High-Level Synthesis of Asynchronous Systems: Scheduling and Process Synchronization*

Rosa M. Badia Jordi Cortadella

Polytechnic University of Catalonia, Dept. of Computer Architecture.
Campus Nord, Modul D4. Gran Capita s/num. Barcelona, E-08071

Abstract

Asynchronous systems are gaining acceptance as the size and complexity of digital circuits increase. Concordantly, synthesis tools for asynchronous systems must be developed to make design process easier. This paper aims at the definition of basic concepts for scheduling algorithms and control synthesis in high-level synthesis of asynchronous circuits. Two scheduling strategies are presented and evaluated. Experiments on different benchmarks show that efficient asynchronous schedules can be obtained. Control is modelled in a distributed fashion with Local Controllers synchronizing between them by means of handshaking protocols.

1 Introduction

Asynchronous circuits present properties that meet the requirements for large, complex systems [1, 2]: no clock skew, modular interconnectivity, low peak currents, and performance determined by average processing speeds. The design of asynchronous systems based on self-timed circuits [1] has been more broadly accepted in the last years. Most work on design automation for asynchronous systems has been focused to logic synthesis of sequential machines [3]. Currently, significant effort is being invested in the synthesis of hazard-free circuits from *Signal Transition Graphs* [2], initially proposed by Chu [4] to describe the behavior of asynchronous sequential machines.

Other approaches synthesize asynchronous circuits from high-level specifications by syntax-directed translation, according to production rule sets. In [5] and [6] a similar strategy is used to translate CSP into delay-insensitive circuits. We cannot consider, however, these approaches within the category of *high-level synthesis*, since no attempt is done to improve the quality of the circuit (size and performance) by using optimization techniques like *operation scheduling* and *hardware allocation* [7]. Syntax-directed translation generates circuits whose size depends linearly on the size of the input description [5].

The efforts on *high-level synthesis* [7] have been mainly focused to synchronous designs. A clear evidence of this tendency is that the proposed *scheduling algorithms* [8, 9] are based on the concept of *control step*—i.e. time is measured in *cycles*, and cycle time is determined by the *worst-case delay* of all the operations executed in a control step. Only Ku and De Micheli [10] consider the possibility of having synchronous operations with unbounded delays.

*Work funded by CYCIT TIC 91-1036 and ACiD-WG (Esprit 7225)

From the point of view of the timing model used for operation scheduling and control synthesis, asynchronous systems present two significant differences:

- Time is considered as a *continuous variable* and initiation and completion of operations are *events* that can occur at any instant.
- Operations have *variable, data-dependent delays*.

Therefore, different scheduling strategies must be conceived if an asynchronous timing model is considered.

This paper aims at the definition of basic concepts, data structures, and primitive functions for asynchronous scheduling algorithms and control synthesis.

The paper is organized as follows. Section 2 describes the architecture model considered for the asynchronous execution of operations. Section 3 presents an overview of a high-level synthesis system. Section 4 defines the basic data structures and functions proposed for scheduling algorithms and presents two algorithms for operation scheduling. Section 5 describes the connection between module binding and process synchronization and how control can be synthesized. Conclusions and future work are presented in section 6.

2 Asynchronous Architecture Model

A high-level synthesis system requires a target architecture model for the mapping of high-level objects (operations, variables, data transfers) into hardware modules (ALUs, registers, multiplexors). This section presents a short summary of the architecture model proposed in [11] (only details referring to operation scheduling and control synthesis will be described).

2.1 Data-Path

The data-path is composed of self-timed blocks (ALUs, registers, multiplexors, etc) synchronized by means of a handshaking protocol implemented with two signals: *request* and *completion* [1]. Registers are implemented as latches (the *request* signal indicates when latching must be initiated). Each hardware module is considered a process which executes operations and synchronizes with other processes when data transfers are required.

2.2 Distributed Control

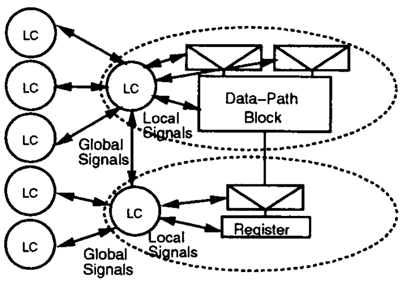

Figure 1: Local Controllers organization

Control is completely distributed in such a way that for each data-path block (or a group of data-path blocks) there is a *local controller* (LC). A local controller has two types of handshaking signals (see figure 1):

- *Local signals*: *request* and *completion* signals for the synchronization with the data-path block being controlled and other signals such as the operation code for ALUs or the selection code for multiplexors.
- *Global signals* for the synchronization associated with data transfers between blocks.

