Designing an Asynchronous Processor using Petri Nets

A. Semenov[*], A.M. Koelmans, L. Lloyd, A. Yakovlev[†]

Department of Computing Science,

University of Newcastle upon Tyne, NE1 7RU, England

Abstract

We describe a technique for the design and analysis of a simple asynchronous microprocessor from a Labelled Petri Net specification. The implementation is obtained by means of refinement, transformation and translation. Several versions of the microprocessor design are presented, evaluated and compared. The Petri net based approach allows an interplay of different formal tasks, such as synthesis, verification and performance evaluation, to be carried out within the single modelling framework.

1 Introduction

Petri Nets (PNs) are becoming increasingly attractive as a formalism for the design of hardware systems. The graphical nature of the PN notation makes it more attractive to circuit designers than algebraic notations, which are much less intuitive. PNs are mathematically well founded, and can be used to check for potential hazards in circuits. They can be used as a modelling language to perform formal synthesis and high level analysis of complex processor designs and signal processing chips. It is possible to translate PNs to VHDL, and vice versa for subsets of VHDL, making it possible to integrate PN tools into existing design environments. Many researchers have proposed extensions to the Petri Net notation for the accurate modelling of circuit properties such as timing information. PNs and their closely related notation, Signal Transition Graphs, are extensively used in the area of design of asynchronous circuits, because the event driven nature of PNs closely matches the event driven nature of asynchronous circuits.

[*]Supported by grants from the Research Committee of the University of Newcastle upon Tyne and CVCP.

[†]Supported by EPSRC grant No. GR/J 52327

Several asynchronous processor designs have been completed. Amongst the most recent ones are AMULET1 from Manchester University [6], an asynchronous microprocessor from Caltech [8], an HP Lab's microprocessor called Mayfly [20], and TITAC from Tokyo Institute of Technology [12]. Each of these design groups used their own formalisms during the design process. For example, the microprocessor designed by Alain Martin's group at Caltech used a CSP-like language. CCS [10] formalisms were used to verify the specifications and implementations of finite state machines for the Mayfly distributed memory microprocessor at HP Labs. On the other hand, the AMULET group, led by Steve Furber, designed their first microprocessor virtually without the use of formal methods.

Attempts to model and analyse processors formally were made as early as in 1970 [5]. In this work, Dennis describes the modelling of the CDC 6600 CPU using PNs. Some work has recently been done on modelling and analysis of control circuits in the AMULET microprocessor [13]. A methodology for modelling and analysis of asynchronous circuits using Circuit Petri Nets has been presented in [23]. This work shows that PNs can be successfully used for these purposes. However, the above examples were aimed at modelling of existing (asynchronous) circuits, rather than the design of new circuits from their initial specifications. To our knowledge, the use of PNs and their related formalisms in actual synthesis of hardware has been scarce in the literature. The best known formalism, Signal Transition Graphs (STG) [19, 4], is typically used for the synthesis of asynchronous interface circuits. However, STGs are low-level models, and are not really suitable for synthesis of relatively large circuits at a high level of abstraction.

While the analysis and synthesis of separate modules is, of course, possible with existing STG-based methods, the complete design of an entire processor is a considerably more difficult task. We feel that the best way of "breaking the ice" for the use of PNs in designing a large circuit should begin with a relatively simple, yet sufficiently generic, example. To undertake such a study we wanted to find a suitable synchronous "prototype", which would play the same role for us as the synchronous ARM did for the AMULET group. We decided upon the simple processor design described by Holton in [7]. This processor was used to demonstrate the fundamentals of processor operation. It is a clear and easy to understand example. We organised our asynchronous processor so as to consist of the same operational modules, with the same instruction set as Holton's processor.

We therefore present a design of a simple asynchronous processor which is scalable and can be developed further into a fully operational version. The aim of this work is not to develop a complete hardware device, but to demonstrate design methods which use PNs and their modelling power. We show how PN analysis tools can assist the designer by pointing out particulars of circuit behaviour. In addition, the processor can serve as an ideal testbed for the analysis of different properties, such as timing properties.

This paper is organised as follows. Section 2 briefly describes PNs and their analysis methods. Section 3 outlines Holton's synchronous version of the processor. Sections 4 and 5 describe several asynchronous versions of the processor. In section 6 we estimate the performance of all asynchronous versions. In Section 7 we transform the specification in terms of Labelled PNs (LPNs) into an asynchronous circuit. Section 8 concludes the paper.

2 Petri Nets and their analysis

We present a brief introduction into PN theory here. For a more comprehensive introduction, the reader is referred to, e.g., [17, 11].

Petri net definition. A *marked Petri Net (PN)* is a tuple $N = \langle P, T, F, m_0 \rangle$ where P and T are non-empty sets of *places* and *transitions* respectively, F is a *flow relation* which connects places to transitions and transitions to places, i.e., $F \subseteq (P \times T) \cup (T \times P)$, and m_0 is the *initial marking*. A PN is represented as a graph with two types of nodes: *circles* are used to denote places and *bars*, or boxes, are used for transitions. A *marking* of a PN is depicted with *tokens* (thick dots). A transition is said to be *enabled* under a given marking, if all its input places contain at least one token. An enabled transition can *fire*, producing a new marking. The firing of a transition removes one token from each input place and adds one token into each output place of the transition. The set of markings of a net that can be reached from its initial marking by means of all possible firings of transitions is called the *reachability set* of the net. A Labelled PN (LPN) is a PN N along with a labelling function $L : T \rightarrow A$, which labels each transition with an action name from the alphabet A.

Petri net properties. A PN is said to be *finite* if sets P and T are finite. A PN is said to be *k-bounded* if there exists a k such that at any reachable marking the number of tokens in any place is not greater than k. A 1-bounded PN is called a *safe* PN. The following properties are useful for checking the behavioural correctness of nets specifying asynchronous circuits.

