that for all operations-groups, all the operations in an operation-group have a direct mapping to a FU.

The starting point is a VHDL behavioral model. From this an intermediate format, they call ICODE is extracted, which is a representation equivalent to a CDFG. Then scheduling allocation and binding is performed, with the “synchronous” schedule represented by a control-step graph. The asynchronous control is handled by mapping the elements of the control-step graph via predefined asynchronous controller-cells to an asynchronous circuit. The datapath is synthesized through a set of templates. The used asynchronous signaling is based on 4-phase handshake-protocols. The flow is illustrated in Figure 3.8 (d).

3.5 Summary

Currently research in behavioral synthesis of asynchronous circuits is primarily focused on syntax-directed synthesis and desynchronization. Besides there is a multitude of more or less successful attempts for high-level synthesis.

There are three aspects we would like our asynchronous behavioral synthesis to contain:

- Ability to construct systems operating in continuous time and using methods from behavioral synthesis and Operations Research in continuous time. Desynchronization methods are limited by their use of a discrete time-evolution to find the optimal schedule.
- Ability to use existing behavioral synthesis methods developed for synchronous synthesis, such as the methods for low-power behavioral synthesis reviewed in the beginning of this chapter. Leveraging on existing techniques that are well proven both in theory and practice will prove very beneficial.
- Use of handshake components both for controller synthesis and datapath synthesis to facilitate constructions of large scale designs. For an asynchronous behavioral synthesis to be effective it has to be able to synthesize industry-scale designs.

The research presented in this thesis tries to implement these aspects by introducing a computation model allowing the use of both synthesis methods of synchronous discrete time and methods for continuous time and targets asynchronous handshake components both for datapath and controller synthesis. As an implementation we currently build upon the balsa language, but this is not a restriction our work could easily be extended to target other languages or design approaches.

CHAPTER 4

Behavioral Synthesis for Asynchronous Circuits

Synchronous circuit synthesis utilizes a simple model for implementing synchronous computation and this method has proven to be highly successful. Therefore, rather than to invent a different computation model, we adapt the existing computation model for asynchronous circuit synthesis. This has the added advantage of opening up for the use of many of the existing methods from synchronous behavioral synthesis in asynchronous circuit synthesis. In this chapter we address this in detail.

4.1 From synchronous to asynchronous behavioral synthesis

Let us first review and analyze the elements of synchronous behavioral synthesis.

Based on the CDFG, synchronous behavioral synthesis involves three sets of transformations in order to create a suitable hardware architecture;

- Scheduling, in which operator nodes of the CDFG are grouped into operation-groups or time-slots, and where the execution of the next operation-group is handled by a synchronization event, E^i, where i strictly orders the events in time. In the case of synchronous behavioral synthesis E^i is controlled by the system clock.

- Allocation, in which the minimum hardware resources/ functional units (FUs), required for execution of the operation-groups are determined.

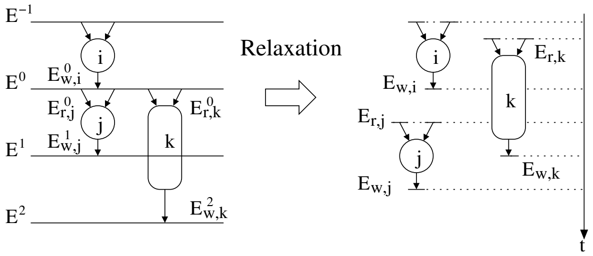


Figure 4.1: Adapting synchronous synthesis (left) into the asynchronous handshake domain (right).

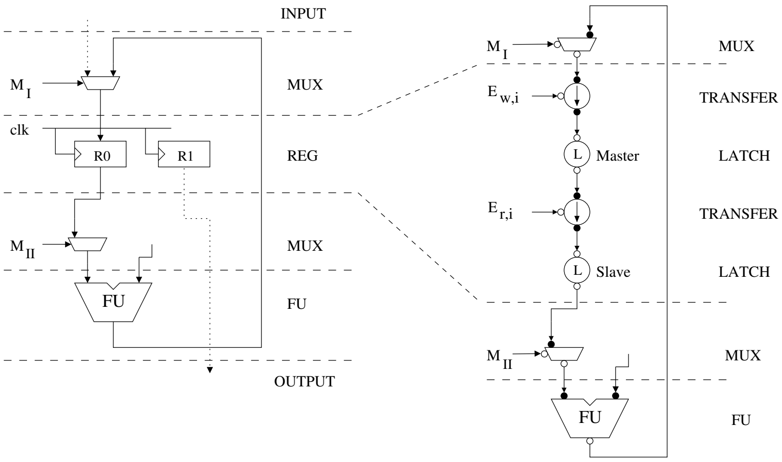


Figure 4.2: First step in adapting the synchronous computation model into the asynchronous domain.

- Binding (or assignment), where individual operator nodes are tied to specific hardware resources.

The synchronization events determine (i) the beginning of executing an operation (ii) writing the result of an operation.

The CDFG extracted in the synchronous behavioral synthesis is a 1-bounded colored Petri net, where colors represent data values, edges represent places, and nodes represent transitions. Interestingly, the Petri net model is based on an asynchronous execution semantics which should make it an obvious model for asynchronous synthesis as well. In the synchronous synthesis, Figure 4.1 (left), operations are ordered according to a global synchronization event, E^i, i.e., read events ($E_{r,j}$) for operator j happens at the same point in time as the write events ($E_{w,i}$) for operator i in the previous operation-group: $E^0_{w,i} = E^0_{r,j} = E^0$, and furthermore all operations in an operation-group are executed simultaneously: $E^0_{r,j} = E^0_{r,k} = E^0$.

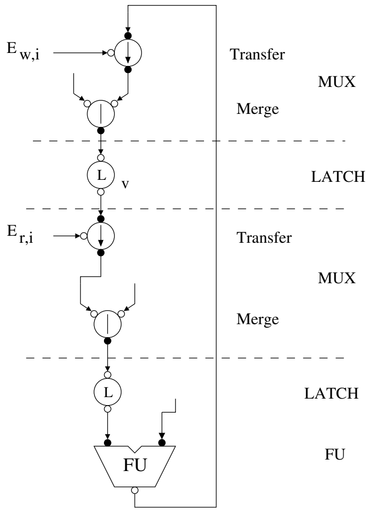


Figure 4.3: Rearranging components to get the initial computation model.

If we relax these assumptions: $E_{w,i} \neq E_{r,j}$ and $E_{r,j} \neq E_{r,k}$ as shown in Figure 4.1 (right), and if we make these synchronization events controlled by the controller, we can create a hardware architecture consisting of a datapath and a controller which operates in continuous time.

We start with the synchronous computation model as shown in Figure 4.2 (left). This is a standard Moore machine datapath with memory (register) controller by a clock and some functional units (combinatorial circuitry) to operate on the data. To move data back and forth between the memory and the functional units two layers of muxes control the data flow, controlled by signals M_I and M_{II}. The first step in adopting this computation model is to move the components into the asynchronous handshake domain. We will use this to model the asynchronous timing assumptions. Then we expand the registers by splitting the synchronizations events: $E_{w,i} \neq E_{r,j}$.

The next step is to let the synchronization events completely control the computation (datapath). This is done by rearranging the latches and transfer components such as reducing the muxes to merge components. From this we get the initial computation model shown in Figure 4.3. In this model the individual synchronization events $E_{w,i}, E_{r,i}$ control the computation. From the model it shows that $E_{w,i}$ is active during the actual computation and $E_{r,i}$ is active only for the transfer from latch to latch. This model is suboptimal as we are using a latch for temporary data and the FU can only have one target.

To continue from here we have two options which reflect the properties of our datapath, and lead to two datapath topologies: The first we designate *alpha* and here the functional units are purely combinatorial without latches on input and output ports. The second we designate *beta* and here the functional units have normally

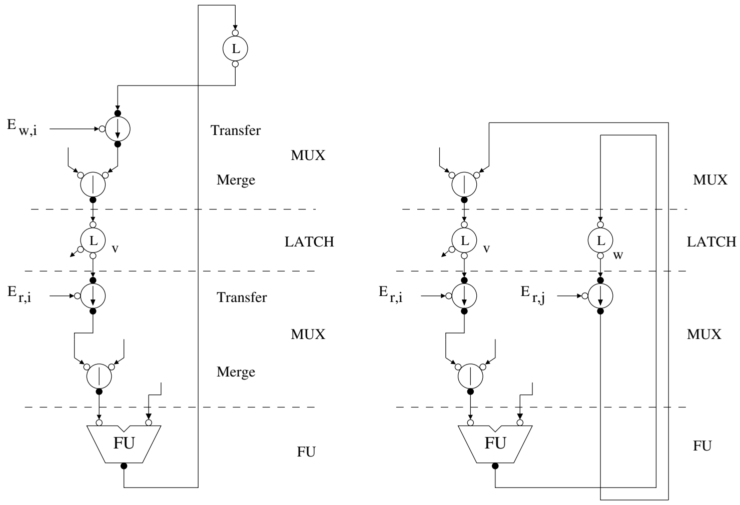


Figure 4.4: Rearranging to get the temporary variable into the memory.

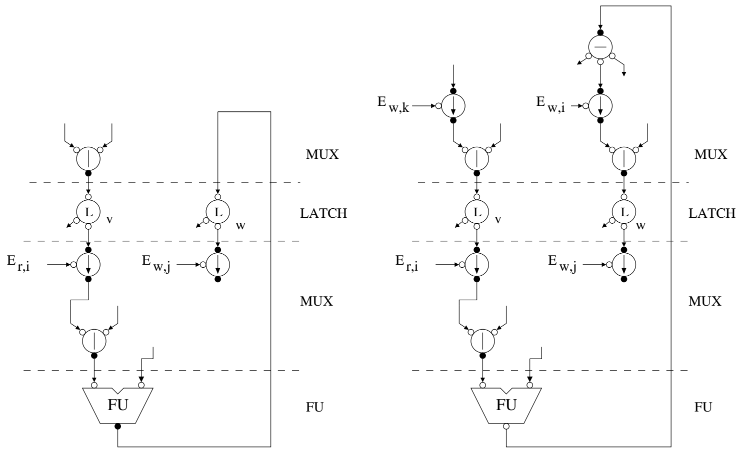


Figure 4.5: Final computation model without normally opaque latches on input and output ports of the functional units.

opaque latches both on input and output ports. The use of input and output latches tends to increase speed and to reduce power consumption by preventing spurious signal transitions to propagate beyond latch boundaries. If input and output latches are not used, more variable latches may be needed in the datapath in order to accommodate the longer lifetime requirements and in order to avoid auto assignments. In the following we pursue both directions, starting with *alpha*, no latches on input and output ports:

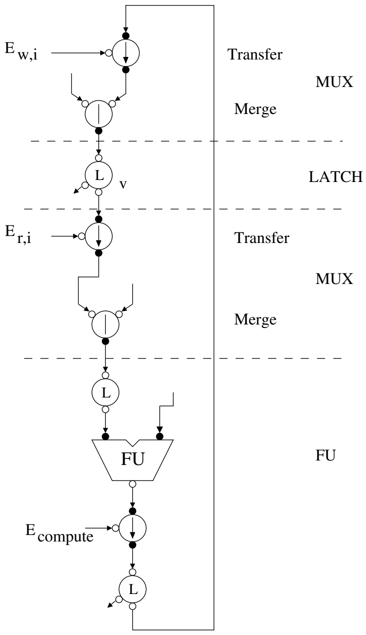


Figure 4.6: Computation model with input and output latches.

> Rearranging the temporary latch after the FU as shown Figure 4.4 (left), next we move the temporary data into the memory becoming Lw by substituting $E_{w,i} \rightarrow E_{r,j}$ getting Figure 4.4 (right). We still have the restriction that the FU always writes to Lw, but Lw can be used by others. By reinserting write synchronization events we get a computation model which allows all latches to be used as source and target for all functional units. This is shown in Figure 4.5. $E_{r,i}||E_{w,j}$ moves data from Lv to Lw through the FU doing computation. Restriction: Lv cannot be used as both source and target and while Lv and Lw are being used in computation, there can be: (i) no other write to Lv and (ii) no-other read or write to Lw.

Next we will pursue the datapath (*beta*) with latches on input and output ports:

> We already have input latches so we insert output latches and are thus forced to get an extra synchronization event controlling the computation. The execution of a computation takes the following form: $\{E_{r,i}\}; E_{compute}; \{E_{w,j}\}$, as shown in Figure 4.6. Then we remove the control of this computation event by decoupling the control of the FU making it an independent process as shown on Figure 4.7 (left). This model can operate with arbitrary synchronization events. The final computation model is shown

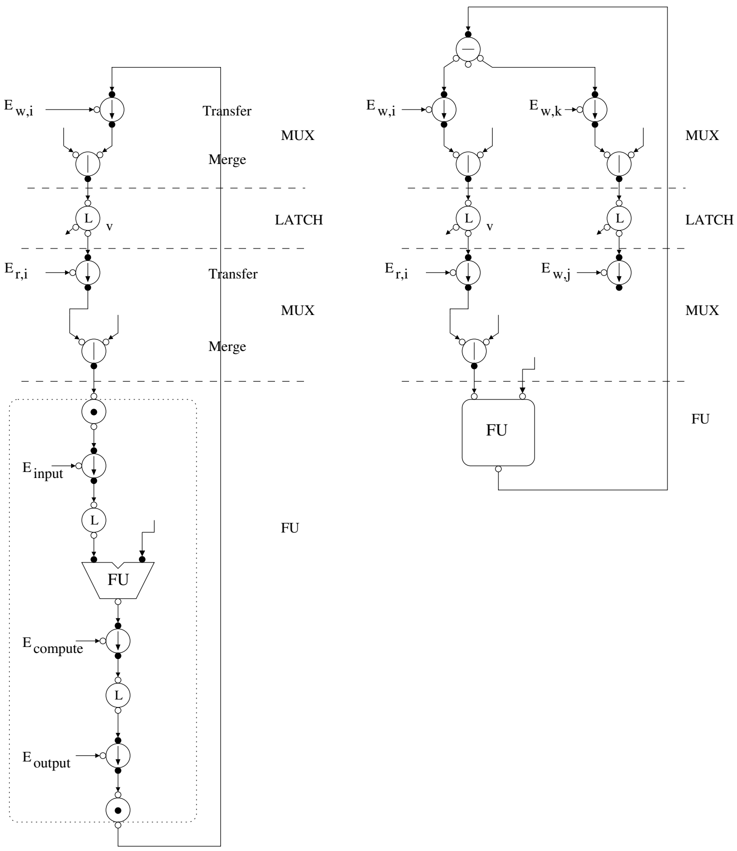


Figure 4.7: Final computation model with normally opaque latches on input and output ports of the functional units.

in Figure 4.7 (right), it resembles the synchronous architecture but it is completely asynchronous.

Both models have the same architecture; the only difference is the time the data needs to be held in the source latch and restrictions on the target latch. Both methods can therefore be used heterogeneously in the same datapath, using the most suitable method for the specific FU, we will denote such a mixed model *gamma*.

This idea allows us to use any of, but not restricted to, the many synchronous behavioral synthesis techniques to obtain a hardware architecture (datapath and controller) and then to implement this architecture using asynchronous circuit techniques. At the same time, this idea allows the use of behavioral synthesis techniques operating in continuous time.

4.2 Asynchronous behavioral synthesis

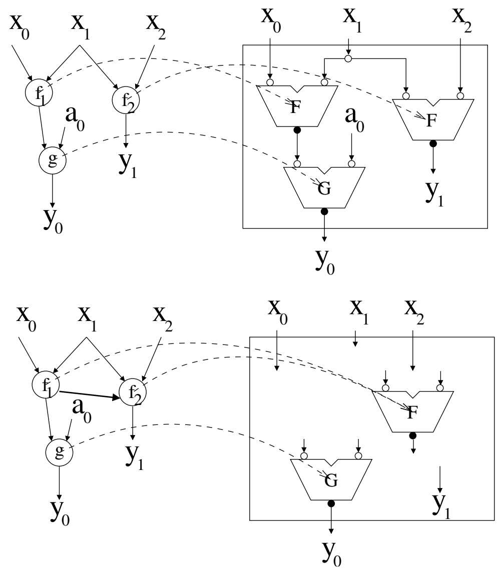


Figure 4.8: (Top) One-to-one correspondence between CDFG and asynchronous circuit. (Bottom) scheduled CDFG using a non-essential precedence-constraint (thick solid line) and mapping to asynchronous circuit.

Having approached our target computation model from the synchronous side we will now approach our model from the asynchronous side. The starting point is the one-to-one correspondence between the CDFG representing the computation and the asynchronous handshake component network, as shown in Figure 4.8 (left) with a small example. For this CDFG there is a single essential precedence constraint: $f_1 < g$. The delay of the circuit is given by $T = \max(T_{f_2}, T_{f_1} + T_g)$ and the total area is given by $A = A_{f_1} + A_{f_2} + A_g$.

The basic idea behind constraint based synthesis and resource sharing is to perform time-multiplexed mapping of several operators onto a smaller set of functional units. As only one operation can be performed per FU, this requires memory. In this setting the time-multiplexing corresponds to the scheduling. The mapping of operators to FUs, correspond to the assignment, and the set of FUs themselves correspond to the allocation. The scheduling can be represented by a minimal set of non-essential precedence constraints [95] or resource-arcs [2], specifying the time-ordering. This is illustrated on Figure 4.8 (right) with the non-essential precedence constraint: $f_1 < f_2$ represented by the thick arrow from f_1 to f_2, which are mapped onto the same functional unit F. In this case the delay of the circuit is given by $T = \max(T_{f_1} + T_{f_2}, T_{f_1} + T_g) = T_{f_1} + \max(T_{f_2}, T_g)$ and the total area is given by $A = A_{f_1,f_2} + A_g$.

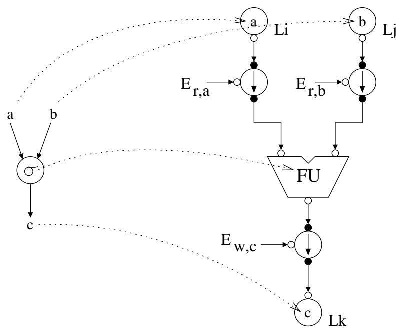


Figure 4.9: Mapping operator σ to a FU.

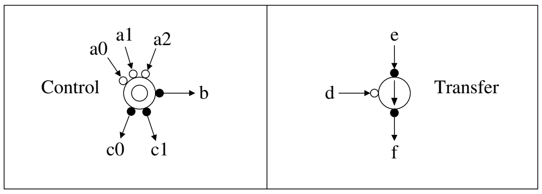


Figure 4.10: The control handshake component and the transfer handshake component.

To proceed from here we need the mapping of a single operator σ with source data a, b in latch Li and Lj respectively, and target data c assigned to Lk which is given in Figure 4.9, as the simplest construction of such a mapping. To construct the control circuits for this mapping we introduce the dual component to the transfer handshake component, the control component c.f. Figure 4.10. The behavior of the control component is a follows: First the component waits for a request from all input ports $a0, a1, ...$ then a request is placed on output port b. When an acknowledge arrives from b the handshake with input ports $a0, a1, ...$ are completed and the handshake with output ports $c0, c1, ..$ are commenced and completed. The STG for a four phase implementation of the component is shown in Figure 4.11.

Together with the transfer component the control component maps the CDFG onto a control part and a data part. This depends whether our functional units have input/output latches or not. Both solutions to this problem are shown in Figure 4.12. We now see there is a direct correspondence between the CDFG node and the control node of our asynchronous circuit and the functional unit mapping. For the *alpha* model there is a direct correspondence between the CDFG node and the control component. For the *beta* model there is a direct correspondence between the CDFG input arcs and the control node responsible for the loading of the data to the functional unit and the direct correspondence between the CDFG output arc and the control node responsible for the reading of the result of the functional unit. We will continue with the *alpha* model.

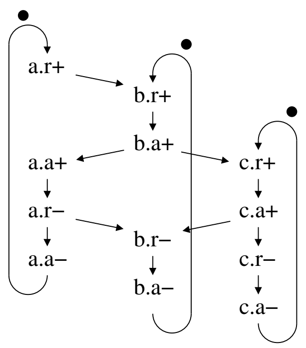


Figure 4.11: Four phase STG for the control handshake component with only one a and c channel. For multiple $a0, a1, ...$ and multiple $c0, c1, ...$ the a and c have to be replaced by concurrent handshaking on all these channels.

Performing a one-to-one mapping of the control nodes in the CDFG and the *alpha* model generates the circuit shown in Figure 4.13. Using this approach we have moved from the one-to-one correspondence between CDFG and functional units to model with a one-to-one correspondence between the CDFG and the control part of the handshake circuit only. The functional units now follow the behavioral synthesis allocation. The control part of the handshake circuit could be implemented using any methodology for asynchronous state-machine design: Burst-mode [109], Petrify [26], set of handshake components [92] and Balsa/Tangram [7, 11] style controller.