The granularity of the control distribution may vary for the sake of the circuit performance. Hereafter and without loss of generality, we will consider that an LC exists for each computation block (or register) and its input multiplexors.

The execution of an operation has the following steps:

1 Input data are read from registers.

2 Multiplexors transfer input data to their corresponding functional unit input.

3 Operation is executed in a functional unit.

4 Multiplexors transfer output data to a register

5 Output data is latched into a register.

3 High-Level Synthesis: an Overview

The system input is a behavior description of the circuit described by a Control Data Flow Graph (CDFG). The first step performed is *Scheduling* and *Allocation* (figure 2). Allocation selects the number and type of hardware modules that will compose the data-path. In operation scheduling, operations are distributed through the time space and are assigned to a type of functional unit. The output is a Scheduled Data Flow Graph (SDFG), which is a modification of the previous CDFG containing information related to scheduling and allocation. In *Resource Binding* operations are bound to hardware modules. The output is a Bound Data Flow Graph (BDFG) where each operation has been bound to an FU instance and each variable to a register. After binding the set of operations that will be executed in each process is totally defined and the behavior of the local controllers can be derived as it is explained in section 5.

4 Scheduling

We will represent the scheduling problem with a *data flow graph* (DFG), $G(V, E)$, where vertices and edges denote operations and dependencies respectively. For each vertex $v \in V$ we will use the following terminology:

v_o :	executed *operation*
v_s, v_c :	*start* and *completion* time (after being scheduled)
v_f :	functional unit type that executes v_o
$pred(v) =$	$\{u \mid (u,v) \in E\}$
$succ(v) =$	$\{u \mid (v,u) \in E\}$

For now on, the following assumptions will be considered (which are close to reality):

- Since control is evenly distributed all over the circuit, it is assumed that delays introduced by the LCs are constant.
- Latching delays are equal and constant.

Thus, synchronization and latching delays can be included in the delay of each operation.

The library of functional units (FUs) available for a given technology is represented by the *Delay Matrix* δ (see figure 4). Each element $\delta_{f,o}$ indicates the delay for the execution of operation o by the FU type f ($\delta_{f,o} = \infty$ indicates that o is not implemented by f). We will denote by $FU(o)$ the set of FU types implementing o. In the environment of asynchronous systems, where execution delays are data-dependent, the *Delay Matrix* δ represents average delays. Thus, scheduling times obtained by using δ must be considered as **estimated average processing delays**[1].

A *resource vector*, $R = < |f_1|, |f_2|, \ldots, |f_n| >$, represents a set of resources available for allocation, where $|f_i|$ indicates the number of instances of the FU type f_i ($|f_i| > 0$). Given a resource vector R, we define the *average delay* for operation o, $\overline{\delta_o}$, as:

$$\overline{\delta_o} = \frac{\sum_{f_i \in FU(o)} |f_i| \delta_{f_i,o}}{\sum_{f_i \in FU(o)} |f_i|}$$

$\overline{\delta_o}$ is a pre-scheduling estimation of the expected execution delay for operation o, assuming that any

[1]A *Worst-Case Delay Matrix* would be also required to calculate worst-case processing delays if timing constraints were imposed in the DFG. Dealing with timing constraints is out of the scope of this paper.

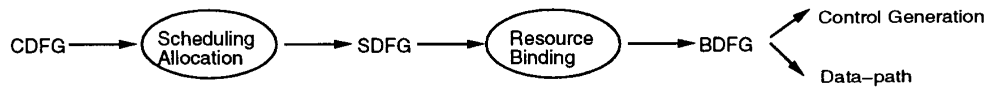

Figure 2: High-level synthesis steps

vertex v can be equiprobabilistically assigned to any $f \in FU(v_o)$.

Finding an asynchronous schedule means defining a **partial ordering** of the vertices and **allocating** each operation to a type of FU so that the total **estimated processing delay** is minimized.

4.1 Data Structures

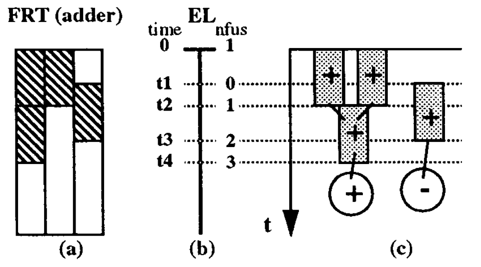

Figure 3: (a) Frame Reservation Table; (b) Event List; (c) Partially-scheduled DFG

A *Frame Reservation Table* (FRT_f) is a data structure bound to a type of FU f and reports the number of active instances of type f at each time instant. FRT_f is updated each time a new operation is scheduled and bound to an FU of type f. The number of active instances can never be greater than $|f|$.