A reachable marking m at which no transition is enabled is called a *deadlock*. A PN is said to be *deadlock-free* if its reachability set includes no deadlocks. Presence of deadlocks is regarded as an error in a system which operates in cycles.

A transition t of a PN is said to be *live* if for any reachable marking m there exists a marking m' reachable from m at which this transition is enabled. A PN is said to be live if every transition is live. This is often called a *strong* form of PN liveness, in which every operation can be activated at some state when the system starts in *any* of its allowable states. This form thus implies cyclicity of all operations. A *weaker* form of liveness requires only that a transition can be enabled at least once in some reachable marking. A transition which is not live usually indicates that some operation of the designed system can never be performed.

A marked PN is said to be *persistent* with respect to some transition t if for any reachable marking m in which t is enabled no other transition t' can be fired, and lead to a marking m' where t is no longer enabled. If there exists a marking at which t' can disable t, then t and t' are said to be in *dynamic conflict*. Clearly, in order to be in dynamic conflict transitions t and t' *must share* at least one input place. This sharing is called a *structural conflict*. A PN is *persistent* if it is persistent with respect to all transitions. Persistency as well as safety are closely related to hazard-free operation of an asynchronous circuit. There are two interpretations of circuit hazards in terms of properties of PNs. For example, if a transition may be disabled by another one, then a signal associated with this transition may be stopped in the process of changing its value. Due to indeterminate timing (any firing delay is assumed to be unbounded but finite) of the signal change, this may produce a hazardous spike on the signal waveform. Similarly, if a place is unsafe, two tokens in it may represent arrival of two consecutive changes of one signal. These changes, one being a rising and the other a falling edge, may arrive close in time and thus cause a spike on an output of the gate associated with the place.

Transitions t_1 and t_2 of a PN are said to be *concurrent* if there exists a marking m at which both transitions are enabled and may be fired at the same time. These two transitions can also fire in any order. Possible orderings of concurrent transitions are called *interleavings*.

Petri net analysis. There are several methods for analysing PN dynamic behaviour. One can build a reachability set which represents all possible states of the system. Analysis using explicit representation of the reachability set is costly – the number of reachable markings may grow exponentially with the number of transitions in the PN.

Several methods have been suggested to overcome the state space explosion. Among those are PN *symbolic traversal* [18], *stubborn set* methods [22] and *PN unfoldings* [9]. PN symbolic traversal uses *implicit representation* of the reachability set in the form of Binary Decision Diagrams [3] (BDDs) which are canonical representations of boolean functions in graphical form. PN symbolic traversal has been shown to be efficient for analysis of "state-based" properties such as freedom from deadlock. However, this method does not allow representation of the relations between transitions. Stubborn set methods use the fact that interleavings of concurrent transitions lead to the same marking. These methods *partially represent* the reachability set. Although efficient in finding deadlocks, they do not produce a complete representation of the reachable state space, and checking for properties other than freedom of deadlocks usually involves exploring other states. PN unfolding represents the full reachability graph using *partial orders* preserving relations between transition occurrences (a transition occurrence is a *unique* event associated with a single act of firing of the transition). Since all reachable markings are represented in the PN unfolding, the concurrency relation for two transitions can easily be obtained. While discussiong the design steps in the next section, we will refer to the use of analysis techniques used in checking the behavioural correctness of the microprocessor.

Unlike ordinary (untimed) PNs, where every transition firing has no specified firing time or delay, a circuit transition is usually associated with an action that takes a finite amount of time. This amount is typically a physical delay associated with a signal change. If two transitions are fired concurrently, the overall time is the maximum of the firing times of the transitions, as opposed to their sum as in the case of sequential operation. A design in which a certain major module is decoupled from the rest of the circuit would be considered more time-efficient. In the following discussion, we observe how time-efficiency can be achieved by introducing a pipelined operation in the system. Such an operation is easily captured by a PN description.

3 Synchronous implementation

The simple 3-bit processor design is described in [7]. Its architectural organisation is reproduced in Figure 1. This design is synchronous. It uses a common clock to synchronise data transfer between processor modules. It consists of the following major operational modules: Instruction Register (IR), Instruction Decoder (ID), Arithmetic and Logic Unit (ALU), Accumulator (Acc), General Register (GR), Program Counter (PC), Address Decoder (AD) and Memory (Mem). All modules are connected to one shared bus through buffers. ID serves as a "processor manager" by configuring

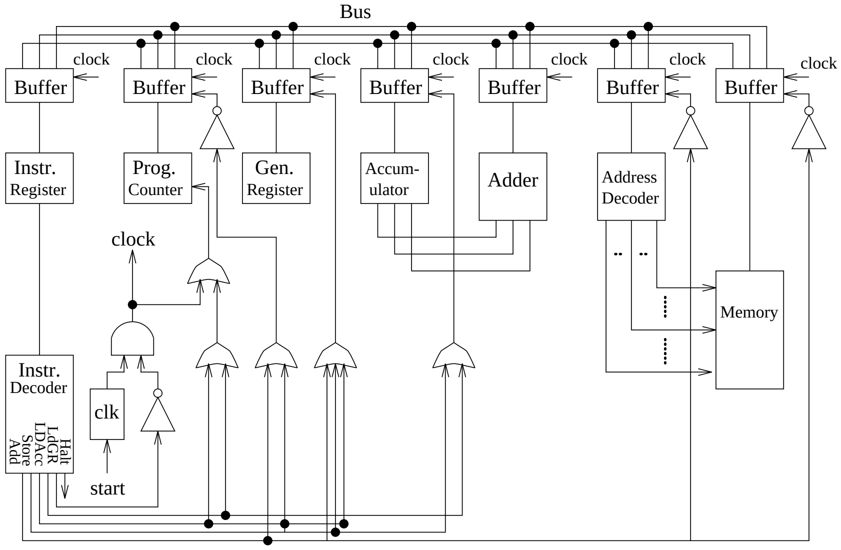

Figure 1: Synchronous implementation of processor.

the processor for execution of the current instruction.