We will implement the control part of the circuit using a different method to generate the events, which uses handshake components such as sequencers and parallel etc. These are better suited for our behavioral synthesis algorithms operating with a sequence of discrete events.

The same datapath and control circuit can be built for the *beta* model, using the same approach. To build a compact efficient computation unit (datapath) we will look at how to generate this in general in the following section.

4.3 Datapath synthesis

Assume we are given a CDFG, and that scheduling, allocation and assignment has been performed as shown in Figure 4.14, using the FU library shown in table 4.1 (to begin with, the schedule will not include the load of input data to the circuit and storing of the results). The FU library has been normalized with respect to the ALU component. We will consider the schedule to operate in continuous time. However it is of no importance whether the schedule has been obtained using an asynchronous scheduling method or through a synchronous method which has been relaxed into continuous time, as discussed in the previous section. Note that the operator nodes have been labeled: 1,2,..,8 and temporary data: w_0,w_1,...,w_7. The branch part of the

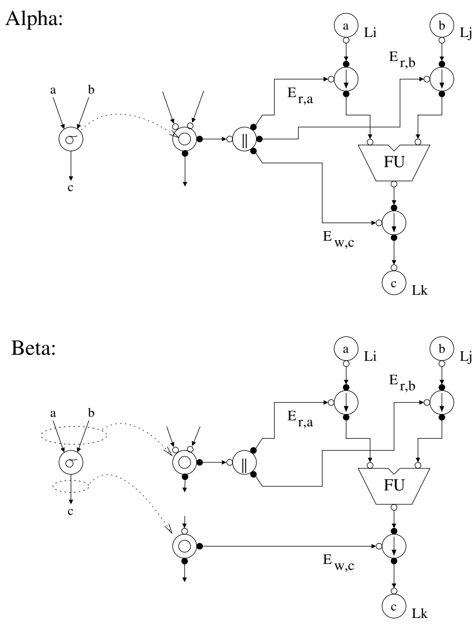


Figure 4.12: Correspondence between CDFG node and asynchronous circuit styles.

FU	σ	t	A	E
ALU	$\{+,-,>\}$	1	1	1
mult	$\{*\}$	2.6	10	13

Table 4.1: Simple example normalized FU library.

CDFG, nodes $\{6,7,8\}$, gives rise to two paths in the schedule. Determined by the execution of node 4, either 6 and then 8, or 7.

The scheduling in Figure 4.14 results in the fastest execution of the CDFG on a datapath containing only one mult and one ALU component.

4.3.1 Datapath with out input/output FU latches (alpha)

The general structure of the asynchronous datapath is shown in Figure 4.15 and it follows the computation model (*alpha*) presented in the previous section. The internal variables (L0...Ln) in our datapath are implemented as latches.

The life time of a variable in this datapath (*alpha*) spans from when the computation producing the variable starts until the variable has been used for the last time including the duration of the last computation.

For our example, the variable lifetime is shown in Figure 4.16 and is generated by the following algorithm: Let Ω be the set of operators $\{\sigma_i\}$, $\sigma_{i,\text{source}} = \{w_j\}$ be

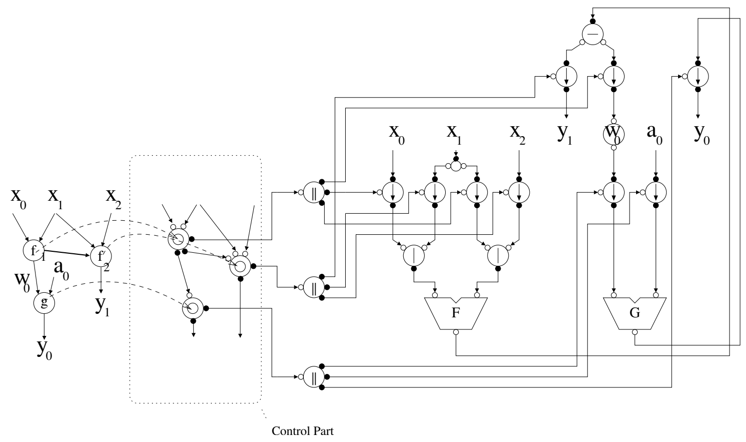


Figure 4.13: Resource-shared asynchronous circuit.

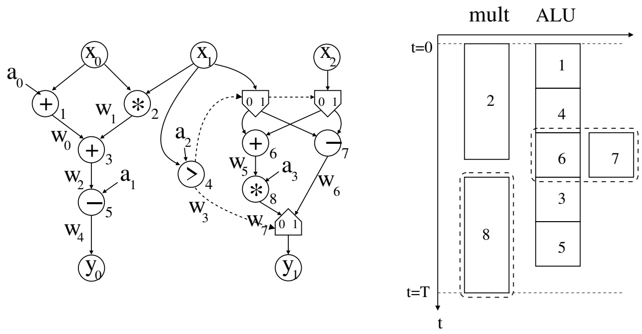


Figure 4.14: (Left) Our example CDFG with labels on temporary data. (Right) Scheduling of our CDFG.

the set of source variables to operator i and let $\sigma_{i,\text{target}} = w_k$ be the target variable. Furthermore let $\sigma_{i,\text{start}}$ be the scheduled start time of operator i and $d_{\text{FU}(i)}$ the delay of the FU σ_i is assigned to. T is the length of the schedule.

Alpha:

Initialization $\forall w_{i,end} = 0$ and $\forall w_{i,start} = T$

Alg For elements $\sigma_i \in \Omega$ {

for elements $w_j \in \sigma_{i,\text{source}}$

if $w_{j,\text{end}} < \sigma_{i,\text{start}} + d_{\text{FU}(i)}$ then $w_{j,\text{end}} = \sigma_{i,\text{start}} + d_{\text{FU}(i)}$

if $w_{k,\text{start}} > \sigma_{i,\text{start}}$ then $w_{k,\text{start}} = \sigma_{i,\text{start}}$ }

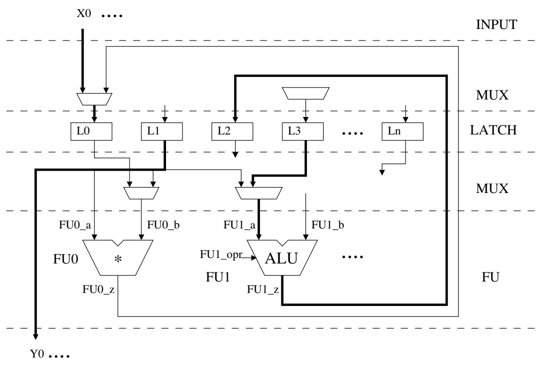


Figure 4.15: General structure of the datapaths without input/output FU latches (*alpha*).

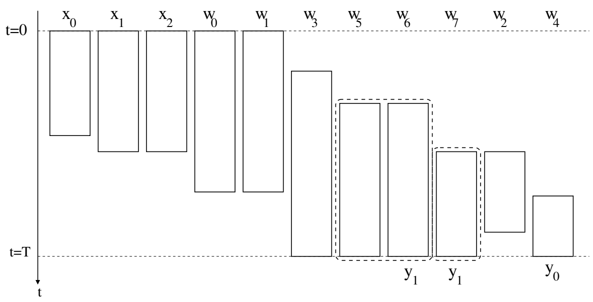


Figure 4.16: Variable lifetime (*alpha*) for our scheduled CDFG.

After we have found the variable time, we need to find the minimum number of latches required and their assignment for the schedule. For this we can use the left-edge algorithm for discrete time [67], if the schedule has been generated through a synchronous method or the left-edge algorithm for continuous time [3], to find the minimum number of latches required in the datapath, which in this case is seven latches. The left-edge algorithm also gives us the variable to latch assignment, shown in Figure 4.17. The conditional part in the variable lifetime algorithms are handled by keeping track of which variables exclude each other, those can be assigned to the same latch. The choice of variable to latch assignment algorithm depends on several factors: a) one might choose an algorithm that considers both the latch area and the multiplexing area [21, 67, 87, 58, 49, 99]. Rearranging the variable to latch assignment could minimize the multiplexing area more than a possible increase in latch area leading to an overall area minimization. b) Another consideration is power consumption. Variables with high data correlation could be grouped together on the same latch leading to a smaller power consumption of the computation [28, 31, 19, 57, 39, 83].

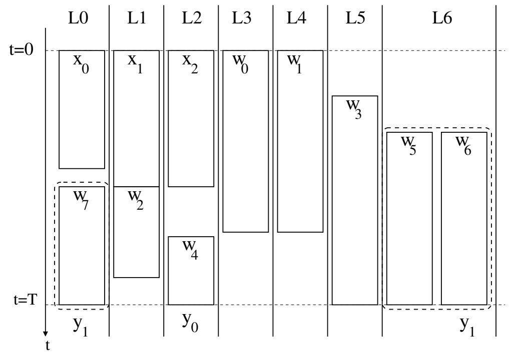


Figure 4.17: Latch assignments (*alpha*) for our scheduled CDFG.

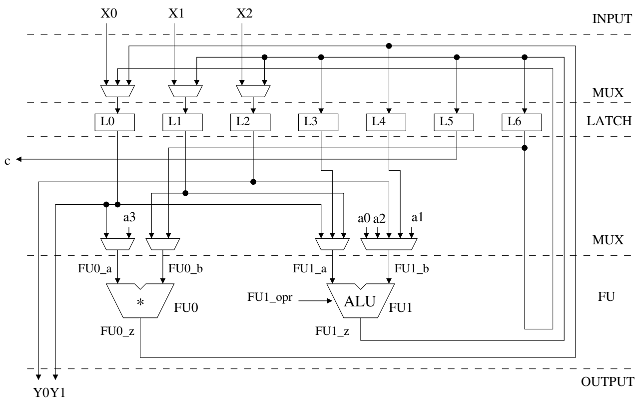


Figure 4.18: Final datapath (*alpha*) for our scheduled CDFG.

With the FU allocation, operator to FU assignment and variable latch assignment the datapath can be constructed by connecting the components through multiplexors. The datapath for our example is shown in Figure 4.18. The controller to this circuit implements the schedule and controls the FUs with the right data at their designated times.

4.3.2 Datapath with input/output FU latches (beta)

The general structure of the datapath with output FU latches is shown in Figure 4.19 and it follows the computation model (*beta*) presented in the previous section. The internal variables (L0...Ln) in our datapath are implemented as latches. The functional units (FU0...FUm) are implemented as independent processing units, with local control, wrapping the computation part with latches on both input and output ports.

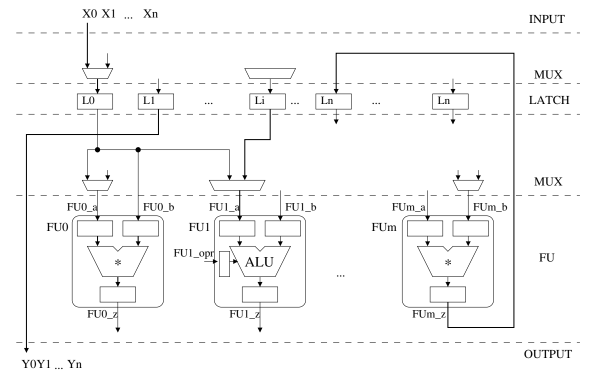


Figure 4.19: General structure of datapath with input/output FU latches (*beta*).

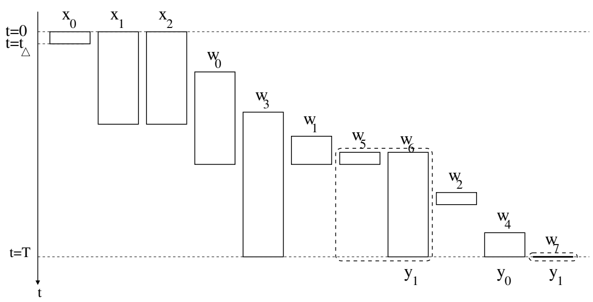


Figure 4.20: Variable lifetime (*beta*) for our scheduled CDFG.

All the latches are implemented as normally opaque latches which gives us a number of advantages:

1. Normally opaque latches on the input ports of the FUs ensures that changing data in the variables does not lead to unnecessary switching activity and power consumption inside FUs which are supposed to be idle.
2. Normally opaque latches on the output port of the FUs ensures that before presenting the result to the rest of the circuit, we let the combinatorial circuit settle (assuming single-rail).
3. Normally opaque latches to hold variables, efficiently reduces the combinatorial depth in the data routing part reducing switching activity and power consumption.

To compute the variable life times we have to look at how long a variable needs to be held in an internal variable. Since our FUs have input latches we only need to hold the variable until it has been read for the last time, at the start of the last computation. This reduces the variable life time requirements, leading to a possible

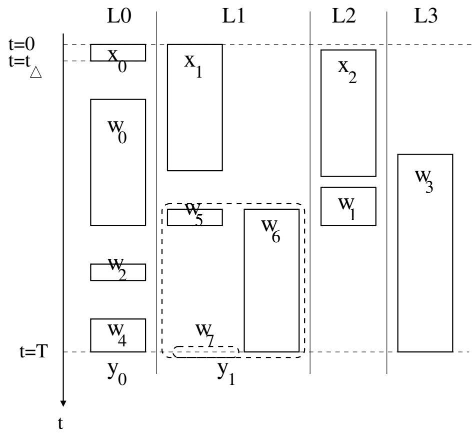


Figure 4.21: Latch assignments (*beta*) for our scheduled CDFG.

reduction in the number of variables needed. We set the overhead for reading and writing a result to a variable latch to be t_Δ, which is added to the variable lifetime.

For our example, the variable lifetime using this approach is shown in Figure 4.20 and is generated by the following algorithm: Let Ω be the set of operators $\{\sigma_i\}$, $\sigma_{i,\text{source}} = \{w_j\}$ be the set of source variables to operator i and let $\sigma_{i,\text{target}} = w_k$ be the target variable. Furthermore let $\sigma_{i,\text{start}}$ be the scheduled start time of operator i and $d_{\text{FU}(i)}$ the delay of the FU σ_i is assigned to. T is the length of the schedule and ΔT is the time overhead of loading and storing data to the latches.

Beta:

Initialization $\forall w_{i,end} = 0$ and $\forall w_{i,start} = T$

Alg For elements $\sigma_i \in \Omega$ {

for elements $w_j \in \sigma_{i,\text{source}}$

if $w_{j,\text{end}} < \sigma_{i,\text{start}} + \Delta T$ then $w_{j,\text{end}} = \sigma_{i,\text{start}} + \Delta T$

if $w_{k,\text{start}} > \sigma_{i,\text{start}} + d_{\text{FU}(i)}$ then $w_{k,\text{start}} = \sigma_{i,\text{start}} + d_{\text{FU}(i)}$ }

The minimum number of latches required in the datapath, given by the left-edge algorithm is in this case is four latches and the variable-to-latch assignment is shown in Figure 4.21. Also here several latch assignment algorithms can be used.

With the FU allocation, operator to FU assignment and variable latch assignment, the datapath can be constructed by connecting the components through multiplexors. The datapath for our example is shown in Figure 4.22.

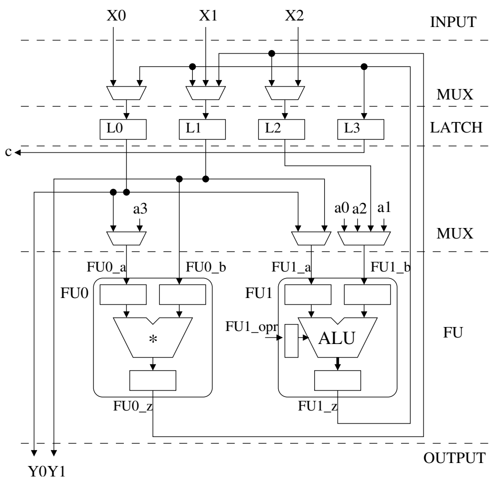


Figure 4.22: Final datapath (*beta*) for our scheduled CDFG.

4.3.3 Datapath with mixed input/output FU latches (gamma)

The general structure for the datapath with mixed input/output functional unit latches is a mix of the two previous models. The internal variables (L0...Ln) in our datapath are implemented as latches. The functional units (FU0...FUm) are implemented as a mixed of independent processing units and as regular functional units.

Computing the variable life times is a mix of the two previous approaches; the start time follows the model corresponding to the type (*alpha* or *beta*) functional unit it is produced by and the end-time follows the model corresponding to the type of functional unit it is used by lastly.

For our example, the low-power solution is to enclose the multiplier with input/output FU latches and letting the ALU operate as a standard FU without input/output FU latches. In this way we shield the unit with the largest combinatorial depth. The variable lifetime using this mixed approach is shown in Figure 4.23 and is generated by the following algorithm: Let Ω be the set of operators $\{\sigma_i\}$, $\sigma_{i,\text{source}} = \{w_j\}$ be the set of source variables to operator i and let $\sigma_{i,\text{target}} = w_k$ be the target variable. Furthermore let $\sigma_{i,\text{start}}$ be the scheduled start time of operator i, $d_{\text{FU}(i)}$ the delay of the FU σ_i is assigned to and $\tau_{\text{FU}(i)}$ be the type of FU: α (with out) or β (with FU latches).Parameter T is the length of the schedule and ΔT is the time overhead of loading and storing data to the latches.

Gamma:

Initialization $\forall w_{i,end} = 0$ and $\forall w_{i,start} = T$

Alg For elements $\sigma_i \in \Omega$

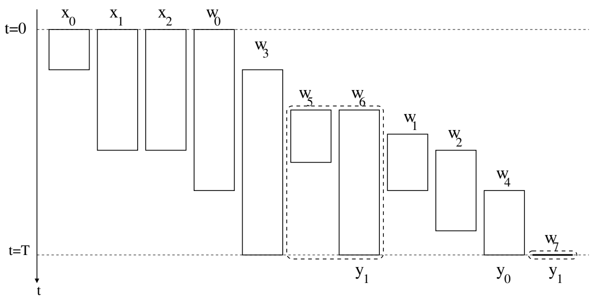


Figure 4.23: Variable lifetime (*gamma*) for our scheduled CDFG.

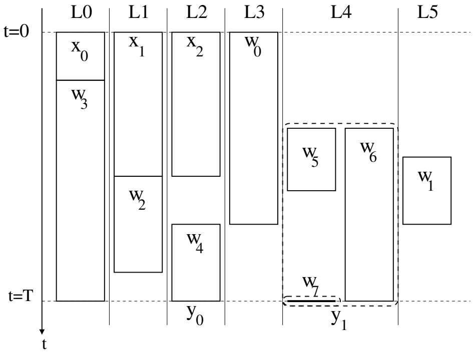


Figure 4.24: Latch assignments (*gamma*) for our scheduled CDFG.

if $\tau_{\mathrm{FU}(i)} = \ beta$ then {

for elements $w_j \in \sigma_{i,\mathrm{source}}$

if $w_{j,\mathrm{end}} < \sigma_{i,\mathrm{start}} + \Delta T$ then $w_{j,\mathrm{end}} = \sigma_{i,\mathrm{start}} + \Delta T$

if $w_{k,\mathrm{start}} > \sigma_{i,\mathrm{start}} + d_{\mathrm{FU}(i)}$ then $w_{k,\mathrm{start}} = \sigma_{i,\mathrm{start}} + d_{\mathrm{FU}(i)}$ }

else {

for elements $w_j \in \sigma_{i,\mathrm{source}}$

if $w_{j,\mathrm{end}} < \sigma_{i,\mathrm{start}} + d_{\mathrm{FU}(i)}$ then $w_{j,\mathrm{end}} = \sigma_{i,\mathrm{start}} + d_{\mathrm{FU}(i)}$

if $w_{k,\mathrm{start}} > \sigma_{i,\mathrm{start}}$ then $w_{k,\mathrm{start}} = \sigma_{i,\mathrm{start}}$ }

The minimum number of latches required in the datapath given by the left-edge algorithm is in this case is six latches and the variable to latch assignment is shown in Figure 4.24. Also here several latch assignment algorithms can be used.

With the FU allocation, operator to FU assignment and variable latch assignment, the datapath can be constructed by connecting the components through multiplexors. The datapath for our example using the mixed approach is shown in Figure 4.25.

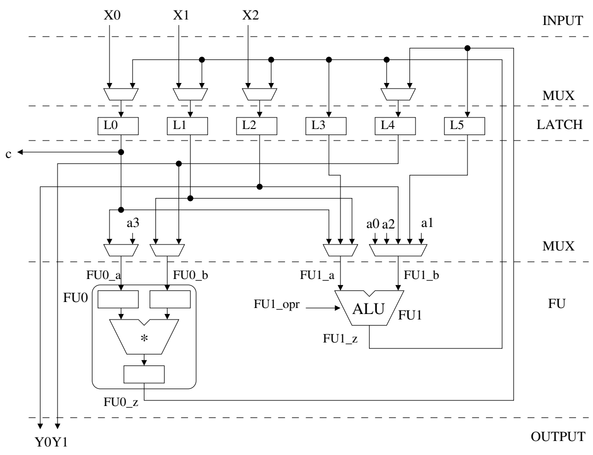


Figure 4.25: Final datapath (*gamma*) for our scheduled CDFG.