FRT_f can be represented as an *Event List* (EL_f). EL_f is formed by a list of pairs $< time_i, nfus_i >$ ordered by time, where $nfus_i$ indicates the number of available (non-active) FUs of type f from $time_i$ to $time_{i+1}$. Figures 3.a and 3.b depict the FRT_f and EL_f corresponding to the scheduled operations shown in 3.c.

Two functions have been defined for managing the *Event List* [12]:

- `start_time = find_free_interval`
  `(EL, min_start_time, delay)`

  This function seeks in the event list `EL` for the first time interval of duration `delay` with `start_time` $\geq$ `min_start_time` such that it has at least one free FU.

- `reserve_interval (EL, start_time, delay)`

  This function reserves a time interval of duration `delay` starting at `start_time`.

4.2 Event-List-Based Scheduling

Two algorithms for operation scheduling in asynchronous systems are presented in this section: *ELS* (Event-List Scheduling) and *ELLAS* (Event-List Look-Ahead Scheduling). Both algorithms select vertices to be scheduled according to a priority function. They differ in the calculation of the priority function: in the former the priority of each vertex is calculated at the beginning of the algorithm, while in the latter it is dynamically evaluated as a result of a look-ahead scheduling function.

During the execution of a scheduling algorithm the set of nodes of the DFG can be partitioned into three sets: the *Ready Set (RS)*, the *Scheduled Set (SS)*, and the *Non-Scheduled Set (NSS)*. *SS* contains all the vertices already scheduled. A vertex v belongs to *RS* if it has not been scheduled yet and for each $u \in pred(v)$, $u \in SS$. The rest of vertices belong to *NSS*. The scheduling algorithm is next described.

```
ELS ($G(V,E)$, $\delta$, $R$) {
for each $f$ in the library do initialize_event_list ($EL_f$, $|f|$);
calculate_path_lengths_to_end ($G$, $\delta$, $R$);
$RS$ = source_vertices($G$); $NSS = V - RS$; $SS = \emptyset$;
while $RS \neq \emptyset$ {
    $v$ = max_priority_vertex($RS$);
    min_start = $\max_{u \in pred(v)} u_c$;
    for each $f \in FU(v_o)$ do {
        min_start$_f$ = find_free_interval ($EL_f$, min_start, $\delta_{f,v_o}$);
        completion$_f$ = min_start$_f$ + $\delta_{f,v_o}$;
    }
    $f_{min} = f$ such that
        (completion$_{f_{min}}$ = $\min_{f \in FU(v_o)}$ completion$_f$);
    reserve_interval ( $EL_{f_{min}}$, min_start$_{f_{min}}$, $\delta_{f_{min},v_o}$);
    $v_s$ = min_start$_{f_{min}}$; $v_c$ = completion$_{f_{min}}$; $v_f = f_{min}$;
    $SS = SS \cup \{v\}$; fireable = $\{v \in NSS \mid \forall u \in pred(v), u \in SS\}$;
    $RS = (RS - \{v\}) \cup$ fireable; $NSS = NSS -$ fireable;
}
}
```

Similarly to *list scheduling*[9], this algorithm calculates, first, a priority for each vertex of the DFG. Then vertices in *RS* are scheduled in order according to their priority. The priority of each vertex v, v_p, is calculated as the path length to the end of the DFG, assuming that each vertex v is executed in $\overline{\delta_{v_o}}$ time. The calculation of v_p can be done recurrently from sink to source vertices as follows:

$$v_p = \max_{u \in succ(v)} u_p + \overline{\delta_{v_o}}$$

First, all the event lists are initialized with their corresponding number of resources ($|f|$), and the priority (path length to end) of each vertex is calculated. The main loop of the algorithm selects, first, the vertex v with maximum priority in the *Ready Set*. Then, it calculates the completion time achievable by each FU that can execute v_o, and selects that FU, f_{min}, that yields the minimum value. Finally, v is scheduled and bound to f_{min}.

Information about utilization of resources is kept in the event lists, and managed by functions

find_free_interval and **reserve_interval**. When an operation can be executed by more than one type of FU, *ELS* tends to bind vertices with more priority to faster FUs. The time complexity of *ELS* is $O(n \log n)$.