The *operational cycle* of the processor is divided into two stages: *Instruction Fetching* and *Instruction Execution*. The operational cycle always requires four clock cycles. In the first stage, Instruction Fetching, the PC is incremented during the first clock period, and the new value of PC is presented to the Memory. During the second clock period, a word fetched from the Memory is latched in the IR. The processor then enters the Instruction Execution part of its operational cycle. During the third clock period the instruction is decoded and the appropriate modules are connected to the bus. The fourth clock period completes execution of the fetched instruction.

ID determines which modules should be connected to the bus. If an arithmetic instruction is fetched, then the ID connects the ALU and the appropriate registers. If the instruction loads one of the registers, then the appropriate register and memory are connected to the bus, AD is presented with an address, and Memory is signalled to produce the data kept at the address decoded by AD. A “Store” instruction causes GR to be connected to the bus together with AD to load the address in the Memory, and Acc and Memory are then connected to write the data kept in Acc.

This simple example demonstrates some problems common to synchronous circuits. Each module is clocked at *every clock period*. Thus, at every clock period, power needed for driving the clock signal is wasted on those modules that are not involved in the execution of the current step. The clock period is determined by the delay of the *longest* execution cycle. Therefore, the average speed of the processor is bounded by the *worst case* delay. The clock signal requires careful routing on the chip to ensure that the clock arrives in all modules at the same time. This is known as the *clock skew* problem, which is increasingly becoming a major issue in chip designs with a high clock rate. Asynchronous circuits do not have clocks, and thus avoid these problems. In the next sections we will develop an asynchronous version of Holton's processor. By using the PN formalism, we aim to ensure that the final design is functionally correct.

4 Design of an Asynchronous version

Basic design. In order to obtain a comparable asynchronous version of the processor we will use "asynchronous equivalents" of the modules which were used in the synchronous version. The main objective of the first design stage is to produce an LPN which has transitions labelled only with actions of the corresponding modules. During the second stage, we will transform this high-level LPN into an LPN which contains explicit transitions of control elements, and can therefore be translated into a circuit. We restrict ourselves to the instruction set specified in [7], which contains the following operations: "Load Accumulator" (LdAcc), "Load General Register" (LdGR), "Arithmetic Operation" (Arth) and "Store". Note that there is no jump instruction, which is one of the main reasons for the relative simplicity of the processor design.

We start with the initial specification shown in Figure 2. This follows the most abstract specification of the operation of the processor: it alternates between the Instruction Fetch and Instruction Execute modes. Thus the initial specification is simply an LPN with two transitions representing both modes.

Action refinement. We now refine these two transitions. The Instruction Fetch transition is refined into the "PC Increment" (denoted by PC) and "Fetch a Word" operations. "Fetch a Word" can be further decomposed into a pair of transitions, loading the Memory Address Register (MAR) and fetching a word from the Memory (Mem) at the address specified by MAR. We assume that the Memory does not have output latching. It accepts an address along with the accompanying request-for-read signal, and produces an acknowledgement when the data on its outputs is stable. Note that there is no requirement for this signal to be generated as a completion signal; it can simply be implemented as a delay inside the Mem module. Memory has another set of inputs which is used for a write operation. Whenever a write request arrives, data from the write bus is stored at the location specified by MAR. This is acknowledged on a separate wire. The MAR and Memory

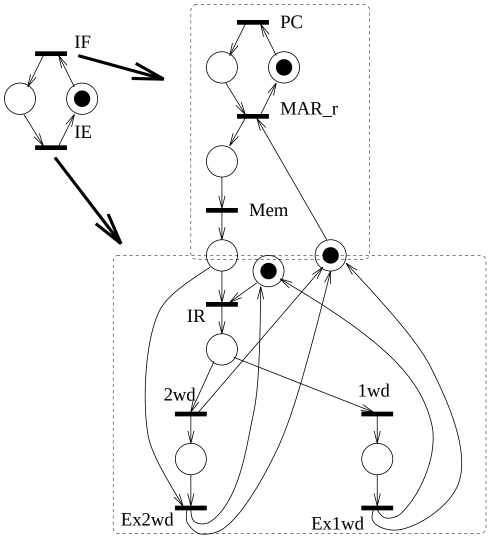

Figure 2: High-level specification of AMP.

modules can therefore be activated in two modes – Instruction Fetch and Instruction Execute. In order to avoid confusion, we will subscribe the Memory module operation with "r", for "read", and "w', for "write", to indicate the mode in which they operate.

Instruction Execute needs careful consideration. The instruction set has two types of instruction — one-word and two-word. When a module signals completion of a one-word instruction, the processor may execute the next instruction. It is fetched from Memory and written into IR using the address in PC. If a two-word instruction, such as "Load Accumulator", is executed, then the next word fetched from Memory contains data. Hence the instruction word must be kept in IR, but an acknowledgement is sent to Memory so that the next word appears. This word is then latched into the appropriate register. Instruction Execute is refined in Figure 2. At this stage, we have only two transitions corresponding to both instruction types. Their refinement is discussed below.

When an instruction is latched in IR, it is decoded by the ID and executed. While the instruction is being executed the contents of IR must not change.