4.4 Summary

In this chapter we have looked at two computation models which have different power characteristics but have the same fundamental type of operation and thus can be mixed. The models are capable of implementing any type of schedule, both discrete and continuous and their resemblance to synchronous computation models allows for the used of methods from that domain to be utilized for asynchronous circuit design. Finally, we have looked at the details of datapath synthesis i.e. variable and latch allocation and assignment for all of the computation models.

Chapter 5

Implementation in Balsa

This chapter presents the Balsa implementation templates for generating our asynchronous circuits for all of the computation models. In the previous chapter we have connected traditional behavioral synthesis with asynchronous circuits using our computation model. This chapter deals with the practical implementation of this model, the back-end of our synthesis tool. Figure 5.2 shows the Balsa handshake circuit equivalent to our datapath from Figure 4.22.

5.1 Program structure

The Balsa handshake circuit structure corresponding to our general datapath structure is shown in Figure 5.1. Such a Balsa handshake circuit is built from handshake components which implement the equivalent RTL operations as latching data, multiplexing data, addition etc. Each of these handshake components has its own local asynchronous control to ensure proper asynchronous functionality and to handle the asynchronous handshake communication protocol [92].

Besides these asynchronous handshake components which have their equivalent RTL counter parts, there are the demux components which handles "wire-forks", and more importantly the transfer handshake components connecting the asynchronous controller with the datapath; the latter play the role of event synchronizers, refer to Figure 1.4, controlling the computation. These extra components augments the mux layers with sublayers of demux and transfer components. Notice the mux components implement a merge functionality and is not directly connected to the controller, neither are the latches, demuxes or FUs (except the opr control signal), only the transfer components are connected to the controller. The FUs are autonomous components

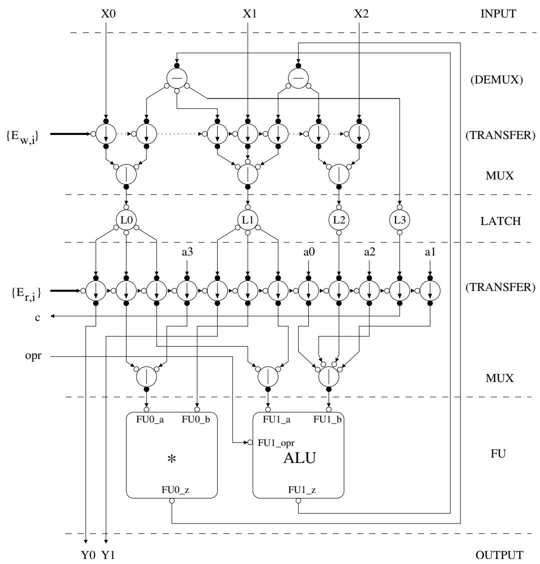


Figure 5.1: Datapath (*beta*) for our scheduled CDFG using Balsa/Tangram handshake components using decoupled functional units.

which start computing when all their input data is present. Using these components and our computation model, there is a one-to-one correspondence between the datapath of Figure 4.22 and Figure 5.1.

In our design we use a bundled data 4-phase protocol where signals contain a 1 bit request and a 1 bit acknowledge wire additional to the data wires. Furthermore, the transfer components degenerate to simple wire connections containing no logic.

The Balsa programs specifying the asynchronous circuit consists of:

FUs Instantiation of the different Balsa FUs used in the design. Each of these descriptions are taken from a FU library of Balsa component descriptions, we have designed for this purpose. The delay, area and power consumptions figures of this library are used by the synthesis algorithm to generate the schedule.

Architecture Balsa implementation of the circuit containing the specification for the control-handshake components, the latch instantiations, and the specification of the routing of data between the variables and the FUs.

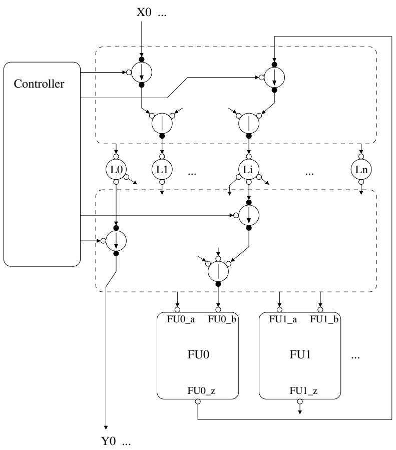


Figure 5.2: Circuit structure using Balsa/Tangram handshake components, corresponding to our datapath (*beta*) structure.

The FUs are implemented using the following Balsa-program structures:

```
procedure FUalpha(inputs a,b,..;
             output z) is
begin
  loop
    select a,b,.... then
      z<-F(a,b,...)
    end
  end
end
```

```
procedure FUbeta(inputs a,b,..;
             output z) is
variable A,B,Z,...
begin
  loop
    a->A || b->B || ...;
    Z:=F(A,B,...) ;
    z<-Z
  end
end
```

where F implements the computation.

The design of the circuits follows the following Balsa-program structure:

```
input [FU_library]
procedure Circuit(inputs X0,X1,...;
                         output Y0,Y1,...) is
variable L0,L1,..,Ln
channel FU0_a,FU0_b,....,FUm_z

begin
  FUj(FUj_a,FUj_b,FUj_z) ||... ||
  [Architecture(X0,X1,..,FUj_a,FUj_b,FUj_z,..,Y0,Y1,...)]
end
```

5.2 Events: using functional units

As an example of how the datapath is constructed using the Balsa-language consider the assignment of a subtraction operator to an ALU designated FU1. This subtraction operator has inputs w_0 w_1 and output w_2 ($w_2 = w_0 - w_1$), assigned to variables L0 L1 and L2 respectively. Starting the computation is performed by executing the following parallel Balsa-statement:

```
FU1_opr<-ALU_sub || FU1_a<-L0 || FU1_b<-L1
```

This set of parallel channel assignment statements tells FU1 to perform a subtraction, and to use the data of L0 and L1. The result w_2 of the computation is written to L2 using the following Balsa-statement:

```
FU1_z->L2
```

Both statements will synchronize the controller with the ALU using the transfer components and implements the process illustrated on Figure 4.9. For the *alpha* type FU, the read and write events need to happen in the same statement:

```
FU1_opr<-ALU_sub || FU1_a<-L0 || FU1_b<-L1 || FU1_z->L2
```

meaning parallel events need to happen in parallel threads. For the *beta* type FU, the read and write events does not need to happen in the same statement, but can happen at separate time-positions:

```
FU1_opr<-ALU_sub || FU1_a<-L0 || FU1_b<-L1 ;
...;
FU1_z->L2
```

in fact parallelism can be implemented in a single thread.

The reading of input X0 to internal variables L0 and placing of results in internal variables L3 on output channels Y0 is executed in a similar way:

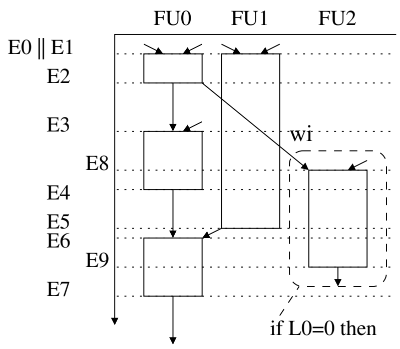


Figure 5.3: Schedule showing all the different types of relative synchronization events.

```
X0->L0 || Y0<-L3
```

These Balsa-statements: i) starting a computation, ii) writing the result of computation or iii) communicating with the outside world, implement the events described in section 4.1.

5.3 Implementing a schedule

A schedule consists of a series of such time ordered events and the architecture part is a series of corresponding Balsa-statements. Consider the example schedule in Figure 5.3, which is different from the running CDFG example. It is illustrating all the different types of relative synchronization events required to implement any schedule. For the construction of the schedule we need to distinguish between the FU types:

alpha The handshakes are active for the duration of the computation on the functional units.

beta The handshakes are active only for the points in time where data is moved to and from the functional units.

Let us begin with the *beta* type, as it is the simplest. Consider events E0..E7, in Figure 5.3 the non-conditional part. These form a sequence of events with E0 and E1 in parallel and the rest ordered E2,..,E7, which can be implemented by the following program fragment:

```
loop
  E0 || E1 ; E2 ; E3 ; E4 ; E5 ; E6 ; E7
end
```

This program fragment is a repetitive execution of the schedule. When we include the conditional execution of the operator on FU2 represented by events: E8 and E9, the Balsa-program fragment becomes:

```
loop
  E0 || E1 ; E2 ; E3 ; if L0=0 then E8 end ;
  E4 ; E5 ; E6 ; if L0=0 then E9 end ; E7
end
```

Notice the single thread of event statements implement the parallel schedule of Figure 5.3.

Next, we will continue with the *alpha* type. As the handshakes now cover durations the single sequence of ordered events only apply to a single thread on a single functional unit. In principle this means there need to be as many parallel threads as there are functional units, communicating to each other using channels. However usually, and so is the case for our example, it is possible to merge some threads, elliminating communication overhead. Here the threads of FU0 and FU1 can be merged, leaving only a separate thread for FU2, the conditional part. Lets start with the unconditional part:

```
loop
  [ E0 || E2 ; E3 || E4 ] || [E1 || E5] ; E6 ; E7
end
```

The parallel operator is here used to merge the first part of the thread for FU0: [E0 || E2 ; E3 || E4] with the thread for FU1: [E1 || E5] , after these the thread for FU0 is continued.

To include the conditional part, in the form of a separate thread, we also need to implement the transfer of the intermediate data in Li over a channel w to Lj, where Li is used exclusively in the thread corresponding to FU0 and Lj is used exclusively in the thread corresponding to FU2. The complete schedule becomes:

```
loop
  [[ E0 || E2 ; Li->w || E3 || E4 ] || [E1 || E5] ; E6 ; E7 ] ||
  [ w->Lj ; if L0=0 then E8 || E9]
end
```

Channel communication represents an area- and time-overhead and as the merging of threads saves channel communications between them, this overhead is reduced. The parallel nature also requires the exclusiveness for variables, if this cannot be guaranteed by the variable to latch assignment, synchronizer channels between threads are required to introduce this exclusiveness.

The condition for parts of two threads to be merged, is if the one part (the sequence of events) is fully inclosed by or executed in serial by the read and write events of the other part. We will denote the read and write events of executing an operation on a FU for an execute interval: $I_{FU,number}$. In Figure 5.4 is shown the threads of the FUs and below the intervals are labelled: $I_{0,0}, I_{0,1} \dots I_{2,0}$. If we generate a thread graph (I, S, D) where the nodes are the execute intervals $I_{FU,number}$ and where the directed arcs $(X \rightarrow Y$ between two nodes are: (i) if interval X can be fully inclosed in interval Y. These arcs are shown as solid arcs and (ii) if two intervals

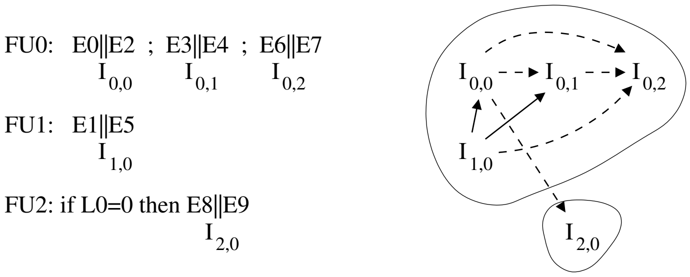


Figure 5.4: Generation of threads: (Left) sequence of events for all FUs and labelling of intervalls. (Right) Clique-partitioning of thread-graph. Each clique becomes a thread.

are fully disjoint and Y is executed after X. These arcs are shown as dotted arcs. Then the optimal merging of all the threads is a clique partitioning of this graph. The thread graphs and the clique partitioning of this graph is shown in Figure 5.4. We use a simple greedy approach for clique partitioning of the thread graph. The resulting partitioning corresponds to our example.

The *gamma* model is treated first as the *alpha* model for the functional units following that model. Then the events for the functional units following the *beta* model are inserted into the appropriate positions.

5.4 Implementing the architecture

Let us look at the datapath being generated by this approach. Consider the following sequence:

```
L0->FU0_a;  -- E0
...;
L1->FU0_a   -- E1
```

giving rise to the circuit shown in Figure 5.5. Each of these events will lead to a transfer component activated by E0 and E1 respectively, followed by a merge component on the input of FU0_a, i.e., implementing a multiplexing of the wires from L0 and L1 to FU0_a, the same goes in the reverse direction.

The architecture part of the program consists of two parts: (i) shared functions (ii) schedule. The shared functions implements the event of the schedule which appear in the schedule more than once. In the schedule below:

```
procedure Architecture(..) is

begin -- schedule
  loop
    ...;
```

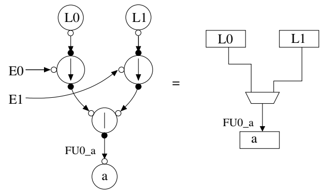


Figure 5.5: Handshake implementation of routing and corresponding datapath.

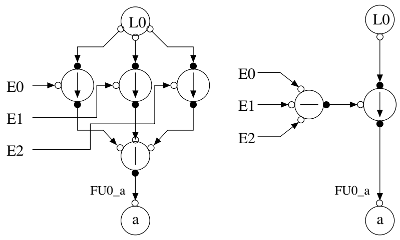


Figure 5.6: Repeated use of hardware with out shared construct (left) and with shared construct (right).

```
    FU1_opr<-ALU_add || FU1_a<-L0 ||  FU1_b<-L2 || FU1_z->L1 ;
    FU1_opr<-ALU_sub || FU1_a<-L0 ||  FU1_b<-L1 || FU1_z->L2 ;
    FU1_opr<-ALU_sub || FU1_a<-L0 ||  FU1_b<-L1 || FU1_z->L3 ;
    ...
  end
end
```

There are several events which reappear e.g.. the FU1_a<-L0 event which appears three times. In the following schedule:

```
procedure Architecture(..) is

shared S0 is
begin
  FU1_a<-L0
end

shared S1 is
begin
  FU1_b<-L1
end

shared S2 is
begin
```

```
  FU1_opr<-ALU_sub || S0() || S1()
end

begin -- schedule
  loop
    ...;
    FU1_opr<-ALU_add || S0() ||  FU1_b<-L2 || FU1_z->L1 ;
    S2() || FU1_z->L2 ;
    S2() || FU1_z->L3 ;
    ...
  end
end
```

it only appears once and the same for every other assignment. Shown in the form of the S0,S1,S2 constructs. The one-to-one syntax directed compilation approach employed by balsa means that in the first circuit there are three assignments from the same latch to the same port of FU1, as shown on Figure 5.6 (left) but by using the shared construct we can “reuse” the hardware and implement the circuit shown on Figure 5.6(right). This saves hardware as the control handshakes are one bit wide, where as the datapath handshake components are N bit wide. This can be extended to include reducing the control circuit, as shown in the program as the S2 shared construct which implements a group of events, which are used several times.

The structure of balsa circuit implementing the schedule with these shared constructs represents a three, with the loop-body component as the root and the events/transfer components as leafs and with some of the leafs merged together [51].
The full Balsa program (*beta*) of our running example, is shown here:

```
import [balsa.types.basic]
import [FU_types]
import [FU_lib]

procedure EX(input X0,X1,X2:word;
             output Y0,Y1:word) is
  variable L0,L1,L2,L3:word
  channel FU0_a,FU0_b,FU0_z:word
  channel FU1_a,FU1_b,FU1_z:word
  channel FU1_opr:ALU_operation
  constant a0= 255
  constant a1= 255
  constant a2= 255
  constant a3= 255


procedure Ex_architecture(input X0,X1,X2:word;
      input FU0_z,FU1_z:word;
      output FU0_a,FU0_b,FU1_a,FU1_b:word;
      output FU1_opr:ALU_operation;
```

```
      output Y0,Y1:word) is

shared S0 is
begin
  FU0_b<-L1
end

shared S1 is
begin
  FU1_a<-L0
end

shared S2 is
begin
  FU1_opr<-ALU_add
end

shared S3 is
begin
  FU1_z->L0
end

shared S4 is
begin
  FU1_a<-L1
end

shared S5 is
begin
  FU1_b<-L2
end

shared S6 is
begin
  FU1_opr<-ALU_sub
end

shared S7 is
begin
  S1() || S2()
end

shared S8 is
begin
  S4() || S5()
end

begin -- schedule
```

```
  loop
    X0->L0 || X1->L1 || X2->L2 ;
    FU0_a<-L0 || S0() || S7() || FU1_b<-a0 ;
    S3() || S4() || FU1_b<-a2 || FU1_opr<-ALU_les ;
    FU1_z->L3 ;
    if L3=0 then S8() || S2()
    else S8() || S6() end ;
    FU0_z->L2 || FU1_z->L1 ;
    if L3=0 then FU0_a<-a3 || S0()
    end || S5() || S7() ;
    S3() ;
    S1() || FU1_b<-a1 || S6() ;
    if L3=0 then FU0_z->L1
    end || S3() ;
    Y0<-L0 || Y1<-L1
  end
end

begin
  mult(FU0_a,FU0_b,FU0_z) ||
  ALU(FU1_opr,FU1_a,FU1_b,FU1_z) ||
  EX_architecture(X0,X1,X2,FU0_z,FU1_z,FU0_a,
      FU0_b,FU1_a,FU1_b,FU1_opr,Y0,Y1)
end
```

The balsa-circuit generates the datapath shown in Figure 5.1 and the controller shown in Figure 5.7.

5.5 Optimizations

For the *alpha* model it is possible to take advantage of the memory in the functional units to optimize the computation. In the situation where a temporary variable, t_i, in a CDFG, is used directly after it is produced and not required to be stored for later use, we can implement a direct feed-forward from FUi to {FUj...FUk}, as shown in Figure 5.8. If FUi has to start another computation immediately after producing t_i then this optimization should only be implemented if all the target FUs {FUj...FUk} are ready to start when t_i is produced, otherwise FUi will be stalled. Similar feed-forward can be implemented from inputs and/or to outputs of the circuit. The purpose of this optimization is to achieve a reduction in the number of variable latches and circuit speed-up.

In the datapath synthesis algorithm these assignments are identified in the variable lifetime computation and separated from the variable latch assignment. In our example computation no latch reduction is possible using this method. Implementing this optimization in Balsa is straightforward. If the value is used by *one* FU or to *one* output only, we get:

FU to FU: `FUi_z->FUj_a`

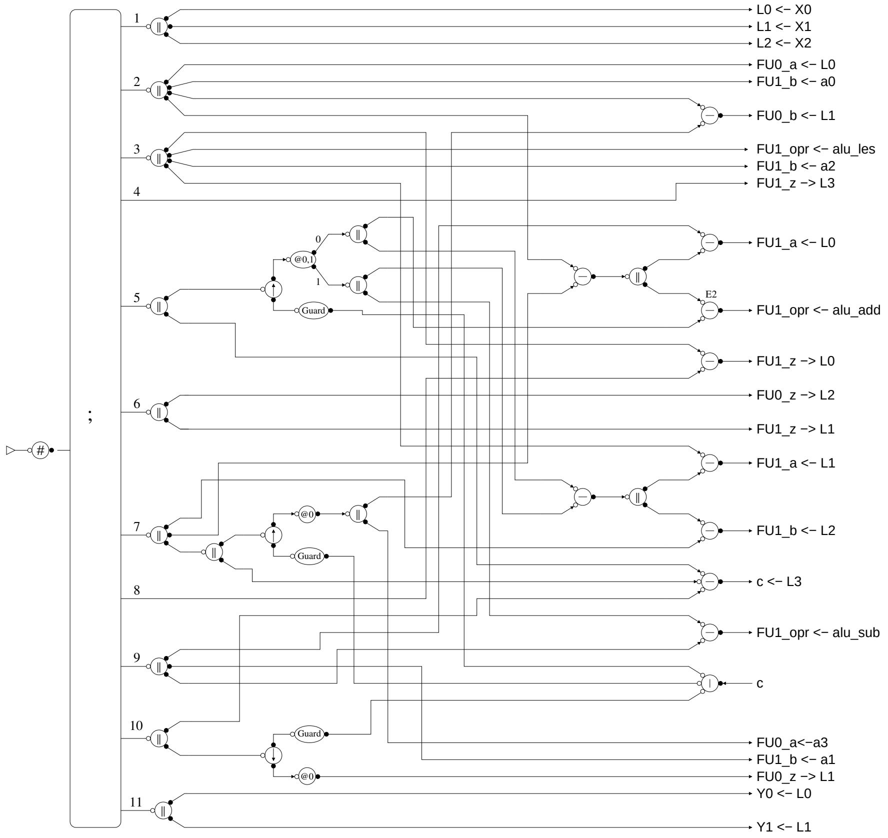


Figure 5.7: Controller to the datapath (*beta*) for our scheduled CDFG using Balsa/Tangram handshake components.

Input to FU: `Xi->FUj_a`

FU to Output: `Yi<-FUj_z`

and assigning a value directly from one FU to multiple FUs are handled using the following Balsa statement:

```
select FUi_z then
  FUj_a<-FUi_z || FUk_a<-FUi_z || ...
end
```

Similar constructs are used for the inputs and outputs. One should note that the implementation of the FUs now require the ability to handle handshakes on both its inputs and outputs simultaneously.