As mentioned before, *ELLAS* dynamically calculates each vertex's priority by using a look-ahead scheduling function. The time complexity of ELLAS is $O(n^3 \log n)$ [12].

4.3 Results

Functional Unit	*delay* +	-	<	*
ALU(◇)	50	50	50	∞
adder(⊕)	35	∞	∞	∞
mult(⊗)	∞	∞	∞	85

Figure 4: Delay Matrix used for the benchmarks.

In this section, the scheduling algorithms previously presented are evaluated. Two benchmarks have been chosen to present the results of the experiments: the *Differential Equation Solver* [8] and the *Fifth-order Wave Digital Filter* [13]. The library used for both benchmarks is represented by the *Delay Matrix* depicted in figure 4.

	Resources	ELS	ELLAS	
		Schedule	Schedule	CPU
Diff. Eq.	⊗ ⊗ ⊗ ⊕ ◇	270 ns	270 ns	0.015 s
	⊗⊗ ⊕ ◇	305 ns	305 ns	0.015 s
	⊗ ⊕ ◇	545 ns	545 ns	0.01 s
Elliptic Filter	⊗⊗ ⊕ ◇◇	740 ns	705 ns	0.12 s
	⊗⊗ ⊕⊕ ◇	705 ns	690 ns	0.12 s
	⊗ ⊗ ⊗ ⊕ ◇◇	720 ns	700 ns	0.12 s

Table 1: Results for the Differential Equation Solver and for the Elliptic Filter with CPU times in a DEC-system 5100 (all CPU times for the ELS are 0.01sec.)

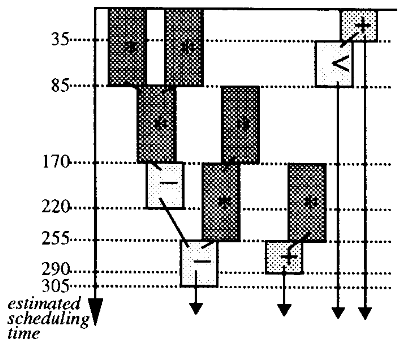

Figure 5: Schedule for the *Diff. Eq. Solver* (⊗⊗⊕◇);

Table 4.3 presents the results obtained for the two benchmarks considering different resource contraints. For the Differential Equation *ELS* and *ELLAS* give the same results. CPU times are similar due to the small size of the problem. Figure 5 depicts the resulting schedule for one of the experiments. For the Elliptic Filter *ELLAS* is superior to *ELS* in most cases, at the cost of higher CPU times (but still moderate). Figure 6 shows the schedule obtained with *ELLAS* for one of the experiments. It is worth to emphasize the skill for binding critical operations to fast FUs and non-critical operations to slow FUs when they can be concurrently executed.

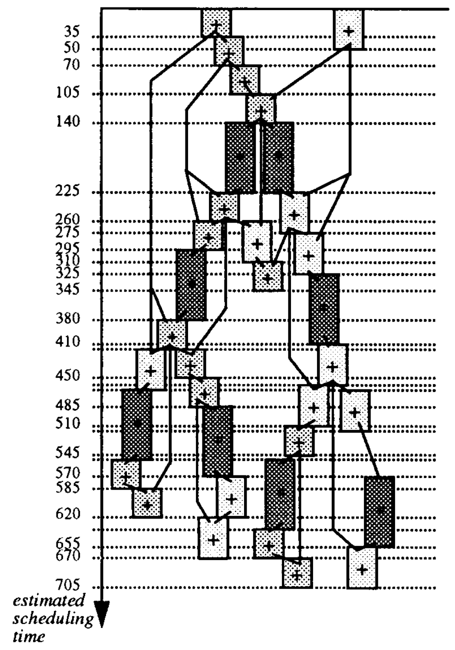

Figure 6: Schedule for the *Elliptic Filter* obtained by *ELLAS* (⊗⊗⊕◇◇)

5 Binding and Process Synchronization

Module binding is performed after scheduling and allocation in order to bind operations to hardware module instances. The criteria used for binding aims at the reduction of the connectivity of the circuit so that routing area and communication delays are minimized. In that respect, binding algorithms already proposed for synchronous circuits can also be used for asynchronous circuits (if they do not use control-step-based approaches) [14].

Once binding is performed, the sequence of operations to be executed in each process is completely defined. Each data transfer between two operations corresponds to a synchronization between processes when the operations are bound to different hardware modules. In figure 7 each arrow corresponds to a synchronization between processes for the scheduling example of figure 5.a.