The "Load Accumulator" instruction is refined into "LdAc", representing the decoding of the instruction, and "Acc_dta" which represents the actual latching of the second word in the register. Instruction "Arth" is decomposed into "ALU" and "Acc_res", which corresponds to activation of ALU and latching of the result in Accumulator. "Store" is refined into "MAR_w", which loads MAR with an address at which the data from GR is to be stored, and "Mem_w", which represents storing of the data in Memory. These refinements are shown in Figure 3. Transitions labelled with "Acc_dta" and "Acc_res" correspond to the Accumulator being used in two modes – register loading

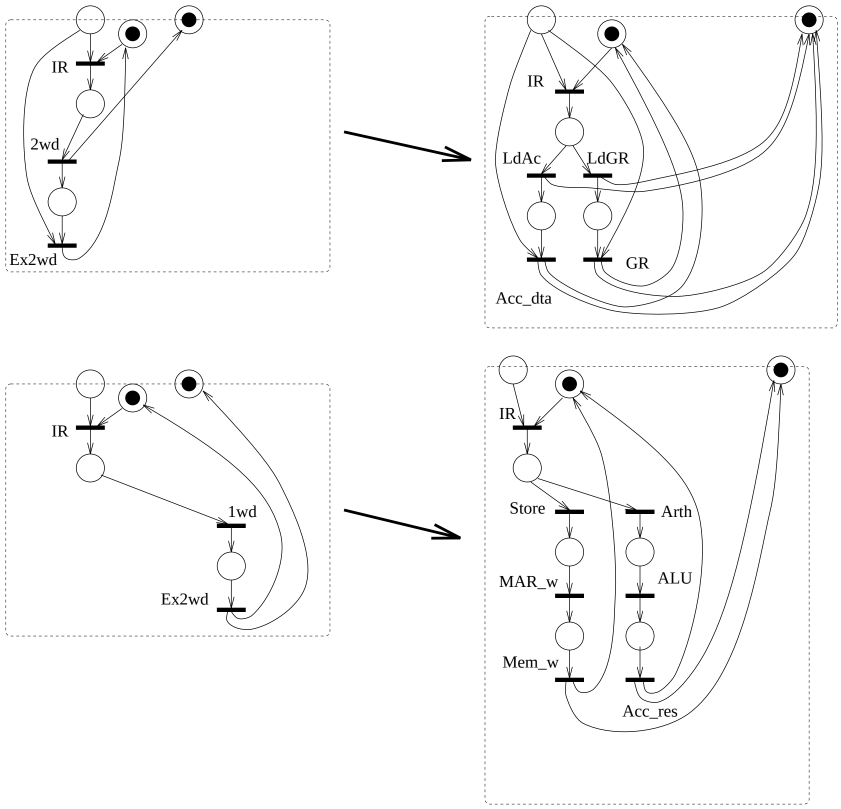

Figure 3: Model refinement.

and arithmetic operation. Note that since there are several transitions corresponding to one module operating in different modes, *mutual exclusion* of these transitions must be guaranteed.

Analysis and improvement of the basic design. We now have an LPN which contains only transitions labelled with actions of modules. Verification of this LPN using the characteristic segment of its unfolding shows that the LPN is live, safe and deadlock-free. Reported non-persistent transitions are transitions representing a *data-dependent choice* of the type of instruction in the instruction decoder.

We now derive *temporal* relations between the transitions of the LPN. Table 1(a) shows these relations for the first version of the processor. Entries marked with "||" represent the fact that two transitions are mutually concurrent. Blank entries represent the mutual exclusion relation between the transitions. Analysis of these relations shows that PC Increment is concurrent to all transitions involved in the execution of instructions.

The analysis also shows that latching data in all multiplexing registers never overlaps with other operations. This LPN therefore represents a behaviour which can be implemented as an asynchronous circuit, and its functionality meets the design specification. In contrast with the synchronous version,

		1	2	3	4	5	6	7	8	9	10	11
1	PC	-		||	||	||	||	||	||	||	||	||
2	MAR_r		-									
3	Mem_r	||		-								
4	MAR_w	||			-							
5	Mem_w	||				-						
6	IR	||					-					
7	ID	||						-				
8	ALU	||							-			
9	Acc_res	||								-		
10	Acc_dta	||									-	
11	GR	||										-

(a)

		1	2	3	4	5	6	7	8	9	10	11
1	PC	-		||	||	||	||	||	||	||	||	||
2	MAR_r		-						||	||		
3	Mem_r	||		-					||	||		
4	MAR_w	||			-							
5	Mem_w	||				-						
6	IR	||					-					
7	ID	||						-				
8	ALU	||	||	||					-			
9	Acc_res	||	||	||						-		
10	Acc_dta	||									-	
11	GR	||										-

(b)

Table 1: Temporal relations between transition for LPNs of versions 1 (a) and 2 (b).

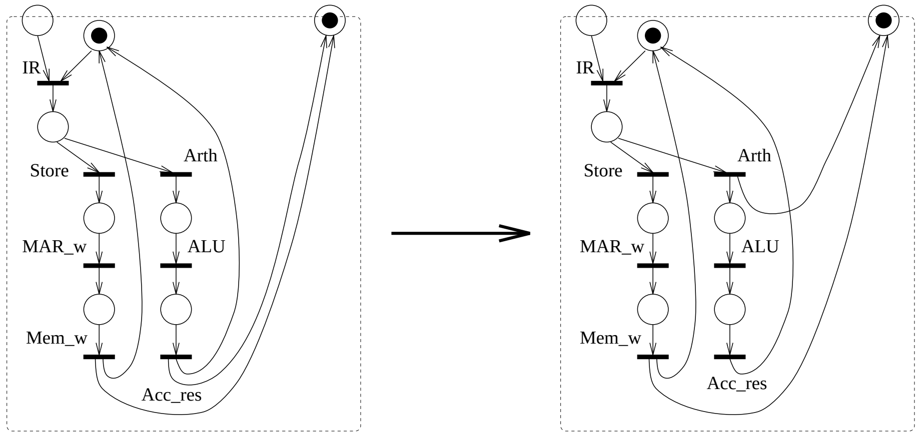

Figure 4: LPN refinement with decoupled ALU action.

the delay of the execution of any particular instruction depends only on the *actual* speed of the modules.