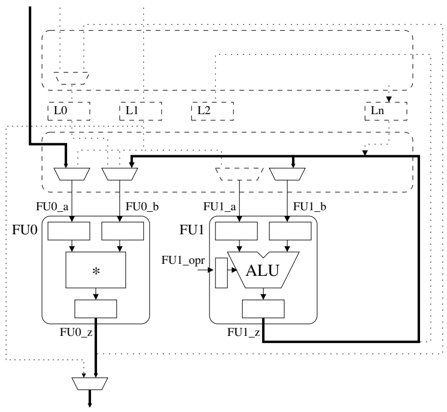


Figure 5.8: General structure of datapaths with speedup/latch reduction paths.

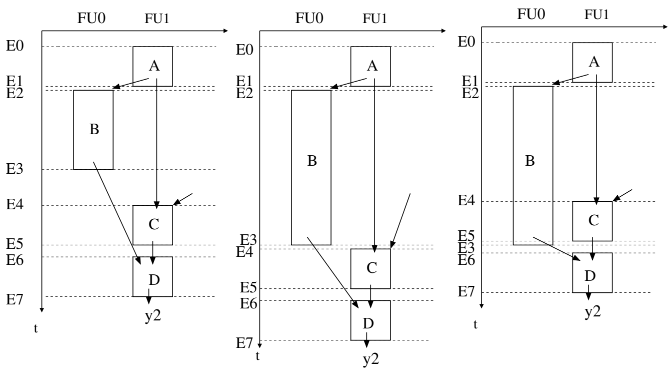


Figure 5.9: Decoupling computation B from computation C to take advantage of the slack time in the schedule.

We can also optimize on the control part, this applies to both models. Consider the schedule of operators: A,B,C,D part of an arbitrary computation, shown in Figure 5.9. In the strict controller/datapath implementation we have:

```
loop
  E0 ; E1 ; E2 ; E3 ; E4 ; E5 ; E6 ; E7
end
```

However we could take advantage of the inherent parallelism of B and C in the CDFG and implement the controller/datapath in the following way:

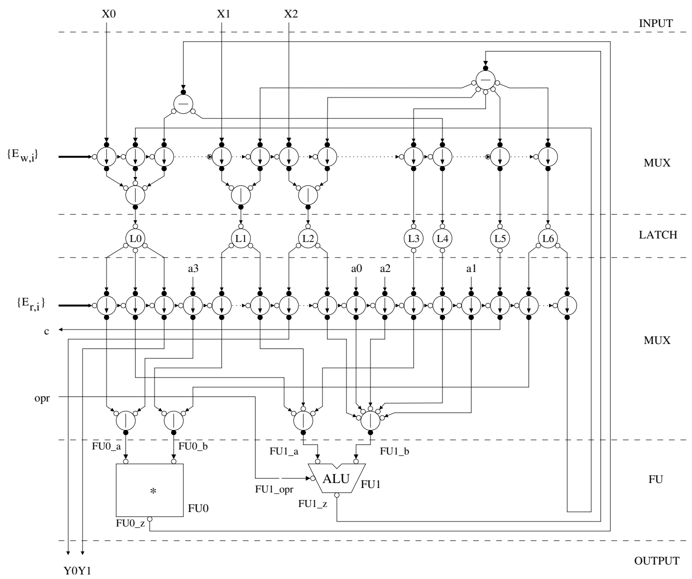


Figure 5.10: Datapath (*alpha*) for our scheduled CDFG using Balsa/Tangram handshake components.

```
loop
  E0 ; E1 ;
  [ E2 ; E3 ] ||
  [ E4 ; E5 ];
  E6 ; E7
end
```

This would still implement the schedule but we have increased the flexibility of the circuit, making the circuit more robust to variable computations times, of e.g.. B, taking advantages of the slack of the non critical paths in the schedule.

5.6 Summary

In this chapter we have presented Balsa program language templates for implementing our asynchronous computation model in the Balsa CAD framework.

The CDFGs used as input to our behavioral synthesis tools could be derived from Balsa it self. In this form of Balsa-to-Balsa compilation one could consider our tools as a way of optimizing a circuit or parts of a circuit at the specification level. This makes it possible to manipulate and manually optimize critical parts of a circuit further than what the tool automatically produces, by manually modifying the output Balsa code.

The Balsa language can be considered a general high-level boundary to the asynchronous world. There is nothing preventing the implementation of other styles of asynchronous circuits, i.e. Burstmode circuits, using the Balsa-language as description language. In fact research of this nature is currently underway. This means the use of the Balsa-language as a back-end represents a variety of implementation styles. However as our templates targets the current one-to-one compilation to handshake-component implementation of Balsa, the “weights” and possibly parts of the implementation templates should be modified to ensure optimal circuit implementation for other implementation styles.

Chapter 6

Algorithms for Behavioral Synthesis

This chapter deals with the fundamental parts of high-level behavioral synthesis: operator scheduling, functional unit allocation and operator to functional unit assignment. We are given a Control Data Flow Graph (CDFG) specifying the behavior/computation which we want to implement onto an Integrated Circuit and we are given a maximum time frame T within which the Integrated Circuit has to perform this computation (e.g.. caused by new data arriving at a frequency of $1/T$, ex. sampled from a sound source).

We will consider behavioral synthesis algorithms targeting a discrete time evolution, for which solutions are relaxed into continuous time. The following algorithm suite have been developed:

- Power aware synchronous synthesis algorithm. This algorithm is a clique heuristic algorithm operating with a time and maximum power per time constraint. This is useful for applications having a power limit e.g. generated by a solar panel. This scheduling algorithm handles CDFG's without repetitive structures.

- Evolutionary synchronous synthesis algorithm and a simulated annealing synchronous synthesis algorithm. These are meta-heuristic algorithms operating with a maximum time constraint. These algorithms only handle DFG graphs.

- Simulated annealing task synthesis algorithm. This algorithm is used to schedule the CDFG where the DFG fragments are scheduled using one of the two

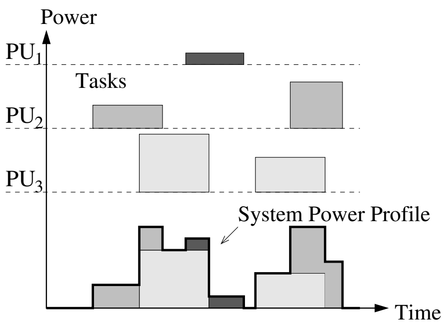


Figure 6.1: Task schedule and the system power profile.

previously mentioned algorithms. This algorithm has not been implemented but the method is outlined.

6.1 Power-aware scheduling

Portable embedded systems face increasing performance demands while running on less power. Therefore, to efficiently use the power available from the power source, task scheduling mechanisms have to take the system power profile into account. Figure 6.1 illustrates a set of scheduled tasks and the resulting system power profile. In low-power or power-aware task scheduling one usually assumes a uniform power profile of the individual tasks, however in reality these individual tasks might have a very irregular power profile. So using the average task power figure in the task scheduling only leads to average system power profile, and the system might have an accumulation of power peaks which would severely violate system power constraints. On the other-hand using the peak power figure would lead to an over-conservative schedule which would comply to the system constraints but would be an inefficient use of system resources.

Another related issue is the non-linear chemical to electrical energy efficiency ratio of batteries which depends strongly on the current profile of the application [102, 5]. Here there are two contributing factors: (1) If the peak-current exceeds a maximum-threshold the life-time starts dropping dramatically. (2) A large current variation also leads to reduction in battery life-time. These factors are more dominant on batteries of low quality. Furthermore there might be a maximum power available to the task restricted by e.g. a solar panel providing the power to the circuit.

Altogether our goal is to synthesize these critical tasks as digital circuits, with a static schedule having an uniform power profile. In this section we present a heuristic synthesis algorithm which solves: (i) scheduling, (ii) allocation and (iii) assignment simultaneously under both a time and power constraint. These 3 tasks are traditionally solved separately which is suboptimal as these typically interfere with each-other.

6.1.1 Problem formulation

The hardware behavioral (time-constrained and power-constrained) synthesis problem, given a non-repetitive CDFG, time constraint T and a maximum energy per time-slot constraint $E_<$, consists of the following subproblems:

Scheduling Determine the schedule ϕ specifying the start time k_i for each operation v_i ($k_i = \phi(v_i)$) such that: *(i)* no precedence constraint is violated: $k_i \geq t_r + d_r, t_r = \phi(v_r), \forall i, r : (v_i, v_r)$, which are connected in the CDFG, such that all operations are completed within the time frame T. *(ii)* no power constraint is violated: $E_k \leq E_<, \forall k = [0..T]$, where $E_k = \sum e_i, \forall i : (v_i)$ which are executing in control-step k.

Allocation Specify which j and how many N_j functional unit instances are required selecting from the provided hardware library R.

Assignment (Operator Binding) Provide a mapping $\alpha : V \rightarrow R$, from each operation v_i to a specific functional unit $\alpha(v_i) = j \in R$. The assignment specifies the execution delay of the operator $\delta(v_i) = d_i$ and the energy consumption per time-slot of the operator $\epsilon(v_i) = e_i$.

We will solve these subproblems simultaneously targeting minimimal the area cost (6.1):

$$cost_\phi = \sum_{j \in R} \left[\omega(j) \times N_j(\sigma)\right], \tag{6.1}$$

where $\omega(j)$ is the area cost of FU j, $N_j(\sigma)$ the required number of these for the schedule.

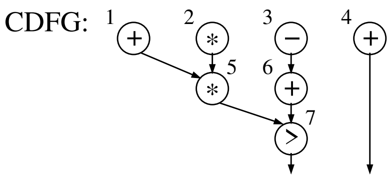
6.1.2 Power heuristic scheduling

In traditional time constrained synthesis the two heuristic low complexity algorithms; **ASAP** and **ALAP** are used to bound the solution space. In Figure 6.2 is shown an example CDFG and its corresponding **ASAP** schedule, where we have assumed all operations, without loss of generality, are executed in one time-slot. In this section we use a different example CDFG, than our running example 2.2, as this new simpler CDFG exemplifies the power variation we want to emphasize for this synthesis method, unlike our familiar CDFG used elsewhere in this thesis.

In the following we present a heuristic algorithm, **PASAP**, which given a power constraint generates a schedule. This algorithm plays the same role as **ASAP** and is being used in our main algorithm to heuristically bound the minimum time separation between two operators, ensuring all CDFG precedence constraints are satisfied

FU	σ	Delay	Area	Energy/time-slot
add	$\{+\}$	1	1	1
ALU	$\{+,-,>\}$	1	1.5	1
mul	$\{*\}$	1	4	3

Table 6.1: Simple example FU library, used for the example only.



t	ASAP	E
0	1 $+$ 2 $*$ 3 $-$ 4 $+$	6
1	5 $*$ 6 $+$	4
2	7 $>$	1

Figure 6.2: Example CDFG and its ASAP schedule

together with the power constraint. The **PASAP** schedule is a"stretched" **ASAP** schedule. "Stretched" to fit the power constraint i.e. the operators are scheduled as fast as possible, but only if there is power available meaning some operators will be delayed additional time-slots.

PASAP ($E_<$):

Initialize: Schedule source start-time to zero and initialize the execution offset o_i (time-steps) to zero for all operators.

step 1: Pick an unscheduled operator v_i

step 2: If v_i has unscheduled predecessors, goto 4.

step 3: If there is power available in the execution time interval $[(t_i + o_i)..(t_i + o_i + d_i)]$, where d_i is the execution delay of v_i and $t_i = \max\{t_j + d_j\}\ \forall v_j \to v_i$, is the earliest start time, otherwise increase o_i by one.

step 4: If unscheduled operators, goto step 1.

For construction of our **PASAP** schedule we use the simplistic functional unit (FU) library shown in table 6.1. In Figure 6.3 is shown the **PASAP** schedule for our example CDFG, here we have set a power limit of $E_< = 3$, which we keep for this example. The algorithm starts in time-slot one and tries to fill it up with operations:

t	PASAP	E	PALAP	E
0	1(+) 3(−) 4(+)	3	2(∗)	3
1	2(∗)	3	3(−) 1(+)	2
2	5(∗)	3	5(∗)	3
3	6(+)	1	6(+)	1
4	7(>)	1	4(+) 7(>)	2

Figure 6.3: The PASAP and PALAP schedules of our example CDFG, both with $E_{<}$=3.

we start by scheduling v_1, which prevents us from scheduling v_2 as this would violate the power constraint. But we can continue to schedule v_3 and v_4. In the next time-slot we have v_2 ready, which is the only one for which there is power available and the algorithm continues. The total **PASAP** schedule takes 5 time-slots to complete as opposed to only 3 time-slots of the **ASAP** schedule. The same algorithm can run backwards which we denote **PALAP**.

Obviously there are many ways of selecting which operators to "pack" into time-slots and it is a hard problem to find the optimal combination i.e.. the solution that results in the schedule using the least amount of time. Here we have simply chosen the order of which they appear in the CDFG. In this way **PASAP** cannot be compared to **ASAP**.

6.1.3 Power and time constrained synthesis

In Figure 6.4 we have re-shown our example CDFG as well as a non-power constrained schedule with a time constraint of T=5 time-slots. Here the partial clique partitioning algorithm in [58] is capable of constructing a schedule and an FU allocation using only one ALU and one mul (the minimal FU-allocation to execute this CDFG no-matter how much time we have available) using a total area cost of 5.5 units. Besides the schedules is shown the total energy consumption for the respective time-slots. Here we note two things: (i) This schedule violates the energy constraint of $E_{<} = 3$ and furthermore (ii) it is very spiky (time-slots 1 and 3). For a power constrained schedule we wish to stay under our constraint and "smoothen-out" the schedule.

As mentioned, our power constrained synthesis algorithm builds upon this algorithm and as in [58] we construct the time-extended compatibility graph, $V1$: Each vertex A_{ijk} represents a possible scheduling, allocation and assignment of operation i on FU type j starting in time-steps k. Each edge $< A_{ijk}, A_{rjt} >$ represents the simultaneously scheduling, allocation and assignment of operator i and r on the same FU instance of type j at times k and t, respectively. We have extended the formulation of a valid $V1$ graph to include power constraints. Thus our allowed vertices (A_{ijk}) are:

i: All operators in the CDFG.

j: The set of FUs where operator i can be executed.

k: The time interval given by $\{t_{\mathbf{PASAP}}, t_{\mathbf{PALAP}}\}$, when operator i is executed on FU j and all other operators are scheduled using delay information from the fastest FU type and power information from the most power hungry FU type.

And the allowed edges,$< A_{ijk}, A_{rjt} >$, are those where there is a dependency in the CDFG, $v_i \rightarrow v_r$, and the execution time of the two operators does not overlap when scheduled on FU_j, as well as it is possible to find a valid PASAP schedule with v_i and v_r scheduled on FU_j at times k and t respectively.

A subgraph of $V1$ which is completely connected by compatibility edges in $V1$ (clique) can be mapped to one FU instance. Then the solution to the synthesis problem with the minimum area and using least interconnect is the problem of finding the **Partial minimal cost clique partitioning of $V1$ which does not violate the power constraint**, where partial refers to a cover containing one-and-only-one vertex for each operator i.

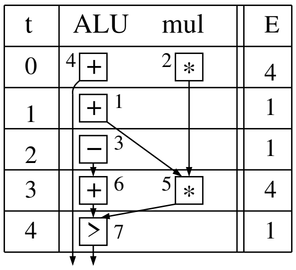


Figure 6.4: CDFG and a non-power schedule with T=5, using only one ALU and one mul with a total area of 5.5.

As in [58] we heuristically solve the clique partitioning problem, through a greedy approach i.e.. evaluate the $V1$ graph and pick a "best" decision which is then scheduled, allocated and assigned. Then this process is repeated until no operators are left. To this end we construct the Mixed-vertex Compatibility Graph ($MCG = (V1, V2, E)$): The $V1$ graph, extended with super-vertices $S_{j,n} \in V2$. The super-vertexes $S_{j,n}$ contain the scheduled, allocated and assigned operators on FU of type j instance n. Initially $|V2| = 0$.

In principle, our algorithm starts with a power and time valid region then aggressively reduces area ensuring the scheduling region stays valid. Our algorithm is as follows:

Initial *Build the MCG.* Here **PASAP** and **PALAP** are used to build the set of allowed vertices and allowed edges, under the power and time constraint.

Step 1 *Pick the best decision.* We select according to maximum clique i.e. find the largest clique A_{ijk} is contained in (a double search of the entire graph) and

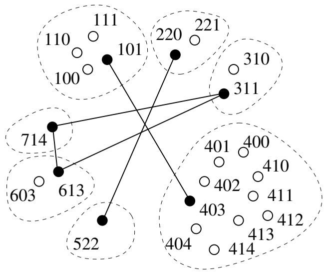


Figure 6.5: Partial-Clique partitioning. Shown are a set of V1 vertices, grouped (by the dotted lines) in operators. The only edges shown are those which are in the maximal clique not violating the power constraint .

compute $cost_{A_{i,j,k}}$ = sum of FU area for maximum clique($A_{i,j,k}$). The selected vertex is merged into an existing super-vertex if it is connected to a super-vertex, otherwise it is made into a new super-vertex.

Step 2 *Transform the MCG in accordance with the decision.* The decision of the previous step has effects on both time and power, again **PASAP** and **PALAP** are used to maintain validity i.e.. ensure the $V1$ graph only contains the set of allowed vertices and allowed edges reflecting the current situation. Furthermore we need to preserve the cliques and disconnect those which no longer form one, refer to [58] for a detailed description.

Step 3 *Ensuring feasibility.* As **PASAP** and **PALAP** are heuristic algorithms they depend on what operators have been scheduled, therefore a sequence of assignments might cause the of deletion unscheduled operators, causing an invalid schedule. The solution is to backtrack one step and lock the start time of all unscheduled operators to the PASAP schedule (which was valid) and then continue, reducing our algorithm to a pure assignment and allocation algorithm from that point on.

Step 4 If any vertices left in $V1$, goto **step 1**.

A comment to step 3, in most cases step 3 will not take effect and the algorithm will continue to the end, however it is possible to construct CDFGs which together with specific constraints causes the algorithm to execute this step. But even if it does, the algorithm has been allowed to operate for some time, during which it has significantly reduced area in comparison with the starting **PASAP** schedule.

In Figure 6.6 we illustrate the construction of a power-constrained schedule using our algorithm and the example CDFG. We use the same time constraint T=5 and power constraint $E_< = 3$ as in Figure 6.3. The onset of the algorithm is the construction of the **PASAP** and **PALAP** schedules, shown in Figure 6.3 and requiring at least 5 time-steps for our power constraint, which generates the scheduling intervals

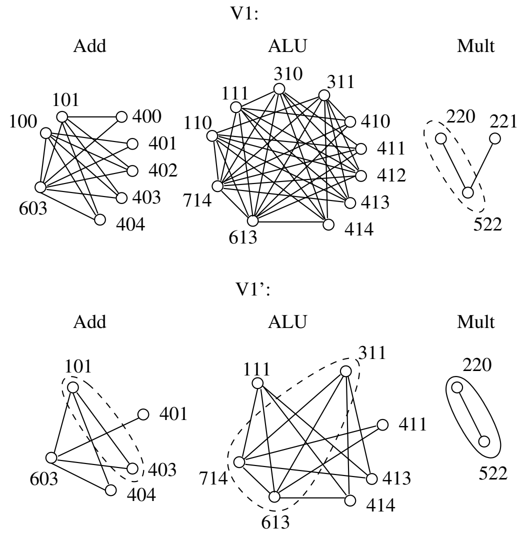


Figure 6.6: CDFG and the construction of the power constrained solution (T=5, $E_{<}$=3).

for our operators. Using the scheduling intervals and our FU-library, shown in table 6.1, we generate the $V1$ graph, shown in Figure 6.6. Initially the algorithms creates a super-vertex of the multiplier operation v_2 scheduled on Mul in time-slot 0, then it merges v_5 scheduled on Mul in time-slot 2 in to it, these are shown enclosed in the dotted ellipse.

The selection of v_2 scheduled on Mul in time-step 0 has consequences in the form of the **PASAP** and **PALAP** algorithms deleting the nodes:$\{100, 400, 110, 310, 410\}$ to maintain the V1 graph in a feasible state. Operation $\{221\}$ is deleted as v_2 now has been scheduled. Merging v_5 scheduled on Mul in time-slot 2, similarly removes operations $\{402, 412\}$ and we arrive at the $V1'$ graph shown in Figure 6.6, with the super-vertex enclosed in the solid ellipse.

As it turns out the $V1'$ graph no-longer contains vertices (i.e.. cliques) which together with the super-vertices can violate the power constraint. Meaning the subsequent execution of the **PASAP** and **PALAP** algorithms in principle reduces to execution of the **ASAP** and **ALAP** algorithms i.e.. the remaining part of the algorithms executes as the original algorithm in [58].