Each data transfer between a computational block and a register requires an explicit synchronization between the processes corresponding to each of the in-

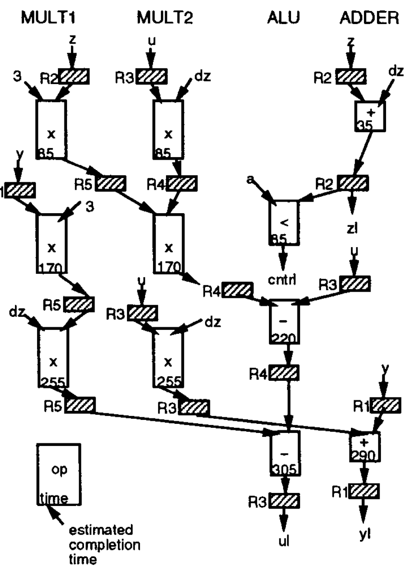

Figure 7: Synchronizations between processes (some register assignments have been replicated to increase the readability of the diagram)

volved hardware modules. Synchronizations are required to assure the sequenciality imposed by data dependencies, as asynchronous systems do not have a global clock that indicates the completion of operations. After the ordering of data transfers and synchronizations has been determined, the behavior of each local controller is derived by defining the transitions of the handshake signals. Signal Transition Graphs (STGs) [4] are used as behavioral description for local controllers. From STGs, a hazard-free circuit can be synthesized for each controller. For more details, we refer the reader to [11], where an approach for the synthesis of distributed asynchronous controllers from high-level descriptions is proposed.

6 Conclusions and future work

As the design of asynchronous circuits is gaining acceptance, tools for high-level synthesis are more necessary. This paper has presented the first approach, to the knowledge of the authors, to scheduling for the high-level synthesis of asynchronous circuits. For an asynchronous timing model, in which no control steps exist, scheduling means defining a partial ordering of the execution of the operations. Basic data structures and primitive functions for the management of *initiation* and *completion* operation events have been defined. Two algorithms, *ELS* ($O(n \log n) - time$) and *ELLAS* ($O(n^3 \log n) - time$) have been proposed and evaluated.

Further research is required in this emerging area. Among the issues not considered in this paper, we mention some of the most significant: scheduling across basic blocks, pipelined functional units, and scheduling under timing constraints. On the other hand, further research is also required in the area of distributed control synthesis for asynchronous systems.

References

[1] C.L. Seitz, *Introduction to VLSI Systems*, Chapter 7, Mead and Conway (Eds.), Addison Wesley, 1981.

[2] T.H. Meng, *Synchronization Design for Digital Systems*, Kluwer Academic Publishers, 1991.

[3] G. Mago, "Realization Methods for Asynchronous Sequential Circuits," *IEEE Trans. on Computers*, Vol. C-20, No. 3, pp. 290-297, March 1971.

[4] T.A. Chu, *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*, Ph.D. thesis, MIT, June 1987.

[5] A.J. Martin, "Compiling Communicating Processes into Delay-insensitive VLSI Circuits," *Distributed Computing*, Vol. 1 (4), Springer-Verlag, pp. 226-234, 1986.

[6] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," *Proc. European Conference on design Automation*, pp. 384-389, Feb. 1991.

[7] M.C. McFarland, A.C. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," *Proc. 25th ACM/IEEE Design Automation Conference*, pp. 330-336, June 1988.

[8] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," *Proc. 24th ACM/IEEE Design Automation Conference*, pp. 195-202, June 1987.

[9] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallet, "Some experiments in local microcode compaction for horizontal machines," *IEEE Trans. on Computers*, vol. C-30, July 1981.

[10] D. Ku and G. De Micheli, "Relative Scheduling Under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits," *IEEE Trans. on Computer-Aided Design*, Vol. 11, No. 6, pp. 696-718, June 1992.

[11] J. Cortadella and R.M. Badia, "An Asynchronous Architecture Model for Behavioral Synthesis," *Proc. European Conference on Design Automation*, pp. 307-311, March 1992.

[12] R.M. Badia and J. Cortadella, "High-Level Synthesis of Asynchronous Digital Circuits: Scheduling Strategies," UPC/DAC Report no. RR-92/6, November 1992.

[13] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms," in *VLSI and Modern Signal Processing*, ed. T. Kailath, pp. 258-264, 1985.

[14] C.J. Tseng and D.P. Siewiorek, "Automated Synthesis of Data Paths on Digital Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-5, no. 3, pp. 379-395, July 1986.