Analysis of the relations between the transitions in the first version of the LPN model shows that PC Increment is the only operation concurrent with the execution of the current instruction. However, any arithmetic instruction can be executed concurrently with fetching the next word from memory. The instruction does not require data from the Memory. Once the “Arth” instruction is latched in IR and decoded in ID, an acknowledgement can be sent to MAR so that it can proceed. Completion of the instruction is acknowledged to IR in order to allow “Arth” to complete. This observation results in a different LPN refinement, which is shown in Figure 4.

Behavioural analysis of this LPN shows that it holds the same properties as the initial LPN. Analysis of the relations between transitions (Table 1(b)) reveals that the execution of an arithmetical operation, including writing into Accumulator, may happen concurrently with loading MAR with a new address and reading the next word from Memory. Thus the average execution time of a program

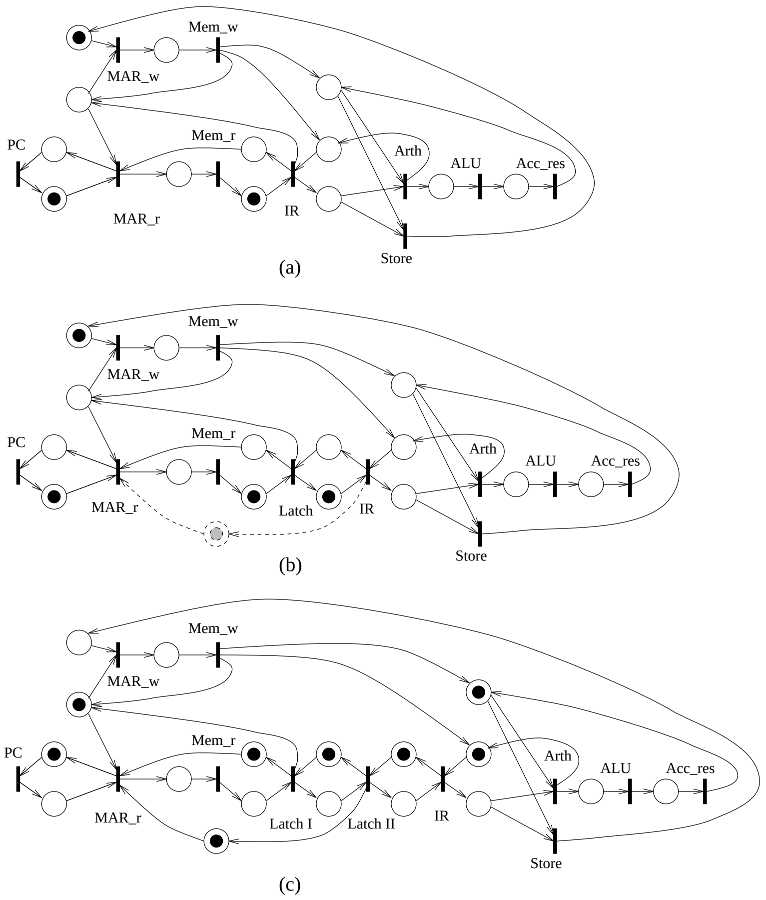

Figure 5: Pipeline models of processor.

containing arithmetic operations is reduced.

5 Introducing pipelining

The design developed in the previous section has a low degree of concurrency. We wish to decouple the modules further. For example, instruction decoding, which may take a relatively long time, could be done concurrently with fetching the next word from memory. In this section, we elaborate the design so as to allow for a higher degree of concurrency between its modules.

In the design described in the previous section, fetching could only happen after the result of instruction decoding was known. If an acknowledgement is sent to MAR to enable the next fetch at an earlier stage, say from IR, mutual exclusion between a pair of requests to MAR cannot be guaranteed. Indeed, the next decoded instruction may be "Store", which may try to access MAR simultaneously with the PC Increment loop. We can resolve this problem by creating an additional place in the net model, which will act as a *semaphore* for the actions involving MAR. Independent requests to MAR will thus have to compete for *one token* in this place, thus resolving the mutual

<table>
<tr><td></td><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr>
<tr><td>1</td><td>PC</td><td>-</td><td></td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td></tr>
<tr><td>2</td><td>MAR_r</td><td></td><td>-</td><td></td><td></td><td></td><td></td><td>||</td><td>||</td><td>||</td><td></td><td></td></tr>
<tr><td>3</td><td>Mem_r</td><td>||</td><td></td><td>-</td><td></td><td></td><td></td><td>||</td><td>||</td><td>||</td><td></td><td></td></tr>
<tr><td>4</td><td>MAR_w</td><td>||</td><td></td><td></td><td>-</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>5</td><td>Mem_w</td><td>||</td><td></td><td></td><td></td><td>-</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>6</td><td>IR</td><td>||</td><td></td><td></td><td></td><td></td><td>-</td><td></td><td>||</td><td>||</td><td></td><td></td></tr>
<tr><td>7</td><td>ID</td><td>||</td><td>||</td><td>||</td><td></td><td></td><td></td><td>-</td><td></td><td></td><td></td><td></td></tr>
<tr><td>8</td><td>ALU</td><td>||</td><td>||</td><td>||</td><td></td><td></td><td>||</td><td></td><td>-</td><td></td><td></td><td></td></tr>
<tr><td>9</td><td>Acc_res</td><td>||</td><td>||</td><td>||</td><td></td><td></td><td>||</td><td></td><td></td><td>-</td><td></td><td></td></tr>
<tr><td>10</td><td>Acc_dta</td><td>||</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>-</td><td></td></tr>
<tr><td>11</td><td>GR</td><td>||</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>-</td></tr>
</table>

(a)