The final schedule, allocation and assignment corresponding is shown in Figure 6.7, requiring one add, one ALU and one mul, using a total area cost of 6.5 units. Alongside the schedule is shown the power consumption in each time-slot, where we now no-longer have a power violation as well as less spikes. We notice the price for the power constrained schedule compared with the non-power constrained schedule (using the same time-frame) is an extra adder, a relative area increase of 18 percent.

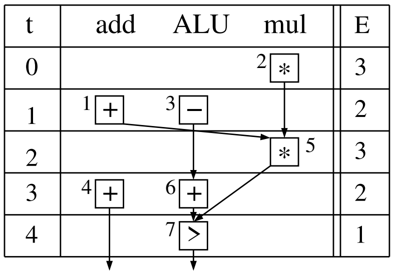


Figure 6.7: Solution (T=5, $E_{<}$=3) using one add, one ALU and one mul, using a total area of 6.5 .

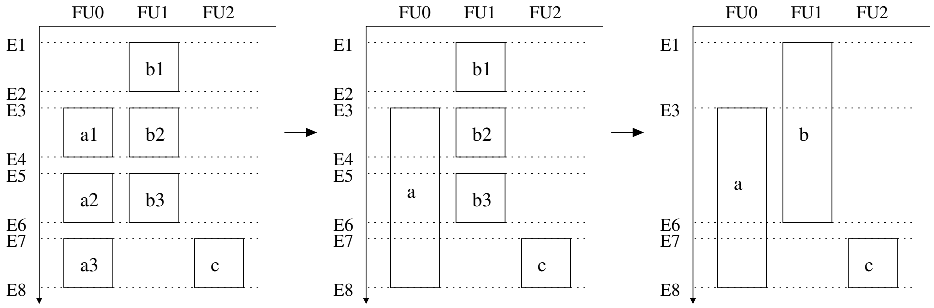


Figure 6.8: Creating multi-cycle operations from single-cycle operations maintaining the global time-line, which prohibits operation "sliding".

6.2 Implementing synchronous power aware schedules in asynchronous circuits

There is a potential danger of violating the power constraint when relaxing a synchronous power aware schedule to continous time and implementing it in an asynchronous circuit, as the synchronous synchronization is removed.

If we restrict our selves to circuits generated by the *beta* model without the optimizations. Or restrict our selves to the circuits generated by the *alpha* model whose threads can be merged into a single main thread. Then we will show there is no power constraint violation relaxing synchronous power aware schedules in asynchronous circuits using our templates.

Let us assume a schedule consisting of single-cycle operations. Then in each control-step there is a set of parallel read events for all operations starting in this cycle, sequenced by, a set of parallel write events for the same operations. This is sequenced by the next cycle. Therefore if the synchronous schedule upholds the power constraint in each cycle, so does this asynchronous circuit.

For multi-cycle operations the picture is a little more complicated, however the same principle applies. First consider the multicycle operation as a sequence of single-cycle operations, as shown in Figure 6.8 (left), the first case. In this picture there is no power constraint violation. Removing the middle synchronization events does not change anything as the start and end of the multi-cycle operation, in the second case, is sequenced now by a series of single cycle operations in between. And in the final case the start and end of all operations is locked on to the global time-line. If we assume a operation has "slided" into violating the power constraint it would have violated the global time-line-sequencing of operations. With respect to the global time-line-sequencing, the *alpha* model, whose threads can be merged into a single main thread, behaves identically as the *beta* model.

6.3 Simulated annealing and evolutionary algorithm

In this section we investigate two meta-heuristic algorithms for solving the behavioral synthesis problem: (i) Simulated annealing and (ii) evolutionary algorithms [78, 42, 79, 66, 43, 32, 52]. Meta-heuristic algorithms are interesting in this context as large DFGs can be scheduled with fast run-times. Furthermore they are easily be stopped if *the* optimal solution is not required to be found, but just a solution which falls within the area requirement. The power-constraint has not yet been implemented into these algorithms.

For these algorithms we target DFG fragments to be scheduled and a time-constraint which specifies the maximum amount of control steps allowed for the execution of the DFG fragment. The DFGs considered here are acyclic directed graph with vertices σ_i, representing the operators to be executed, and edges $\sigma_i \rightarrow \sigma_l$, specifying the order in which they have to be executed for the computation to be correct (σ_i has to be executed before σ_l). The DFG is augmented with a source (connecting to inputs, I) and a target vertex (connecting from outputs, O). To execute operations we use the same resource library of functional units, defined in table 6.2.

With the hard time frame constraint we need to find schedule in which to execute the operations in the DFG onto some FUs such that we finish all operators before the time frame T (without violating their dependencies) and at the same time minimize the area. This involves trade-offs between scheduling e.g. many $\{+, -, >\}$ operations in parallel (requiring more "cheap" ALUs), to serialize more $\{*\}$ operations (requiring fewer "expensive" mul1), as well as tradeoffs between different "subtypes" of FUs (fast or slow). All this depends strongly on the specific DFG and the time frame T we have available.

6.3.1 Problem formulation

First, we formulate the behavioral synthesis problem as an ILP problem. We have a DFG with operators σ_i $i = 1 \ldots n$ and dependencies $\sigma_i \rightarrow \sigma_l$, a resource library with functional units of type FU_j $j = 1 \ldots m$ having a silicon area w_j. And a time interval $k = 1 \ldots T$ giving for each operator σ_i a time interval where it can be scheduled: $S_i \ldots L_i$. We want to minimize the used silicon area. Let us start by introducing the variables in our formulation:

x : Let $x_{i,j,k}$ be a $0, 1$ integer variable associated with the operator σ_i: $x_{i,j,k} = 1$ if σ_i is scheduled to start in time-step k assigned to execute on FU_j and $x_{i,j,k} = 0$ otherwise.

N : Let N_j be an integer variable which denotes the number of functional units of type FU_j we will allocate on our IC.

The objective function is:

$$\text{minimize } A = \sum_{j=1}^{m} w_j * N_j \tag{6.2}$$

Subject to

$$\begin{aligned}
\sum_{k=S_i}^{L_i} \sum_{j=1}^{m} x_{i,j,k} &= 1, \text{ for all } i && (6.3)\\
\sum_{k=S_l}^{L_l} \sum_{j=1}^{m} k \times x_{l,j,k} &- \\
\sum_{k=S_i}^{L_i} \sum_{j=1}^{m} (k - d_j) \times x_{i,j,k} &\geq 0, \text{ for all } \sigma_i \rightarrow \sigma_l && (6.4)\\
N_j - \sum_{i=1}^{n} \sum_{p=0}^{d_j - 1} x_{i,j,k-p} &\geq 0, \text{ for all } j, k && (6.5)\\
E_< - \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{p=0}^{d_j - 1} e_j x_{i,j,k-p} &\geq 0, \text{ for all } k && (6.6)
\end{aligned}$$

The objective function (equ. 6.2) states we want to minimize the total used silicon area and sums over all functional unit types and for each multiplies its area by the number required for the schedule. The first constraint (equ. 6.3) simply states that all operators must be scheduled to start in some time step and on some FU_j. The second constraint (equ. 6.4) specifies that for each DFG dependency $\sigma_i \rightarrow \sigma_l$ operator l can only start after operator i finishes $t_l \geq d_j + t_i$ (which depends on which FU i is scheduled on). The thierd constraint (equ. 6.5) states a FU can only execute one operation at a time. The final constraint (equ. 6.6) ensures that there nowhere is used more power than availeble. This last constraint will be ignored in the following.

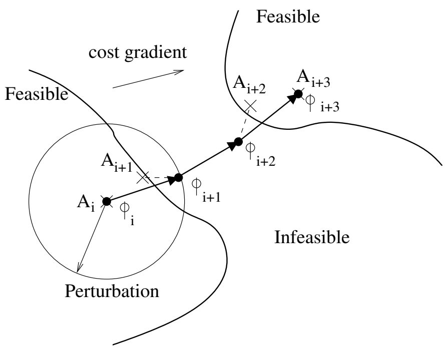


Figure 6.9: Crossing from one island of the solution space to another by keeping the infeasible solutions, when the perturbation is smaller than the minimum required distance. The sequence of ϕ_j's indicated by the dots are the actual solutions and the sequence of $\mathcal{F}(\phi_j) = A_j$ indicated by the crosses, correspond to the feasible solutions the cost area function is computed from.

6.3.2 Representation and feasibility

We use a solution vector containing n tuples (one for each operator), consisting of the pair (k_i, j_i) where k_i is the time step, where operator i starts and j_i is the FU type to execute it on ($k_i \in S_i \dots L_i$ and $j : \sigma_i \in FU_j$). Let the schedule be defined by:

$$\phi = [(k_1, j_1), (k_2, j_2), \dots, (k_n, j_n)]$$

In both simulated annealing and evolutionary algorithms we will likely produce (and start with) solutions which are infeasible. Where infeasible means we are violating DFG dependencies, therefore we need to make the solution feasible $\phi \to \phi'$.

We also use this feasibility algorithm to allow for easy crossing over regions of infeasible solutions, as illustrated on Figure 6.9. We keep the infeasible solution but compute the cost of this infeasible solution by making the solution feasible and then compute the cost of this solution. This requires however that the feasibility algorithm is deterministic, such that the best solution (feasible) can be regenerated from a possible infeasible best solution. This is a better solution than working with a penalty function or removing the infeasible solutions.

First, let us revisit the **ASAP** algorithm. Before the algorithm starts assume we assign an operator σ_i to time step within $t_i \in S_i \dots L_i$ and with j_i equal to the fastest FU_j. The output is the earliest time S'_l the other operators σ_l can be scheduled with σ_i is scheduled in time step k_i. Only successors to σ_i are affected $S_l \leq S'_l$.

Critical for this to be of any use is $S'_l \leq L_l \, \forall \, l$: Assume we at some point get $S'_l > L_l$ after assigning operator r to time step t_r ($\in S_r \dots L_r, S_r \leq L_r$). Let p be the longest path $\sigma_r \to \sigma_l$ and q the longest path $\sigma_l \to \sigma_r$ (going 'backwards'): $S'_l \geq t_r + |p|$ and $L_r \leq L_l - |q|$. Since the DFG is acyclic $|p| = |q|$, so $S'_l \geq t_r + |p|$

and $L_r + |p| \leq L_l$, therefore if $S'_l > L_l \Leftrightarrow t_r + |p| > L_r + |p|$ or $t_r > L_r$, which is a contradiction.

The same applies to the **ALAP** algorithm and by running both algorithms in succession, we reduce the time intervals for all other operators σ_l: $k_l \in S'_l \ldots L'_l$, $S_l \leq S'_l, S'_l \leq L'_l, L'_l \leq L_l$.

Up until now we have assumed j_i was assigned onto the fastest FU. The available delay is the minimal L'_l time for its successors σ_l minus the start time: $delay_i = min\{L'_l\} - k_i$. So any FU_j with $d_j \leq delay_i$ can be chosen.

The algorithm for feasibility is as follows:

Initial set ϕ' empty.

Step 1 Pick an unscheduled operator σ_r in ϕ.

Step 2 Schedule σ_r in time step: $\phi'.k_r = \phi.k_r$.

Step 3 Compute $delay_r = min\{L'_l\} - k_r$

Step 4 If $\phi.j_r \leq delay_r$: $\phi'.j_r = \phi.j_r$ else assign : $\phi'.j_r = j$ (j is the one with the slowest allowable execution) where $\sigma_r \in FU_j$ and $d_j \leq delay_r$.

Step 5 ASAP (update $S_l \rightarrow S'_l$)

Step 6 ALAP (update $L_l \rightarrow L'_l$)

Step 7 For all unscheduled operators σ_l in ϕ: if $\phi.k_l < S'_l$ set $\phi.k_l = S'_l$ and if $\phi.k_l > L'_l$ set $\phi.k_l = L'_l$.

Step 8 If any unscheduled operators in ϕ goto **step 1**.

The algorithm works by iteratively scheduling operators one at a time and each time running **ASAP** and **ALAP** reducing the valid time intervals for unscheduled operators and a feasible schedule can be obtained. The algorithm is deterministic and has complexity $\mathcal{O}(n^2)$.

6.3.3 Simulated annealing

The simulated annealing algorithm is a meta-heuristic algorithm for solving ILP problems which borrows from the physical model of near adiabatic crystallization i.e. the formation of a perfect crystal lattice.
Simulated annealing algorithm:

Initial Generate initial feasible solution vector $\rightarrow \phi$ and compute its area cost A

Step 1 Perturb ϕ, by randomly moving an operator in time and changing its FU assignment $\rightarrow \phi'$.

Step 2 Generate a feasible solution from the perturbed solution vector $\mathcal{F}(\phi') \rightarrow \phi'_{feasible}$

Step 3 Compute the area cost of $\phi'_{feasible} \rightarrow A'$.

Step 3 If the new cost is smaller than the existing solution ($A' < A$) accept the new solution ϕ', otherwise conditionally accept ϕ' depending if $exp(-(A' - A)/Temp) > random(1)$ is true.

Step 4 Update the solution space $(\phi', A', Temp') \rightarrow (\phi, A, Temp)$ and while not thermal equilibrium goto **step 1**.

Step 5 Reduce the temperature exponentially $Temp' = \alpha Temp$, with $0 < \alpha < 1$.

Step 6 If the temperature $Temp'$ is larger than $Temp_{crystal}$ (the stopping temperature) and A' is larger than A_{accept} goto **step 1**.

In the iteration step a random operator σ_i is chosen and random (acceptable) values are inserted for both k_i and j_i. Then the schedule is made feasible starting with scheduling σ_i and then scheduling the rest. In this way we ensure the perturbation survives the feasibility process. Then depending on the cost and the temperature we accept this new schedule or not. The fundamental difference between simulated annealing and local search lies in the ability at "high" temperatures to move "uphill" i.e. accept solutions which are less optimal (as well as always move "downhill" i.e. accept more optimal solutions). This is handled by the accept function maintaining the Boltzmann distribution from statistical mechanics. Initially the algorithm is started with an random solution which is made feasible. The thermal equilibrium condition repeats the inner-loop a certain amount, this is determined in the following chapter. $Temp_{crystal}$ stops the algorithm if the temperature comes down to 1. It can be shown mathematically that by selecting the correct temperature function specific to the problem, the simulated annealing algorithm will find the optimal solution. However the time spent on finding the optimal solution can be shown to be equal to or larger than the time to perform an exhaustive search. We set the start temperature to 10000 and it can be shown that a adiabatic cool-off in temperature corresponds to an exponential temperature decay i.e. the new temperature is generated by $Temp' = \alpha Temp$ with $0 < \alpha < 1$. We determine the appropriate value for α in the following chapter.

6.3.4 Evolutionary algorithm

The evolutionary algorithm approach is a meta-heuristic algorithm for solving ILP problems which is biologically inspired and implements the concept of "survival of the fittest".
Evolutionary algorithm:

Initial Generate initial set of feasible solution vectors $\rightarrow \Phi = \{\phi\}$, the population, and compute their respective area costs $\mathcal{A} = \{A\}$ and set the generation count to zero $G = 0$.

Step 1 Remove the half part of the population Φ with the lowest area cost $\rightarrow \Phi_{\frac{1}{2}}$ and set $\Phi' = \emptyset$.

Step 2 Select two elements from $\Phi_{\frac{1}{2}} \rightarrow \{\phi_a, \phi_b\}$, the parent solution vectors, and remove the elements from the set $\Phi_{\frac{1}{2}} \backslash \{\phi_a, \phi_b\} \rightarrow \Phi'_{\frac{1}{2}}$.

Step 3 Select a random crossover position and form two new solution vectors $\{\phi_a, \phi_b\} \rightarrow \{\psi, \varphi\}$, the child solution vectors.

Step 4 Mutate $\{\psi, \varphi\}$, by randomly moving an operator in time and changing its FU assignment $\rightarrow \{\psi', \varphi'\}$ using a low probability χ for mutating the solution vectors.

Step 5 Add the parent and the the child solution vectors to the new population $\Phi' + \{\phi_a, \phi_b, \psi', \varphi'\} \rightarrow \Phi''$.

Step 6 Update the solution sets$(\Phi'_{\frac{1}{2}}, \Phi'') \rightarrow (\Phi_{\frac{1}{2}}, \Phi')$ and if $\Phi_{\frac{1}{2}}$ is non-empty goto **step 2**.

Step 7 Generate feasible solutions from the perturbed solution vectors in Φ':$\mathcal{F}(\Phi'_{perturbed}) \rightarrow \Phi'_{feasible}$.

Step 8 Compute the area cost of $\Phi'_{feasible} \rightarrow \mathcal{A}'_{feasible}$.

Step 9 Increment the generation count G and update the solution space $(\Phi', A') \rightarrow (\Phi, A)$.

Step 10 If the best solution A_{best} is larger than A_{accept} and the generation G is less than G_{stop} goto **Step 1**.

The algorithm works by first deleting the most unfit half of the population. Then for two survivor pairs we select a random crosspoint and perform the crossover thereby producing two new children. Then we randomly sometimes add a mutation to the children. Then the children are made feasible (in the same way as for the simulated annealing) and the cost functions are evaluated and they are put into the new population. The fundamental difference between the local search/simulated annealing and the evolutionary algorithm is the use of a population of solutions in the latter. The deletion of the most unfit half in principle works as the "downhill" moving part and with the cross-over and mutation as the potential "downhill/uphill" moving part. Initially the algorithm is started with set of random solutions, made feasible and evaluated. The mutation rate is included in the evolutionary algorithms to prevent the entire population from converging to a single collection of similar solutions. The mutation rate should not be the principal solution space exploration method of the algorithm and should be very low; we chose $\chi = 0.01$. The generation count terminates the main loop if more than G_{stop} generations has passed. In the following chapter we determine both the population size and the G_{stop} parameter.

Module	Oprs	Area	Time-slots	E/time-slot [nJ]
add	$\{+\}$	2032.75	1	0.0266
sub	$\{-\}$	2032.75	1	0.0266
comp	$\{>\}$	2032.75	1	0.0266
ALU	$\{+,-,>\}$	2965.00	1	0.0266
mul1	$\{*\}$	$41978,50$	3	0.1046
mul2	$\{*\}$	28414.50	6	0.0523
mul3	$\{*\}$	14638.75	17	0.0319
input	i	43.00	1	0.0
output	o	43.00	1	0.0

Table 6.2: 16 bit functional unit library based on balsa-cost numbers, available to the synthesis algorithm.

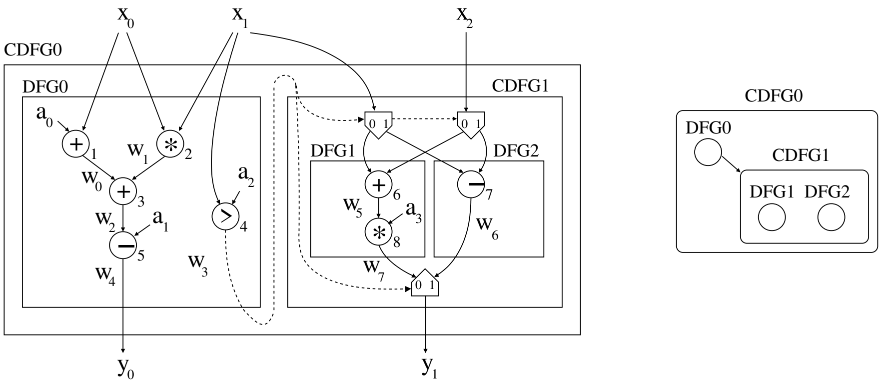


Figure 6.10: (Left) Partition of our CDFG into DFG fragments. (Right) The corresponding task graph to the partition of the CDFG.

6.4 Control data flow graph synthesis

For synthesis of control data flow graphs a basic block synthesis procedure is used. Thus repetitive and conditional segments of the CDFG are scheduled as independent parts or independent tasks i.e. the synthesis problem of the CDFG is reduced to a synthesis problem of a set of DFG's [64, 103], as we have presented in the previous sections i.e. this algorithm builds on top of these algorithm.

The partition of the CDFG into basics blocks follows a hierarchical decent into the CDFG where the DFG-fragments are identified as the largest sets of deterministically related operators in the CDFG. The largest set of deterministically related operators is defined as the largest group of operators for which a static execution order can be found.