<table>
<tr><td></td><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr>
<tr><td>1</td><td>PC</td><td>-</td><td></td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td><td>||</td></tr>
<tr><td>2</td><td>MAR_r</td><td></td><td>-</td><td></td><td></td><td></td><td>||</td><td>||</td><td>||</td><td>||</td><td></td><td></td></tr>
<tr><td>3</td><td>Mem_r</td><td>||</td><td></td><td>-</td><td></td><td></td><td>||</td><td>||</td><td>||</td><td>||</td><td></td><td></td></tr>
<tr><td>4</td><td>MAR_w</td><td>||</td><td></td><td></td><td>-</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>5</td><td>Mem_w</td><td>||</td><td></td><td></td><td></td><td>-</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>6</td><td>IR</td><td>||</td><td>||</td><td>||</td><td></td><td></td><td>-</td><td></td><td>||</td><td>||</td><td></td><td></td></tr>
<tr><td>7</td><td>ID</td><td>||</td><td>||</td><td>||</td><td></td><td></td><td></td><td>-</td><td></td><td></td><td></td><td></td></tr>
<tr><td>8</td><td>ALU</td><td>||</td><td>||</td><td>||</td><td></td><td></td><td>||</td><td></td><td>-</td><td></td><td></td><td></td></tr>
<tr><td>9</td><td>Acc_res</td><td>||</td><td>||</td><td>||</td><td></td><td></td><td>||</td><td></td><td></td><td>-</td><td></td><td></td></tr>
<tr><td>10</td><td>Acc_dta</td><td>||</td><td>||</td><td>||</td><td></td><td></td><td></td><td></td><td></td><td></td><td>-</td><td></td></tr>
<tr><td>11</td><td>GR</td><td>||</td><td>||</td><td>||</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>-</td></tr>
</table>

(b)

Table 2: Temporal relations between transitions for LPN versions 3 (a) and 4 (b).

exclusion problem. This is illustrated in Figure 5(a), where for the sake of simplicity only the decoding of “Store” and “Arth” is shown.

Unfortunately, adding such a dependency appears to be insufficient. If “Store” has been decoded, and its request loses competition for the mutual exclusion token to the request coming from PC, the LPN will deadlock. The newly fetched word will not be able to advance because it is waiting for IR to be cleared, and at the same time IR will be waiting for the “Store” instruction to complete. This corresponds to the marking in Figure 5(a). Thus, an extra register is required to store the newly fetched word, and allow MAR to accept the request from “Store”. The modified LPN model is shown in Figure 5(b). Yet, this modification is still insufficient for avoiding a deadlock. The processor will stall if the pipeline fills with pre-fetched PC values waiting to be decoded, but IR is occupied by a “Store” instruction. Thus the request from PC should only be allowed to “bid” for access of MAR when there is room in the pipeline. This is introduced in form of an additional dependency constraint, a place shown dashed in Figure 5(b).

Analysis of this LPN shows that it is safe, live and deadlock-free. From an analysis of the temporal relations between the transitions we conclude that instruction decoding is now concurrent with fetching a new word from memory (see the table of relations in Table 2(a)). An additional benefit is that when a two-word instruction is executed, the second word is fetched in parallel with the decoding of the instruction. After decoding is completed, the appropriate register can start to latch the data earlier.

It is still possible to increase concurrency between the modules. Notice that if an additional latch is introduced, which decouples IR and Memory register, latching of data into IR can also be done concurrently with fetching of new words from memory. Analysis of the LPN in Figure 5(c) shows

Op. module	PC	MAR	Mem	IR	ID	Acc	GR	ALU
Time (ns)	14+(n+1)	20	55	20	50	20	20	17

Table 3: Average execution times of modules.

that this is true. Thus we obtain an even more concurrent implementation, which gives us the fourth design version of the processor.

6 Performance estimation

The framework presented in the previous sections demonstrates techniques for designing asynchronous circuits using PNs. In this section, we demonstrate how such designs, expressed in the form of LPNs, can be analysed with respect to their performance. We use the above four versions of the processor design. Note that the technique analyses performance of the design specifications, i.e. *before* they are implemented in real physical elements. Since LPNs have transitions labelled with actions that are associated with the operational modules, we will only need the delays of these modules to estimate the performance of the whole design. As the criterion for the performance we use the length of the *PC firing cycle*.

Delay assumptions. As in [7] we will assume that the processor has a 3-bit word length[1]. A reasonable estimate of the delays associated with asynchronous modules can be taken from the AMULET1 description [15]. These delays are shown in Table 3. Note that the delay associated with latching data in a register (effectively, one stage in a micropipeline) is 20 ns, most of which is used to convert the two-phase control between the stages of the pipe, into the four-phase control of the latches [15].

The delay of the PC incrementor depends on the highest changing bit n. In our example, the PC incrementor can be modelled by eight separate, mutually exclusive transitions (one for each combination of three-bit values) with appropriate associated delays. According to [16], the delay of ALU, in any arithmetic operation, is 17 ns for carry chains whose length is less than 4. Since in our case the word length is 3, we can use this figure. The delay of ID is chosen to be equal to the corresponding figure of AMULET1. Of course, for our simple microprocessor this is a pessimistic assumption.

[1]Additional study can be done to examine performance of AMP with different word lengths.

However, this allows us to illustrate how the pipelining affects the performance. Performance of the processor is estimated while executing a test program. For simplicity, we assume that the instructions "LdAc", "LdGR", "Arth" and "Store" are executed in arbitrary order, but that there are no other instructions involved. On average, this would correspond to the example program of [7]. The processor is assumed to operate in cyclic mode, i.e. after PC has reached the value "111" it resets to "000".

Performance analysis of design versions. To estimate the performance, we used an existing tool for analysis of timed and stochastic PNs – UltraSAN [2]. We also measured the cycle times for different designs executing only one particular instruction. Since "LdAc" and "LdGR" are similar, only one measurement is presented.