Having partitioned the CDFG into basic blocks a hierarchical task graph con-

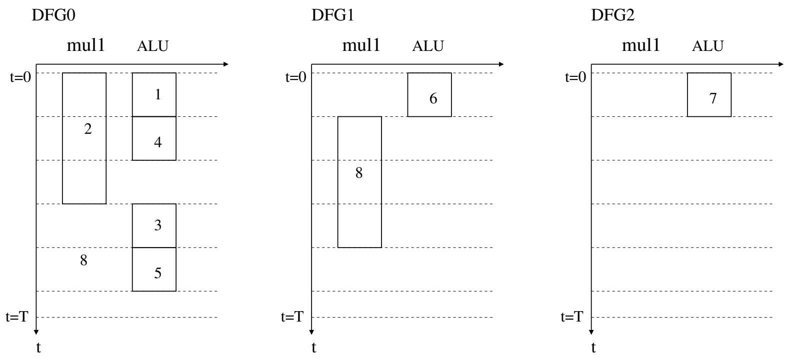


Figure 6.11: Scheduling of the DFG fragments: DFG0, DFG1, DFG2.

taining the relationships between the different DFG fragments is generated. This is illustrated on Figure 6.10 for our example CDFG. For our example the largest group of deterministicly related operators in CDFG0 are operators:$\{1, 2, 3, 4, 5\}$, which is denoted DFG0. Besides that there exists a branch-section which we denote CDFG1. The procedure is then repeated for CDFG1, which contains two sets of deterministicly related operators DFG1=$\{6, 8\}$ and DFG2=$\{7\}$. Each of these DFG fragments are nodes in the corresponding task graph. The task graph has a single dependency between DFG0 and CDFG1, which originates from the execution of the conditional choice, operator $\{4\}$ which is computed in DFG0 and used in CDFG1.

To keep track of the current solution the algorithm is working on, we introduce a solution vector ϕ containing n tuples (one for each DFG$_i$ fragment), consisting of the pair (t_i, d_i) where t_i is the start time-step for DFG$_i$ and d_i is the synthesis delay constraint for this DFG$_i$ fragment i.e. the maximally allowed execution time for DFG$_i$. Let the schedule be defined by:

$$\phi = [(t_1, d_1), (t_2, d_2), \ldots, (t_n, d_n)]$$

The time-steps t_i are bound by the ASAP and ALAP times for the task graph, where it is assumed all the DFG$_i$ fragments are executed using their ASAP schedules. The individual synthesis delay constraints range from the ASAP time of the DFG$_i$ fragment to the ALAP time of the DFG$_i$ fragment computed where all other DFG fragments are executed using their ASAP times and all predecessor DFG$_i$ are scheduled at ASAP start-time intervals and all successors are scheduled using their ALAP time intervals. This specifies the maximally allowed time interval for that DFG fragment.

The main synthesis algorithm operates in "two-levels": The principal level schedules the DFG fragments (task-scheduling) using the $\{t_i\}$ start-times and the sublevel or innerloop reschedules a single DFG$_i$ fragment using its d_i synthesis delay constraint.

CDFG scheduling:

Initial Generate the task-graph by descending hierarchically into the CDFG dividing deterministic sets into DFG_i which are nodes in the task graph. Generate the initial solution vector by setting the set of start times $\{t_i\}$ to the ASAP start-time for the task-graph. And set the set of synthesis time-constraints $\{d_i\}$ to the length of the ASAP schedules for the $\{\mathrm{DFG}_i\}$.

Step 1 Perturb ϕ, by randomly selecting a tuple i and randomly move the start time t_i and change the synthesis constraint $d_i \rightarrow \phi'$. All has to be selected within their respective ASAP-ALAP intervals.

Step 2 [*Innerloop:*] Reschedule the selected DFG_i using one of the methods presented in the previous sections, using the corresponding constraint d_i.

Step 3 Schedule the task graph using the task solution vector and allocate using groups of FUs from the DFG fragments. For CDFG fragments containing choices between several DFG's use the worst-case time-delay and area usage. For conditional repetitive CDFG fragments assume a single execution. The resulting functional unit allocation is the maximal concurrent use of each type of FU.

Step 4 Locally optimize the resulting combined schedule, by taking advantage of the slack but without allocation more functional units than allocated in the current iteration. Compute the area cost of $\phi' \rightarrow A'$ from the functional unit allocation.

Step 5 If the resulting schedule violates the system time constraint T add a large penalty area to the area cost: $A' + P \rightarrow A'$.

Step 6 If the new cost is smaller than the existing solution ($A' < A$) accept the new solution ϕ', otherwise conditionally accept ϕ' depending if $exp(-(A' - A)/Temp) > random(1)$ is true.

Step 7 Update the solution space $(\phi', A', Temp') \rightarrow (\phi, A, Temp)$ and while not thermal equilibrium goto **step 1**.

Step 8 Reduce the temperature exponentially $Temp' = \alpha Temp$, with $0 < \alpha < 1$.

Step 9 If the temperature $Temp'$ is larger than $Temp_{crystal}$ (the stopping temperature) and A' is larger than A_{accept} goto **step 1**.

The algorithm operates similar to the simulated annealing synthesis algorithm in subsection 6.3.3, the principal difference is in step 2, the innerloop, where a DFG fragment is scheduled. Here a penalty cost is used for infeasible solutions as no good feasibility algorithm has been found yet.

The scheduling of the different DFG fragments are shown in Figure 6.11. For our simple example task-graph there is, because of the dependency between DFG0 and CDFG1, only one possible task schedule, which is shown in Figure 6.12 (left). The corresponding schedule at operator level is shown following thereafter, this schedule

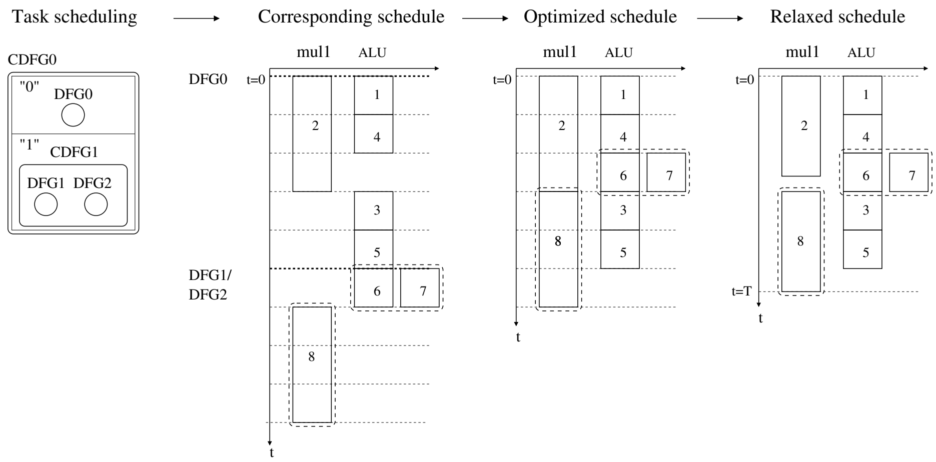


Figure 6.12: Synchronous task-scheduling and the corresponding schedule of operators. Slack exploitation leads to the optimized schedule, which is finally relaxed into an asynchronous schedule.

contains a lot of slack stemming from the individual scheduling of the DFGS and not the CDFG as a hole. In this and other cases the schedule can be compressed following a “first come first serve” principle where operators are moved upwards in time to empty time slots, preserving the relative scheduling of the operators in DFG and their relative dependence between the DFGs. The resulting schedule for example is shown on the same figure. Finally the time-slot restrictions are removed, shortening the execution time of the multiply operation and relaxing the schedule into an asynchronous schedule. The resulting schedule has been used through-out in this thesis. This schedule is not optimal when compared to the optimal schedule generated through a continuous time exhaustive-search method, but the difference is marginal.

For the power aware scheduling algorithm considered in the first section, the basic block is extended to include the conditional sections of the CDFG, but not repetitive structures. This means our entire example is one basic block for that algorithm. The power aware scheduling is a clique based algorithm which operates using operator disjunctiveness. There are two types of disjunctiveness to characterize the relationship between two operators. The operators can be:

Path disjunctive For operators to be path disjunctive, there should exist a dependence relation between them i.e. there should exist a path in the CDFG connecting the two operators together and preventing the operators from having overlapped execution times.

Branch disjunctive For operators to be branch disjunctive each operator should semantically exclude the execution of the other i.e. if each operator belong to different branches in a branch construction only one of the operators can be executed and therefore no overlapping execution can occur.

Operators that are disjunctive will only take up one execution slot on a functional unit and thus can be advantageously scheduled onto the same functional unit.

A power-constraint could be included alongside the task execution-time constraint d_i and thus be used to power constraint the scheduling of the DFGs. The system power constraint could then be handled by a penalty function, similar to the penalty introduced by violating the system time constraint T.

6.5 Summary

In this chapter we have presented a set of behavioral synthesis algorithms: A power-aware synthesis algorithm for CDFGs without repetative structures, which we have implemented. A simulated annealing algorithm and an evolutionary algorithm for synthesis of DFG fragments and we have developed a feasibility algorithm which enables the possibility of easy crossing between areas of feasible solutions in the solution space for these meta-heuristic algorithms. All of which we have implemented. Finally we have outlined a behavioral synthesis algorithm for synthesis of CDFGs. In the following chapter we compare the implemented algorithms.

Chapter 7

Results

This chapter presents an evaluation of the efficiency of the computation model and our methods. The purpose here is not to compare asynchronous vs. synchronous, as each have their own application domains and acts as supplements. Neither is direct comparison with other asynchronous synthesis methods attempted, as this involves comparing different technologies and implementation styles which renders any comparisons debatable/inconclusive.

We benchmark our algorithms on a representative set of problems from the classical set of synthesis benchmark CDFGs: FIR is a eight-tap FIR filter. HAL is an iterative Euler integration of a differential equation. ELLIPTIC is a fifth order elliptic wave filter. COSINE is a part of the DCT algorithm. Throughout in this chapter we will use the FU library shown in Figure 6.2. This FU library consist of "balsa-cost" numbers of corresponding balsa-programs that implement the functionality of the functional units.

We begin with presenting the results of the behavioral synthesis algorithms where we are interested in their run-time. For these we only consider the area of functional units. Then we proceed by investigating the circuit implementation method presented in this thesis; we use our method on the GCD algorithm, which we compare against a manually optimized design. For these results we use the full circuit area. Then we implement the benchmark set and investigate the overhead of implementing resource-sharing using this method. Finally we look at the circuit characteristics at layout level.

Module	Oprs	Area	Time-slots	E/time-slot [nJ]
add	$\{+\}$	2032.75	1	0.0266
sub	$\{-\}$	2032.75	1	0.0266
comp	$\{>\}$	2032.75	1	0.0266
ALU	$\{+,-,>\}$	2965.00	1	0.0266
mul1	$\{*\}$	41978,50	3	0.1046
mul2	$\{*\}$	28414.50	6	0.0523
mul3	$\{*\}$	14638.75	17	0.0319
input	i	43.00	1	0.0
output	o	43.00	1	0.0

Table 7.1: 16 bit functional unit library based on balsa-cost numbers, available to the synthesis algorithm.

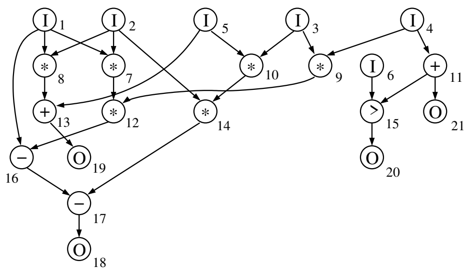


Figure 7.1: CDFG for the HAL computation, where I and O are the input and output nodes.

7.1 Results for power aware scheduling

We have benchmarked the algorithm on a set of CDFGs, using our FU library shown in table 7.1, all performed on a 200MHz Pentium II, with 96 MB memory. We do not take an eventual correlation among input data in to account and assume worst-case power measures for computations in the different FU components. The first test is of the **PASAP** algorithm where we investigate the required time delay of the CDFGs, as a function of the power constraint. The results are shown in table 7.2. The second test is of the main clique-partitioning algorithm where we investigate the area of the resulting circuits as a function of the power constraint, with a constant time frame. We perform this test for a few different time-frames. The results are shown in Figure 7.2. Finally some different power and time constraints and the circuit area and the CPU time to find the solutions is shown in table 7.3.

As shown in Figure 7.2 (eg. ELLIPTIC with T=30 and COSINE with T=15) using a global synthesis algorithm we can trade in area to obtain a solution which fits our power requirements. The average area penalty ranges in the region of 20

HAL, vertices=21, edges=25								
$E_<[nJ]$	inf	0.500	0.400	0.300	0.250	0.200	0.150	0.125
T_{PASAP}	9	9	11	12	12	20	20	22
FIR, vertices=24, edges=24								
$E_<[nJ]$	inf	1.00	0.600	0.400	0.300	0.200	0.150	0.125
T_{PASAP}	8	8	10	13	16	28	27	29
ELLIPTIC, vertices=49, edges=43								
$E_<[nJ]$	inf	0.500	0.400	0.300	0.250	0.200	0.150	0.125
T_{PASAP}	21	21	23	23	24	31	32	38
COSINE, vertices=57, edges=77								
$E_<[nJ]$	inf	1.00	0.800	0.500	0.300	0.200	0.150	0.125
T_{PASAP}	11	11	14	17	27	51	54	56

Table 7.2: Time vs. power using the PASAP scheduling for the set of benchmarks.

$E_<[nJ]$	T	A	$T_{CPU}[s]$
inf	11	440,499	15.82
0.500	17	314,485	46.75
0.400	26	138,310	118.29
0.300	32	96,289	160.22
0.300	37	95,289	297.03
0.200	56	96,289	442.36
0.125	66	56,386	193.79
0.125	71	56,386	357.58

Table 7.3: Different power and time constraints generated by the main synthesis algorithm, the resulting area and the CPU synthesis time for COSINE.

percent which is an acceptable penalty, as power is the critical parameter here.

An interesting aspect is that with a large time and power constraint, the algorithm might find a worse solution with respect to area, than when the power constraint is tight. The reason for this lies in the greedy approach which might make a bad decision early on. With the tight power constraint this is prevented (no need to allocate many FUs in parallel if only one or two is used at a time), an example of this is COSINE T=25 and T=20.

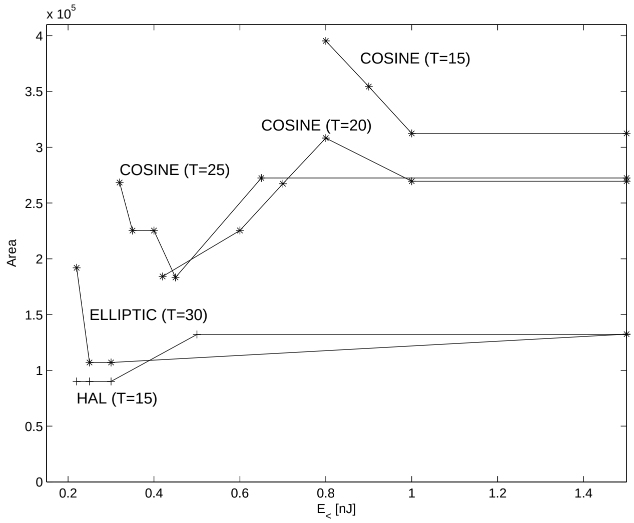


Figure 7.2: Power vs. area under different time constraints for HAL, COSINE and ELLIPTIC.

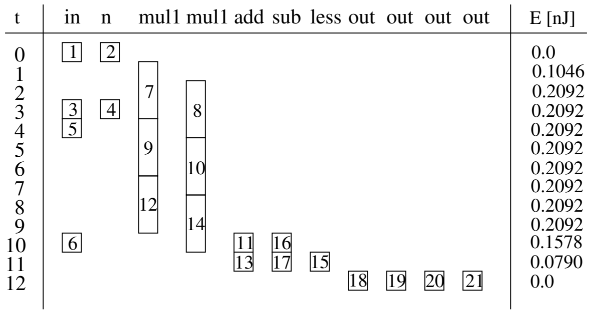


Figure 7.3: Tightly constrained power-aware schedule for the HAL computation, T=13, E=0.210nJ. Requiring 2 inputs, 2 mults (fast), 1 add, 1 sub, 1 les and 4 outputs, with a total area of 90311.

7.2 Results for simulated annealing and evolutionary algorithm

For the meta-heuristic algorithm we first need to adjust the meta heuristic parameters for the algorithms. This is in many cases more of an art, than a science. In the following we will experimentally find the best parameter setting. The test case we use to adjust the parameters from, is the HAL computation with a time frame of $T = 20$. This is an arbitrary case, and there is no guarantee this will lead to the

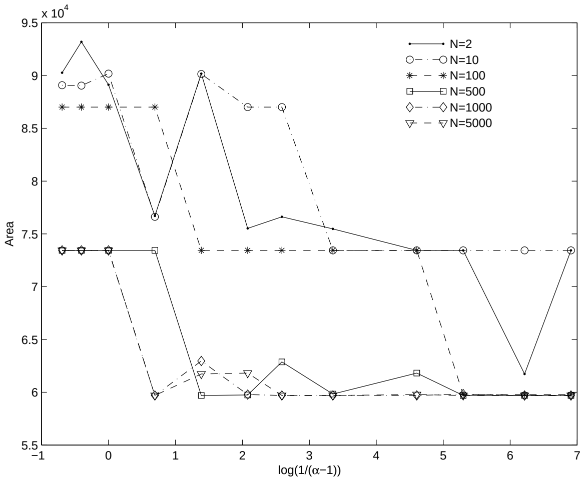


Figure 7.4: Solution (HAL $T = 20$) from simulated annealing as a function of the α temperature change coefficient and the number N of iterations to reach "thermal equilibrium".

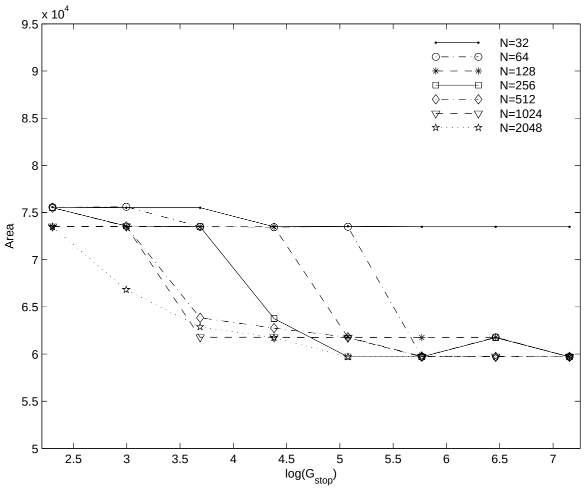


Figure 7.5: Solution (HAL $T = 20$) from evolutionary algorithm as a function of the G_{stop} generation count and the population size (N).

optimal set of parameters for all other cases. In particular one should beware of fine-tuning the algorithm precisely to this case as it might mean the meta-heuristic algorithms is really good at finding this solution, but terrible for all other cases and problems. On the other hand we need to adjust the parameters for something and a

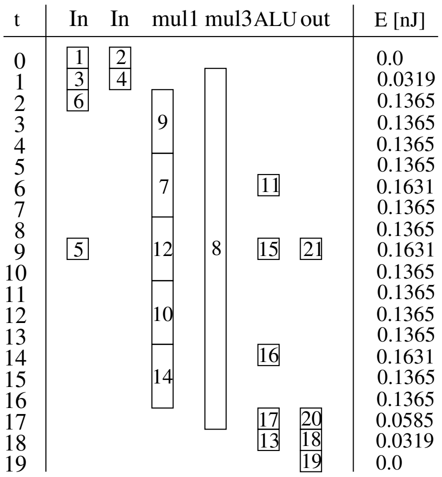


Figure 7.6: Schedule, FU allocation and operator assignment generated by simulated annealing for HAL with $T = 20$ constraint, giving a total "balsa-cost" area of 59700.

small example where to the exact optimum is known is good test for narrowing down the parameter setting.

We begin with simulated annealing, where we need to find the temperature change coefficient α and the "thermal equilibrium" number N. In Figure 7.4 we have shown several runs of the algorithm for various parameter settings and plotted the solution the algorithm finds. Each point represents an entirely new run. As can be seen the simulated algorithm is rather unstable capable of getting stuck at a local minimum. However for $N = 500$ and larger, the algorithm tends to become more stable and produce good solutions (actually the optimal solution) at every run. The best parameter setting for α seems to be $\alpha = 1.250$ for larger α the algorithm does not produce better solutions, only taking exponentially more time to complete. These setting also seem to produce good solutions for the other problems in the benchmark set.

Next is the evolutionary algorithm, where we need to find the G_{stop} generation count and the N population count. In Figure 7.5 we have shown several runs of the algorithm for various parameter settings and plotted the solution the algorithm finds. Again each point represents an entirely new run. As can be seen the simulated algorithm is rather stable capable of producing reliable results. Another factor is the high-dependency on the population size. With a population around 512 the algorithm starts converging towards the global optimum with the fast convergence and choosing a large population size does not increase the convergence. The best value for the maximum generation count G_{stop} seems to be around in the range from 320 to 640. To be on the safe side we chose 640 generations. Again these parameters settings seems to produce good solutions for the other algorithms in the benchmark set except for COSINE, for which the algorithm have problems finding some particular solutions.

We have benchmarked the algorithms on two DFGs: HAL ($T_{ASAP} = 10$) and COSINE ($T_{ASAP} = 11$). We are interested in the CPU-time i.e.. the amount of time it takes running the algorithms to get a solution satisfying our area requirements. For the two DFGs we apply the two meta-heuristic algorithms, giving us four primary test cases (shown in table 7.4). For each test case we set five silicon area requirements and six time frame requirements $T = dt + T_{ASAP}$, (the blanks are where the meta-heuristic-algorithms fail to find a solution either because there is no optimal solution satisfying the requirement or in border cases because the algorithms are heuristic).

Again, all tests are performed on a 200MHz Pentium II, with 96 MB memory and all numbers reflect a statistical average of running the algorithms 500 times on each problem instance.

In general the simulated annealing out-performs the evolutionary algorithm in terms of CPU time required to find a solution for large problems (i.e. COSINE). The primary reason stems from the evolutionary algorithm working on a large population, which in every iteration has to be made feasible and cost evaluated, whereas the simulated annealing only works with one problem instance. On the other-hand the evolutionary algorithm seems to perform more "stable", unlike simulated annealing which is capable of getting "stuck" in local-minimums for some runs. Comparing the evolutionary algorithm with the simulated annealing the evolutionary algorithm takes significantly longer time to run and does find just as good solutions as simulated annealing. In particular in the COSINE case the evolutionary algorithm has problems. This does not mean the evolutionary algorithm cannot find the solutions eg. if run free the evolutionary algorithm is capable of finding a solution for COSINE, $T = 107$, below the area requirement of 49200, however it took $25857.4s$ or approximately 7.18 hours. The evolutionary algorithm does not however have similar problems for FIR or ELLIPTIC.

A property of the proposed CDFG synthesis algorithm is that one of these algorithms will be run for the DFG fragments, until the main synthesis algorithm converges, it is therefore important that these algorithms generate the solutions fast. This favours the simulated annealing over the two other algorithms.

Finally in Figure 7.6 is shown the optimal schedule generated by the meta-heuristic algorithms in the parameter investigation.

7.3 Results for asynchronous behavioral synthesis

In order to demonstrate the feasibility of the proposed approach and in order to evaluate the efficiency of the proposed implementation template. We begin in subsection 7.3.1 with applying our approach on the GCD algorithm and then continue in subsection 7.3.2 to our benchmark circuits and finally for FIR and HAL we have produced layouts and in subsection 7.3.3 we report on the area, speed and power figures.

But first we report on the area cost of our running example. The original Balsa-

Simulated Annealing (HAL)						
T	Area requirement					
	140,000	120,000	90,000	75,000	60,000	46,000
10	0.165					
13	0.012	0.270	2.418			
16	0.000	0.092	0.220			
18	0.000	0.056	0.165	4.505		
20	0.000	0.010	0.07	3.576	23.91	
22	0.000	0.000	0.35	1.202	11.43	18.86
Simulated Annealing (COSINE)						
T	Area requirement					
	350,000	160,000	110,000	92,000	78,000	49,200
13	189.9					
21	0.165	195.6				
32	0.070	1.593	202.6			
35	0.110	0.659	42.03	205.6		
86	0.0505	0.440	3.077	8.846	55.54	
107	0.210	0.385	2.418	10.33	39.23	259.1
Evolutionary Algorithm (HAL)						
T	Area requirement					
	140,000	120,000	90,000	75,000	60,000	46,000
10	0.275					
13	0.210	0.330	0.934			
16	0.000	0.270	0.275			
18	0.000	0.165	0.261	10.934		
20	0.000	0.015	0.031	2.582	40.01	
22	0.000	0.002	0.011	2.637	6.593	30.49
Evolutionary Algorithm (COSINE)						
T	Area requirement					
	350,000	160,000	110,000	92,000	78,000	68,000
13	22.253					
21	0.031	369.0				
32	0.00	1.923				
35	0.000	1.978	302.2			
86	0.0201	0.771	2.253	167.5	271.8	
107	0.000	0.010	2.410	2.363	204.0	804.1

Table 7.4: Run-times ($T_{CPU}[s]$) for two CDFGs (HAL and COSINE) by simulated annealing and evolutionary algorithm.

code in Figure 2.1 would have a Balsa-cost of 96,787.5 (using the numbers from our multiplier), whereas the resulting synthesized Balsa-code shown on pages 65-67 have

```
import [balsa.types.basic]
type word is 16 bits

procedure gcd(input a,b: word ; output c: word) is
  variable ai,bi : word
begin
  loop
     a -> ai || b -> bi ;
     while ai/=bi then
       if ai>bi then
         ai:=(ai-bi as word)
       else
         bi:=(bi-ai as word)
       end
     end ;
     c<-ai
  end
end
```

Figure 7.7: The GCD-algorithm.

a balsa-cost of $60,037.5$. Representing an area reduction of 38%.

7.3.1 GCD

In [53, section 13.2.3] the process of syntax directed and optimizations at the source code level (using Tangram) is illustrated using GCD as an example. Figure 7.7 shows the well known algorithm expressed in Balsa code. The problem is that the source code contains 4 operator symbols, and that the corresponding circuit have 4 functional units as well. In order to optimize the area the designer has to rewrite the code. Figure 7.8 shows one such optimized design. It is slightly different from the Tangram code in [92] as Balsa does not support exactly the same constructs as Tangram, but the ideas underlying the optimization are the same. Even this simple example hints that the process of optimizing the circuit and exploring alternatives *can* be tedious. In behavioral synthesis one would take the basic code in Figure 7.7 and synthesize it with area minimization as the constraint. The work presented here does exactly this, i.e. from a CDFG extracted from the basic code in Figure 7.7 we automatically synthesize a circuit containing two compares and one subtraction operator. Table 7.5 shows the area estimates ("balsa-cost") reported by Balsa for the different versions of the circuit. It is seen that behavioral synthesis in this example actually outperforms the manually optimized design.

The important message here is that the overhead introduced by our method is so small the resulting area cost is in the same region as a manually optimized circuit.

Program	balsa-cost
gcd_basic	7435.25
gcd_opt	7161.75
gcd_synt	6846.00

Table 7.5: Comparison of the plain GCD, the optimized GCD and the synthesized GCD. "balsa-cost" is an area measure reported by the Balsa tool.

```
import [balsa.types.basic]
type word is 16 bits

type twoword is record
  a,b:word
end

procedure gcd(input ab: twoword ; output c: word) is
  variable data : twoword
begin
  loop
     ab->data ;
     while data.a/=data.b then
       if data.a>data.b then
         data:=(twoword {((data.a-data.b) as word),
                                    data.b as word)})
       else
         data:=(twoword {data.b,data.a})
       end
     end ;
     c<-data.a
  end
end
```

Figure 7.8: An optimized version of GCD.

7.3.2 Benchmarks

Using our behavioral synthesis methods, more precisely simulated annealing, together with our computation model and our implementation templates, we have synthesized the range of benchmarks as shown in table 7.6. Again the area is expressed in terms of the "cost" reported by Balsa. As seen, it is possible to automatically synthesize implementations with a range of constraints. The table is divided into six groups: The first group shows the balsa implementation as a designer would implement them without resource sharing. The second group shows the area of the synthesized versions as produced directly from the simulated annealing algorithm before latch assignment

```
import [balsa.types.basic]
type word is 16 bits

procedure gcd(input a,b: word ; output c: word) is
channel FU0_a,FU0_b,FU0_z:word

procedure FU0_sub(intput FU0_a,FU0_b:word;output FU0_z:word) is
begin
  loop
    select FU0_a,FU0_b then
      FU0_z<-(FU0_a-FU0_b as word)
    end
  end
end

procedure gcd_architecture(input a,b,FU0_z:word;
                           output FU0_a,FU0_b,c:word)
variable L0,L1,L2 : word
channel cL2:word
begin
  loop
     a -> L0 || b -> L1 ;
     while L0/=L1 then
       if L0>L1 then
         FU0_a<-L0 || FU0_b<-L1;
         cL2->L0
       else
         FU0_a<-L1 || FU0_b<-L0;
         cL2->L1
       end ||
       [ FU0_z->L2 ; cL2<-L2 ]
     end ;
     c<-L0
  end
end

begin
  FU0_sub(FU0_a,FU0_b,FU0_z) ||
  gcd_architecture(a,b,FU0_z,FU0_a,FU0_b,c)
end
```

Figure 7.9: The synthesized version based on the basic algorithm in Figure 7.7.

i.e. only the pure FU area is reported. The third (3a) and fourth (3b) group shows to the second group corresponding balsa-implementation using the alpha and beta templates respectively, but without using the control and mux-optimizing algorithm.

For the fifth (4a) and sixth (4b) groups these optimizations have been included. Thus the difference between items of the second group and the third or fourth group is the implementation overhead of using these approaches and the overhead of the implementation templates proposed by this thesis.

The first observation is that again there is a large area saving when applying resource-sharing. Secondly, the overhead of implementing the circuits, consisting of controller area, latch area and multiplexor/-demultiplexor area is around 40% of the total area of the circuits and the functional units make up around 60%. This is not-unexpected as these additional area contributions are significant also in synchronous behavioral synthesis, and for digital circuit design in general. Finally, there is the comparison between the two computation models, should there be power guarding input/output-latches around functional units or not with respect to area? The area difference between the two is very little and for the four benchmarks here there is two cases where the input/output latch is smaller than the input/output-latch circuit, one case where there is almost equality and one case the non-input/output-latch circuit is smaller than the input/output-latch circuit. In general the non-input/output-latch circuits have a smaller total latch count, however there is usually a larger mux-depth associated with these circuits, which counters this effect. Based on the current observations, we believe it to be application dependent which type of computation model that have the smallest area.

The next question is how efficient these implementations are. To answer this question we have produced and simulated layouts for FIR and HAL.

7.3.3 Layout results

For the benchmarks FIR and HAL in beta-style, we have used the back-end part of the Balsa tools and actually produced a layout targeting handshake components using the single-rail 4-phase early protocol. We have used the existing synthesis flow at Manchester University, which is based upon a $0.18\mu m$ STM standard-cell technology, which have been augmented with standard cell components for implementing various special asynchronous components such as Muller C-elements.

Simulation results are obtained by simulating the post place-and-route Verilog netlist together with extracted layout information in NanoSim. We simulate 200 computations, using random numbers with out any correlation. All the circuits are implemented using 16-bit variables and are simulated at 1.8V and at a temperature of 25^oC.

It is important to stress the results do not represent an attempt to evaluate the asynchronous implementations against corresponding synchronous ones; our focus is on the efficiency of the automated resource sharing within the asynchronous domain.

The benchmark results are shown in table 7.7, where t is the average time to do one computation, A is the layout area and E is the average energy consumption per computation. In a similar way we have characterized the ALU and multiplier operators, see table 7.8. The speed figures in table 7.8 have been used in calculating the schedules.

Implementations 1 and 3 in table 7.7 are the direct non-resource-shared circuit implementations of the computations. These have also been designed using latches on the input and output of the multipliers. Although this gives an extra area overhead it is insignificant compared to the area of the multiplier. The important fact is that it reduces the combinatorial depth of the circuit and thus reduces the power consumption, which leads to a more fair comparison. The speed figures in table 7.7 includes a $20ns$ handshake delay in the testbench used to simulate the layouts.

The results in table 7.7 shows that resource sharing saves area at the expense of reduced speed. This is as could be expected. Concerning energy consumption it is interesting to note that it remains constant. Given that resource sharing leads to more control circuitry for the same computation, an increase in energy consumption could be expected. *It seems that the smaller size of the layout and the reduced wire length, which results from this leads to a power saving which corresponds to the increase caused by the added control.*

A visual comparison of the layouts for implementation 3 and 4 is shown in Figure 7.10, illustrating the area reduction achieved by resource sharing.

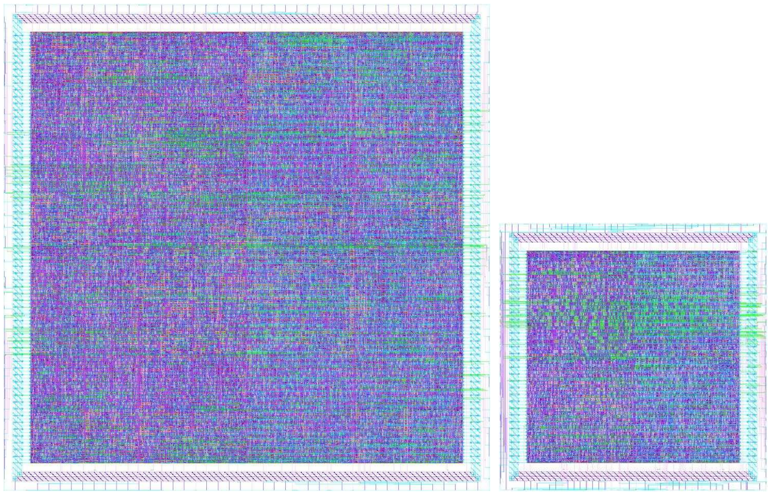


Figure 7.10: Visual layout comparison of the non-resource shared HAL computation (left) and the maximally resource shared HAL computation (right).

7.4 Summary

In this chapter we have presented results for our behavioral synthesis algorithms. We have applied the power aware synthesis algorithm on several examples and investigated different regions in the time-power-constraint space. The algorithm is capable of finding low area solutions fulfilling the constraints and for the chosen silicon library we find the power constraint in the worst case adds an increase in silicon area of roughly 20 percent. Furthermore we have implemented two meta-heuristic algorithms for solving high-level behavioral synthesis: Simulated Annealing and Evolutionary Algorithm. In general the Simulated Annealing performs faster and finds better solutions to the problem, however the Evolutionary Algorithm is more stable. Both methods find better solutions than the power-aware synthesis algorithm with infinite power constraint. As the CDFG synthesis algorithm will require several iterations for each individual task (DFG problem) it is important the DFG synthesis algorithm is fast. Therefore based on the effectiveness of the simulated annealing we recommend that solution.

Then we have demonstrated that for a small design with few opportunities for resource sharing (i.e. where the overhead of an automated method is high) our approach is doing very well. Finally, for a benchmark suite we have implemented and shown the resource sharing behaves as we predict and that there is no unexpected penalty, like excess power consumption.

(1)	Original code									
Program	T	add	sub	les	ALU	mul1	mul2	mul3	lt	cost
FIR	-	7	0	0	0	8	0	0	0	459,749.25
HAL	-	2	2	1	0	6	0	0	0	348,093.75
ELLIPTIC	-	26	0	0	0	8	0	0	7	518,017.75
COSINE	-	13	13	0	0	16	0	0	0	964,470.25
(2)	Synthesized functional units only									
Program	T	add	sub	les	ALU	mul1	mul2	mul3	lt	cost
FIR	11	1	0	0	0	2	0	0	-	85,989.75
HAL	7	0	0	0	1	2	0	0	-	86,922.75
ELLIPTIC	18	2	0	0	0	2	0	0	-	88,022.50
COSINE	18	2	2	0	0	2	0	0	-	92,088.00
(3a)	Synthesized code in/output latch no ctrl. optimization									
Program	T	add	sub	les	ALU	mul1	mul2	mul3	lt	cost
FIR	11	1	0	0	0	2	0	0	21	142,539.25
HAL	7	0	0	0	1	2	0	0	16	135,218.50
ELLIPTIC	18	2	0	0	0	2	0	0	23	163,014.75
COSINE	18	2	2	0	0	2	0	0	32	170,984.00
(3b)	Synthesized code no in/output latch no ctrl. optimization									
Program	T	add	sub	les	ALU	mul1	mul2	mul3	lt	cost
FIR	11	1	0	0	0	2	0	0	12	140,535.00
HAL	7	0	0	0	1	2	0	0	9	135,214.50
ELLIPTIC	18	2	0	0	0	2	0	0	19	168,873.50
COSINE	18	2	2	0	0	2	0	0	17	161,792.25
(4a)	Synthesized code in/output latch with ctrl. optimization									
Program	T	add	sub	les	ALU	mul1	mul2	mul3	lt	cost
FIR	11	1	0	0	0	2	0	0	21	128,893.25
HAL	7	0	0	0	1	2	0	0	16	133,586.50
ELLIPTIC	18	2	0	0	0	2	0	0	23	143,248.75
COSINE	18	2	2	0	0	2	0	0	32	160,889.50
(4b)	Synthesized code no in/output latch with ctrl. optimization									
Program	T	add	sub	les	ALU	mul1	mul2	mul3	lt	cost
FIR	11	1	0	0	0	2	0	0	12	131,598.25
HAL	7	0	0	0	1	2	0	0	9	133,664.25
ELLIPTIC	18	2	0	0	0	2	0	0	19	150,256.00
COSINE	18	2	2	0	0	2	0	0	17	155,626.75

Table 7.6: Benchmark results generated by simulated annealing. Column T is the time-constraint given to the synthesis tool. Columns add, sub, les, ALU, mul.. and lt lists the number of adders, subtractors etc. in the circuits. Cost is “balsa-cost”, an area measure reported by the Balsa tool.

id	Alg.	$*$	ALU	t $[ns]$	A $[mm^2]$	E $[nJ]$
1	FIR	8	7	124.7	0.877	2.95
2	FIR	2	1	284.8	0.282	2.80
3	HAL	5	5	171.2	0.667	2.03
4	HAL	2	1	309.6	0.260	1.89
5	HAL	1	1	397.4	0.151	2.01

Table 7.7: Layout results (beta-style).

FU	σ	t$[ns]$	A $[mm^2]$	E $[nJ]$
ALU	$\{+, -, >\}$	25.5	0.0112	0.0266
Mult	$\{*\}$	56.3	0.105	0.314

Table 7.8: FU library (16-bit) based on layout in $0.18\mu m$ technology, used by our synthesis algorithm.

Chapter 8

Conclusion

This thesis presented a novel approach for behavioral synthesis of asynchronous circuits. The proposed approach seeks to merge the domains of traditional behavioral synthesis and asynchronous circuits. This is accomplished by providing a computation model, that is based upon asynchronous handshake components and which allows us to use the transformations and optimizations used in synchronous synthesis directly in asynchronous circuits. Furthermore the same model allows the use of the transformations and optimizations developed for continuous time.

The central elements in this thesis evolves around the connection between the synchronization events used in traditional techniques of behavioral synthesis and the transition handshake component locally controlling the beginning of an operation and writing the result of an operation. This is bound together by our hardware architecture consisting of a datapath with the transition handshake component and a controller determining these events. This computation model relaxes the strict ordering of the synchronous circuit and the synchronous schedule into the continuous time domain, the schedule for the asynchronous circuit.

We have accomplished the following: (i) a method for synthesizing a CDFG to a Balsa-description has been developed using a methodology closely related to, but not restricted to, traditional synchronous behavioral synthesis. This allows us to use existing techniques for design space exploration and resource sharing by adding physical constraints to the circuit. (ii) A series of behavioral synthesis algorithms has been developed for this purpose. The first is a power-aware synthesis algorithm, which targets a power profile below a certain threshold. Here we have shown it is possible to trade-in area to obtain this power profile. We have also shown that even though the power profile directly leads to a restriction on the number of multipliers in the circuit, the other smaller contributor operations still have a significant impact and are very important for finding the optimal schedule. Then we have implemented a more conventional resource sharing synthesis algorithm based on the meta-heuristic algorithms; simulated annealing and evolutionary algorithm. For these we have shown the simulated annealing algorithm outperforms the evolutionary algorithm with respect to run-time. We have also shown the meta-heuristic algorithms outperform the first power-aware algorithm with respect to run-time. (iii) We have developed different computation models depending on the requirement to isolate functional units when they are idle and developed the associated variable-lifetime algorithms. (iv) We have shown our approach to be efficient even for small circuits and that the overhead of implementing our approach is small compared to the area saving achieved using our method. (v) Using this method and the Balsa and Cadence design tools several layouts have been designed and simulated. The results show that it is possible to do tradeoffs between area and circuit delay and to do so without any increase in power consumption for asynchronous circuits. This gives us proof of concept. Furthermore we have an indication that significant resource sharing leads to a reduction of the average load capacitance and thus a reduction of the power consumption.

The rest of this chapter will present the advantages of the proposed approach, put the method in perspective and discuss future directions.

8.1 Advantages of the approach

There are several advantages of our approach to behavioral synthesis of asynchronous circuits:

Traditional datapath and controller The fact that our target computation model is the asynchronous equivalent to the synchronous computation model allows us the use of existing traditional behavioral synthesis approaches. This enables an entire range of behavioral synthesis algorithms to become available.

Continuous time Our computation model directly targets schedules generated through the use of continuous time synthesis methods, this includes methods from operations research.

Only handshake components Our approach builds entirely on asynchronous channels and handshake components, including the controller part. This avoids the often complex task of synthesizing an asynchronous controller and allows for asynchronous circuits of any size to be easily constructed.

Building upon syntax directed synthesis Our approach targets a high-level syntax directed hardware description language which specifically targets asynchronous circuits. This has the advantage that we do not need to keep up with technology change and maintaining a working silicon back-end.

One can also consider such a high-level language as an interface to the asynchronous world. Therefore several back-ends are available as target, ranging from simple variations in handshake protocols and circuit implementation styles to entirely different operations characteristics as Burstmode circuits.

The fact that we target a high-level hardware description language built for design of asynchronous circuits, means that the designer, if unhappy with parts of the design generated by the behavioral synthesis tool, can either replace these parts with his own designs or directly modify these parts to improve the characteristics of the resulting circuit.