In the first design, only PC Increment could happen concurrently with execution of any instruction. Thus the average delay of instruction execution is simply determined as an average of execution times of all instructions.

In version 2, with a decoupled ALU, arithmetic instructions can be executed concurrently with fetching the next word from memory. Observe the reduction of the value in line 4 in Table 4 for the mode when only arithmetic operations are executed. This is the only value affected by the change of order manifested by version 2. The average instruction execution time for a processor with such a small word size is only slightly changed, as can be seen in Table 4.

The remaining two versions are in fact three- and four-stage micropipelines with some extra feedback. Introducing pipelining in version 3 allows concurrent fetching of data from the memory and instruction decoding. Since instruction decoding is included into the execution cycle of each instruction, the average time required for instruction execution is reduced (see Table 4).

The last version has IR decoupled to enable its latching to be done concurrently with instruction fetching. As can be observed, introducing an additional register only slightly affects the PC Increment cycle. This register allows decoupling of the IR, but it also introduces extra latency in the execution of "Store". Therefore, a new PC value has more chances to win arbitration and fill up the pipeline. In addition, a new register has little effect on register loading instructions because in most cases the Memory register latches incoming data before ID has decoded an instruction. Thus the PC cycle time of this version is close to the previous one.

Let us compare the synchronous version of the processor with its asynchronous counterparts. Ex-

No.	Measure	Ver. 1	Ver. 2	Ver. 3	Ver. 4
1	PC cycle	141.8	137.6	112.4	109.0
2	LdAcc	240.6	240.6	200.3	200.0
3	Store	220.5	220.5	175.2	179.3
4	Arth	183.0	145.3	100.3	100.0

Table 4: Performance of different versions of processor (ns).

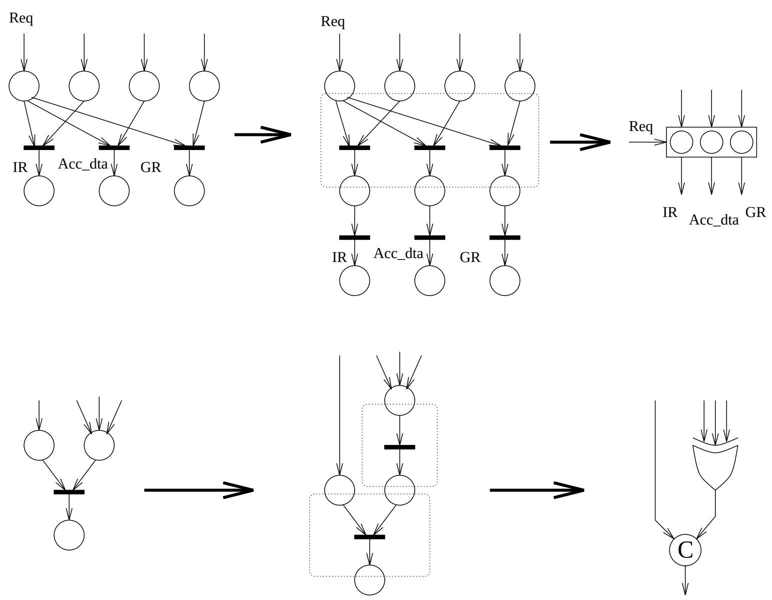

Figure 6: Translations of PN segments into asynchronous circuit elements.

ecution of each instruction in the synchronous version takes two clock cycles (four periods). Usually, the period involving computations in ALU dictates the clock period length for the whole processor. However, when the ALU is small, like in this case, its cycle time is less than that of the Memory. Therefore the period involving Memory operations will take more time. It is reasonable to assume that this period takes up to 55 ns. This value also includes the time needed for address decoding and for latching the data in a register. Thus average instruction execution time is at least 220 ns, which is close to the worst case results obtained for the asynchronous version when it executes one type of instruction. Obviously, the ability to "save up time" while dealing with faster instructions results in a reduction of the average instruction execution time of the asynchronous processor.

7 Hardware synthesis

Net-level transformation. The next step in the design process is the transformation of an LPN with transitions labelled with actions of modules into an LPN which can be translated into a circuit. Each place in the high level LPN is considered to be an input of a module. There are two types of transformation, one to be applied to *places with multiple input arcs* and the other for *synchronising transitions*.

The first one is required because no two circuit modules can have their outputs connected. In this case we need to introduce some control elements for merging the signals. Each place represents a merge operation on its inputs. Since this place is safe, which is dictated by the hazard-freedom condition, the merging operation can be implemented by an XOR element. In terms of the LPN, we introduce an explicit auxiliary transition which separates the merging operation from the inputs of the modules. All inputs will arrive mutually exclusive in time, and each such event is signalled on the output. Sometimes complex XOR elements with more than two inputs may not be available in the element library. We then refine the places with multiple input arcs into a "tree-like" LPN segment so that each place has no more than two input arcs. An event on any of the inputs of such a segment will be forwarded to the output.

The second transformation is required because each module itself is not capable of synchronision of requests. They have only one request input for each operation. Thus all sychronisations need additional control logic. In this case we introduce additional transitions which correspond to extra elements that synchronise the inputs.

Examples of both types of LPN transformations are shown in Figure 6. Each place of the LPN corresponding to an module has only one input arc.