Low power datapaths Our approach targets the generation of low-power datapaths, where computational intensive functional units with large combinatorial depths or that have a large load capacitance through a large number of output connection, can be isolated by the use of non-transparent latches.

8.2 Perspective on the approach

Over the last decade asynchronous design has slowly but surely moved into industry scale designs and has found its way into commercial applications by two primary driving forces:

Application domain There are a number of applications for which one or more of the properties of asynchronous design is a requirement. Examples are; control circuits on analog circuitry, where the clock would introduce noise to the analog circuitry, and smartcards where the circuit only has access to power when used and often in very unreliable form. Most of these circuits are currently small and are manageable for the designer to optimize manually. However as we have seen our synthesized circuits either outperforms or performs equally well to small customized circuits i.e. the GCD algorithm. Furthermore for these application domains the circuit delay constraint is usually easy to meet, leaving a large room for resource sharing. As the size and computational demands of these circuits increase beyond what can be handled by small customized asynchronous hardware and asynchronous processors, there will be a strong application for our approach here.

The clocking problem Large digital circuits designed using the System on Chip paradigm face large problems when it comes to managing the clock in the final layout generation phase. A solution to this problem is the Globally Asynchronous Locally Synchronous (GALS) approach [56, 34], where the interconnection structure is asynchronous and the computation takes place on small synchronous islands. For the interconnection itself there is usually little computation taking place and a custom designed datapacket routing network will probably outperform a synthesized version, unless the routing-protocol and -algorithm have a sufficiently high complexity. However in the future, it will not be unlikely that some of these synchronous islands will be replaced by fully asynchronous circuit variants. These asynchronous circuits will become the target for the work presented in this thesis.

8.3 Future directions

The benchmark set, upon which we have applied our methods, is a small set of synthesis problems. The next step is to apply our method to a larger "real" circuit and compare with a manually designed asynchronous circuit. A possibility could be a low-power 3D-graphics render engine application for portable devices. The render process is a rather inhomogeneous application inwhich characteristics depend highly on the triangle set upon which it operates [47].

As we have seen the meta-heuristic algorithms are very effective, therefore an interesting direction would be the implementation of a power-aware meta-heuristic simulated annealing algorithm. In particular, this only involves finding a new feasibility algorithm, which fast can generate a power- and time-constrained schedule from a infeasible solution [78, 32] If this is impossible one could simply use the existing feasibility algorithm and add a heuristic cost penalty for those schedules which violate the power constraint. This heuristic could simply be based on finding the maximal violation and look at how many operations violate the constraint and then convert these into the area required to implement these, corresponding to executing them at another point in time.

The next improvement concerns the cost function, which we use to compute the area cost during design space exploration. Currently only the FU area is accounted for and we need to make a better modeling of the target circuit including the latch area, interconnect (multiplexor, demultiplexor) area and the area required to implement the controller [68, 37].

Asynchronous circuits operate in continuous time and it would be natural to apply some of the continuous time scheduling algorithms, and compare with the schedules from discrete time. This will investigate if there is a need to include such algorithms and which are the most appropriate for asynchronous circuit design [4, 3].

For certain critical sub-algorithms a specific manual design effort will lead to a significant performance advantage. If such a sub-algorithm is sufficiently common to warrant the design effort it could be made available to the target resource library. These more "complex" operators will be able to enter into our task-level CDFG synthesis algorithm as a DFG fragment. It would be necessary to be able to identify these special fragments in the CDFG [60].

Many of the algorithms, which with advantage can be implemented as asynchronous circuits, are very dynamic in nature. The one-to-one mapping of the CDFG to an asynchronous circuit resembles this as it is a very "elastic" computation. Whereas the schedules produces by the behavioral synthesis algorithms considered in this thesis are static. These algorithm operate by finding the near global optimum by the information available at compile time. However a lot of information is not available at compile time; the path through the conditional parts of the algorithm and conditionally repetitive parts. One approach would be to take advantage of the asynchronous nature and look into methods for making the control of the circuit more dynamic, perhaps even a primitive form of dynamic scheduling.

Bibliography

[1] J. Monteiro, S. Devadas, P. Ashar and A. Mauskar. Scheduling techniques to enable power management. In *Proceedings of the 33rd conference on Design automation*, 1996.

[2] H. Zheng, B. Bachman and C. Myers. Architectural synthesis of timed asynchronous systems. In *International Conference on Computer Design (ICCD '99)*, pages 354–363, Washington - Brussels - Tokyo, October 1999. IEEE.

[3] B. M. Bachman, H. Zheng, and C. J. Myers. Architectural synthesis of timed asynchronous systems. In *Proc. ICCD'99 (IEEE International Conference on Computer Design: VLSI in Computers and Processors)*, pages 354–363, October 1999.

[4] Rosa M. Badia and Jordi Cortadella. High-level synthesis of asynchronous systems: Scheduling and process synchronization. In *Proc. European Conference on Design Automation (EDAC)*, pages 70–74. IEEE Computer Society Press, February 1993.

[5] J. Liu, P.H. Chou, N. Bagherzadeh and F. Kurdahi. A constraint-based application model and scheduling techniques for power-aware systems. In *Proceedings of the ninth international symposium on Hardware/software codesign*, 2001.

[6] A. Bardsley. *Implementing Balsa Handshake Circuits.* PhD thesis, Department of Computer Science, University of Manchester, 2000.

[7] 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.

[8] A. Bardsley and D. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000.

[9] Peter A. Beerel, Wei chun Chou, and Kenneth Y. Yun. A heuristic covering technique for optimizing average-case delay in the technology mapping of asynchronous burst-mode circuits. In *Proc. European Design Automation Conference (EURO-DAC)*, September 1996.

[10] E.Y. Chung, L. Benini and G.de Micheli. Dynamic power management using adaptive learning tree. In *Proceedings of the IEEE/ACM international conference on Computer-aided design, p.274-279*, 1999.

[11] 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. IEEE Computer Society Press, 1988.

[12] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[13] Tobias Bjerregaard, Shankar Mahadevan, and Jens Sparsø. A channel library for asynchronous circuit design supporting mixed-mode modeling. In *Proceedings of the Fourteenth International Workshop on Power and Timing Modeling, Optimization and Simulation, PATMOS2004*, 2004.

[14] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 149–158. IEEE Computer Society Press, April 2004.

[15] Ivan Blunno and Luciano Lavagno. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 84–92. IEEE Computer Society Press, April 2000.

[16] Alex Branover, Rakefet Kol, and Ran Ginosar. Asynchronous design by conversion: Converting synchronous circuits into asynchronous ones. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 870–875, February 2004.

[17] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991.

[18] C. Gi, Lyuh, Tewhan and Kim. High-reliability, low energy microarchitecture synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):364–375, 2003.

[19] J.M. Chang and M. Pedram. Register allocation and binding for low power. In *IEEE conference on Design Automation Conference, DAC95*, 1995.

[20] Tiberiu Chelcea and Steven M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, June 2002.

[21] D. Chen and J. Cong. Register binding and port assignment for multiplexer optimization. In *Proceedings of the Asia Pacific Design Automation Conference*, 2004.

[22] J. Cortadella and R. M. Badia. An asynchronous architecture model for behavioral synthesis. In *Proc. European Conference on Design Automation (EDAC)*, pages 307–311. IEEE Computer Society Press, 1992.

[23] J. Cortadella, R. M. Badia, E. Pastor, and a: Pardo. Achilles: a high-level synthesis system for asynchronous circuits. In D. D. Gajski, editor, *Proc. 6th International Workshop on High-Level Synthesis*, pages 87–94. Univ. California, 1992.

[24] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces.* Springer-Verlag, 2002.

[25] J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. From synchronous to asynchronous: an automatic approach. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1368–1369, February 2004.

[26] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[27] D. Kang, S. Crago and J. Suh. Power-aware design synthesis techniques for distributed real-time systems. In *Proceedings of ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, 2001.

[28] Jim Crenshaw and Majid Sarrafzadeh. Low power driven scheduling and binding. In *Great Lakes Symposium on VLSI '98*, 1998.

[29] A. Dasgupta. High-reliability, low energy microarchitecture synthesis. *IEEE Transactions on Computer-Aided Design*, 17:1273–1280, 1998.

[30] A. Dasgupta and R. Karri. Simultaneusly scheduling and binding for power minimization during microarchitecture synthesis. In *Proceedings of the 1995 international symposium on Low Power Design*, 1995.

[31] A. Davoodi and A. Srivastav. Effective graph theoretic techniques for the generalized low power binding problem. In *International Symposium on Low Power Electronics and Design, ISLPED03*, 2003.

[32] D.E. Goldberg, editor. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[33] Jack B. Dennis. Data Flow Computation. In *Control Flow and Data Flow — Concepts of Distributed Programming, International Summer School*, pages 343–398, Marktoberdorf, West Germany, July 31 – August 12, 1984. Springer, Berlin.

[34] 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.

[35] Jo Ebergen. Squaring the FIFO in GasP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 194–205. IEEE Computer Society Press, March 2001.

[36] D. Edwards and A. Bardsley. Balsa – an asynchronous hardware synthesis system. In J. Sparsø and S. Furber, editors, *Principles of asynchronous circuit design – A systems perspective*, chapter 9–12, pages 155–218. Kluwer Academic Publishers, 2001.

[37] Y.M. Fang and D.F. Wong. Simultaneous-functional unit binding and floorplanning. In *In Digest of Technical Papers, International Conference on Computer-Aided Design (ICCAD))*, 1994.

[38] Karl M. Fant and Scott A. Brandt. NULL conventional logic: A complete and consistent logic for asynchronous digital circuit synthesis. In *International Conference on Application-specific Systems, Architectures, and Processors*, pages 261–273, 1996.

[39] Jiong Luo, Lin Zhong, Yunsi Fei and Niraj K. Jha. Register binding based rtl power management for control- flow intensive designs. Technical report, Dept. of Electrical Engineering Princeton University, Princeton., 1999.

[40] R. M. Fuhrer and S. M. Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines Algorithms and Tools.* Kluwer Academic Publishers, June 2001. ISBN 0-7923-7425-8.

[41] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[42] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi. Optimization by simulated annealing. *Science*, pages 671–680, 1983.

[43] Fred Glover. Tabu search. *ORSA Journal on Computing*, pages 190–206, 1989.

[44] F. Gruian and K. Kuchcinski. Operation binding and scheduling for low power using constraint logic programming. In *IEEE EUROMICRO 98*, 1998.

[45] S. Gupta and S.Katkoori. Force-directed scheduling for dynamic power optimization. In *IEEE/ISVLSI Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2002.

[46] J. P. Hammerstoft. High-level synthesis of asynchronous circuits from control data flow graphs. Master's thesis, IMM-thesis-2001-44, Technical University of Denmark, Dept. of Informatics and Mathematical Modelling, August 2001. (In Danish).

[47] H. Holten-Lund. *Design for scalability in 3D computer graphics architectures.* PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2002.

[48] Hans Jacobson, Erik Brunvand, Ganesh Gopalakrishnan, and Prabhakar Kudva. High-level asynchronous system design using the ACK framework. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 93–103. IEEE Computer Society Press, April 2000.

[49] R. Rim, M. Mujumdar, A. Jain and R. de Leone. Optimal and heuristic algorithms for solving the binding problem. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(2):211–225, 1994.

[50] J. S. Jensen. High-level synthesis of asynchronous circuits. Master's thesis, IT-E-840, Technical University of Denmark, Dept. of Information Technology, June 2000. (In Danish).

[51] J.F. Sowa, editor. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* Brooks cole Publishin Co, 1999.

[52] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, pages 256–278, 1974.

[53] Joep Kessels, Ad Peeters, Torsten Kramer, Markus Feuser, and Klaus Ully. Designing an asynchronous bus interface. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108–117. IEEE Computer Society Press, March 2001.

[54] Euiseok Kim, Jeong-Gun Lee, and Dong-Ik Lee. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 104–113. IEEE Computer Society Press, April 2000.

[55] C.M. Krishna and Y.H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, 2000.

[56] 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.

[57] Lars Kruse, Eike Schmidt, Gerd Jochens, Ansgar Stammermann, Arne Schulz, Enrico Macii, and Wolfgang Nebel. Estimation of lower and upper bounds on the power consumption from scheduled data flow graphs. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(1):3–15, 2001.

[58] Jer-Min Jou, Shiann-Rong Kuang and Ren-Der Chen. Clique partitioning based integrated architecture synthesis for vlsi chips. In *Proceedings of International Symposium on VLSI Technology, Systems, and Applications, pages 58-62*, 1993.

[59] P. Kudva, G. Gopalakrishnan, and V. Akella. High level synthesis of asynchronous circuit targeting state machine controllers. In *Asia-Pacific Conference on Hardware Description Languages (APCHDL)*, pages 605–610, 1995.

[60] G. Lakshminarayana and N. K. Jha. Synthesis of power-optimized and area-optimizaed circuits from hierarchical behavoiral descriptions. In *In proceedings of 35th annual ACM IEEE conference on Design aytomation*, 1998.

[61] K.S. Khouri, G. Lakshminarayana and N.K. Jha. Impact: A high-level synthesis system for low power control-flow intensive circuits. In *In proceedings Design Automation and Test in Europe DATE'98*, 1998.

[62] S.C. Leung and H.F. Li. A syntax-directed translation for the synthesis of delay-insensitive circuits. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 2(2):196–210, 1994.

[63] K.-J. Lin and C.-S. Lin. Removing CSC violations in asynchronous circuits by delay padding. *IEE Proceedings, Computers and Digital Techniques*, 143(6):413–420, November 1996.

[64] T.Kim, N. Yonezawa, J.W.S. Liu and C.L. Liu. A scheduling algorithm for conditional resource sharing - a hierachical reduction approach. *IEEE Trans. Computer Aided Design Integrated Circuits Syst*, 13(4):425–437, 1994.

[65] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev. Visualization and resolution of coding conflicts in asynchronous circuit design. In *Proc. Design, Automation and Test in Europe (DATE).* IEEE Computer Society Press, March 2003.

[66] Michael R. Garey and David S. Johnson, editor. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H Freeman, 1979.

[67] G. De Micheli. *Synthesis and optimization of digital circuits.* McGraw-Hill, 1994.

[68] H. Mecha, M. Fernandes, F. Tirade, J. Septien, D. Motes and K. Olcoz. A method for are estimation of datapath in high level synthesis. *IEEE Trans. Comput. Aided Des. Integ. Circuits Syst*, 15(2):258–265, 1996.

[69] E. Musoll and J. Cortadella. High-level synthesis techniques for reducing the activity of functional units. In *Proceedings of ISLPD95*, 1995.

[70] E. Musoll and J. Cortadella. Scheduling and resource binding for low power. In *Proceedings of the 8th international symposium on System synthesis*, 1995.

[71] Chris J. Myers. *Asynchronous Circuit Design.* John Wiley & Sons, July 2001. ISBN: 0-471-41543-X.

[72] N.H.E. Weste and K. Eshraghian, editor. *Principles of CMOS VLSI Design, A systems perspective.* aw, 1993.

[73] S. F. Nielsen, J. Sparsø, J. Madsen, J. Hammerstoft, and J. S. Hansen. High-level synthesis of asynchronous circuits from control data flow graph representations. In *Second ACiD-WG Workshop (of the European Commission's fifth Framework Programme)*, January 2002.

[74] Sune F. Nielsen and Jan Madsen. Power constrained high-level synthesis of battery powered digital systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 2003.

[75] Sune F. Nielsen, Jens Sparsø, and Jan Madsen. High-level synthesis of asynchronous circuits from control data flow graph representations. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, January 2002.

[76] Enric Pastor, Jordi Cortadella, Alex Kondratyev, and Oriol Roig. Structural methods for the synthesis of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 17(11):1108–1129, November 1998.

[77] P.G. Paulin and J.P. Knight. Scheduling and binding algorithms for high-level synthesis. In *Proceedings of the 26Sth ACM/IEEE Design Automation Conference (DAC),p.1-6*, 1989.

[78] P.J.M. Van Laarhoven and E.H.L. Aarts, editor. *Simulated Annealing: Theory and Practice.* Kluwer Academic Publishers, 1987.

[79] P.J.M Van Laarhoven and E.H.L. Aarts, editor. *Simulated Annealing and Boltzmann Machines.* John Wiley and Sons, 1989.

[80] I. Hong, M. Potkonjak and M.B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998.

[81] A. Chandrakasan, M. Potkonjak, J. Rabaey and R. W. Brodersen. Hyper-lp:a system for power minimization using architectural transformations. In *International Conference on Computer-Aided Design, pp.300-303*, 1992.

[82] A. Chandrakasan, R. Mehra, M. Potkonjak, J. Rabaey and R. W. Brodersen. Optimizing power using transformations. In *IEEE Transactions on CAD, Vol. 14, No. 1, pages 12–31*, 1995.

[83] A. Raghunathan and N. Jha. An ilp formulation for low power based on minimizing switched capacitance during datapath allocation. In *IEEE Symposium on Circuits and Systems*, 1995.

[84] N. Ranganathan and A.K. Murugavel. Advances in embedded software scheduling techniques: A low power scheduler using game theory. In *IEEE/ACM/IFIP international conference on Hardware/Software codesing and system synthesis*, 2003.

[85] M. Renaudin, P. Vivet, and F. Robin. A design framework for asynchronous/synchronous circuits based on CHP to HDL translation. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 135–144, April 1999.

[86] S. M. Nowick and M. B Josephs and C. H. (Kees) van Berkel (editors). Special Issue on Asynchronous Circuits and Systems. *Proceedings of the IEEE*, 87(2), February 1999.

[87] Sabih G. Gerez, editor. *Algorithms for VLSI Design*. Kluwer Academic Publishers, 1999.

[88] M. Sacker, A. Brown, P. Wilson, and A. Rushton. A general purpose behavioural asynchronous synthesis system. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 125–134. IEEE Computer Society Press, April 2004.

[89] D. Shin and K. Choi. Low power high level synthesis by increasing data correlation. In *IEEE/ACM/IFIP international conference on Hardware/Software codesing and system synthesis*, 1997.

[90] Gerald E. Sobelman and Karl Fant. CMOS circuit design of threshold gates with hysteresis. In *Proc. International Symposium on Circuits and Systems*, pages 61–64, June 1998.

[91] D. Sokolov, A. Bystrov, and A. Yakovlev. STG optimisation in the direct mapping of asynchronous circuits. In *Proc. Design, Automation and Test in Europe (DATE)*. IEEE Computer Society Press, March 2003.

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

[93] Sune F. Nielsen, Jens Sparsø and Jan Madsen. Towards behavioral synthesis of asynchronous circuits - an implementation template targeting syntax directed compilation. In *Proc. EUROMICRO DSC*, Aug 2004.

[94] L. Kruse, E. Schmidt, G. Jochens, A. Stammermann and W. Nebel. Lower bound estimate for low power high-level synthesis. In *Proceedings of ISSS 2000*, 2000.

[95] Stephen A. Ward and Robert H. Halstead, editor. *Computation structures*. MIT Press, 1990.

[96] Leon Stok. *Architectural Synthesis and Optimization of Digital Systems*. PhD thesis, Eindhoven University of Technology, 1991.

[97] Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, March 2001.

[98] Michael Theobald and Steven M. Nowick. Fast heuristic and exact algorithms for two-level hazard-free logic minimization. *IEEE Transactions on Computer-Aided Design*, 17(11):1130–1147, November 1998.

[99] Tseng and D.P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, CAD*, 5(3):379–395, 1986.

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

[101] M. van der Korst, A. Peeters, and H. Schols, editors. *Design and Implementation of Asynchronous Circuits*. Koninklijke Nederlandse Akademie van Wetenschappen, North-Holland, June 1992. Proceedings of workshop Amsterdam, 10–14 November 1991.

[102] D. Rakhmatov, S. Vrudhula and C. Chakrabarti. Battery-conscious task sequencing for portable devices including voltage/clock scaling. In *Proceedings of the 39th conference on Design automation*, 2002.

[103] K. Wakabayashi and T. Yoshimura. A resource sharing control synthesis method for conditional branches. In *In Digest of Technical Papers, International Conference on Computer-Aided Design (ICCAD))*, 1989.

[104] Catherine G. Wong and Alain J. Martin. High-level synthesis of asynchronous systems by data-driven decomposition. In *Proc. ACM/IEEE Design Automation Conference*, pages 508–513, June 2003.

[105] T. Yoneda, H. Onda, and C. Myers. Synthesis of speed independent circuits based on decomposition. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 135–145. IEEE Computer Society Press, April 2004.

[106] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits: Part I and II. *IEEE Transactions on Computer-Aided Design*, 18(2):101–117, 118–132, February 1999.

[107] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits: Part I (specification and hazard-free implementation). *IEEE Transactions on Computer-Aided Design*, 18(2):101–117, February 1999.

[108] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits: Part II (automatic synthesis). *IEEE Transactions on Computer-Aided Design*, 18(2):118–132, February 1999.

[109] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 346–350. IEEE Computer Society Press, October 1992.

[110] L. Zhong and N.K. Jha. Interconnect-aware high-level synthesis for low power. In *IEEE ICCAD p.110-117*, 2002.