Circuit synthesis. The LPN is now translated into an asynchronous circuit. The method of translation is based upon Patil's work [14]. It uses the close correspondence between the event-driven semantics of a two-phase micropipeline control logic [21, 15] and that of LPNs. During the design process we made sure that all transitions corresponding to one operational module working in different modes are not mutually concurrent. Therefore, all such transitions are translated into *this module together with a corresponding number of inputs and outputs*. Transitions introduced for merging inputs are translated into XOR elements. Synchronising transitions *which are not in conflict* with any other transitions are translated into Muller *C-elements* [21], also called Join elements [14]. By using the persistency relations between transitions we can identify *unique choice* (a structural

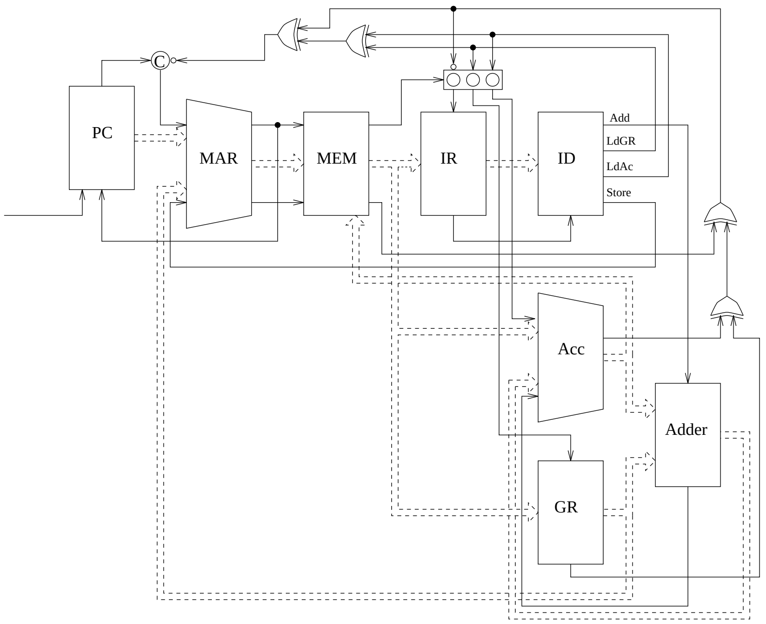

Figure 7: Straightforward implementation of AMP.

conflict with no dynamic conflict). A unique choice structure is translated into a Decision-Wait element [15], which functions as a generalised C-element. The use of a Decision-Wait element, as opposed to a collection of C-elements, is required because there is no guarantee that the phases of the signals being synchronised will be the same. This may happen, for example, when synchronising a request for the latching of a new instruction in IR and an acknowledgement from another module. If a two-word instruction is executed before or after a one-word instruction, one phase of synchronisation is "skipped" for IR and used to activate another register.

All other choice structures between transitions that represent the control logic are translated into Arbitration modules for resolving the conflict. The resulting circuit is shown in Figure 7. Note that Accumulator uses one signal to acknowledge the latching of data both from Memory and ALU. The data path is shown with dashed lines.

Another example is the circuit for the third version, shown in Figure 8. Transitions preceding "MAR_r" and "Mar_w" are in dynamic conflict. In the implementation we translate this structure into an arbiter.

At this point, we arrive at an implementation for each particular design. The performance of each implementation can be estimated more accurately, taking into account the delays of the control logic. Other properties such as area and power consumption may also be estimated. However, these issues are outside the scope of this paper.

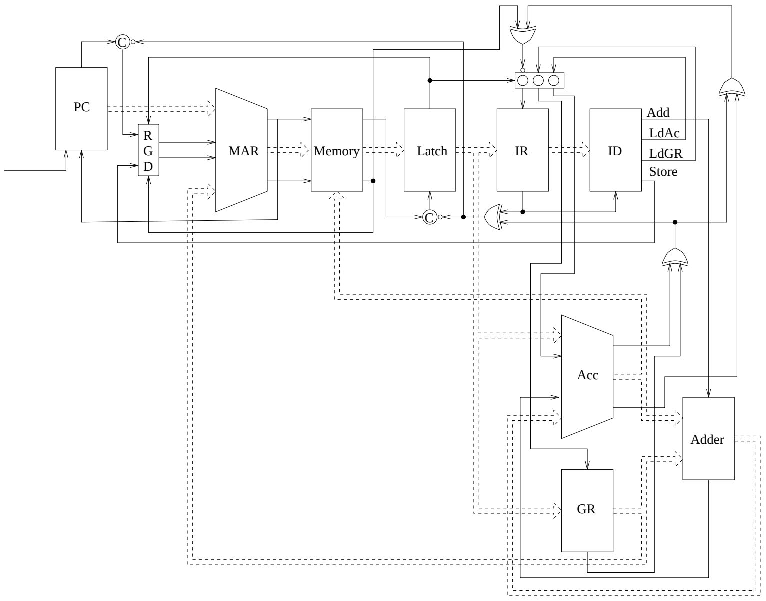

Figure 8: Implementation of decoupled version of processor with arbiter.

8 Conclusions

We have described the design of an asynchronous processor using PNs. We used a simple example whose functionality was described in [7]. Our method leads to an implementation by means of a step-wise refinement of an LPN specification, initially in very abstract terms. It allows analysis of the behaviour specified by the LPN. The design process is assisted by the analysis of the relations between transitions, making this approach even more flexible.

We have shown performance estimates for several versions of the design. This estimation is done at the specification level, well before the circuit implementation stage is reached. This allows the designer to address certain bottlenecks at an earlier design stage, and thus improve the resulting circuit.

We have also suggested a transformation technique for converting the specification LPN into a circuit by means of a mechanical process. The authors plan to continue investigation of this technique.

References

[1] Amulet1 group workshop: Presentation materials. Technical report, Manchester University, Department of Computer Science, Amulet Group, Lake District, England, July 18-22 1994.

[2] UltraSAN, user's manual, version 3.0. Technical report, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-

Champaign, Illinois, USA, 1995.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulations. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[4] T.A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications.* PhD thesis, MIT, 1987.

[5] J.B. Dennis. Modular, asynchronous control structures for a high performance processor. In *Proceedings of Project MAC Conference on Concurrent Systems and Parallel Computation.*, pages 55–92, June 1970.

[6] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V.Woods. A micropipelined ARM. In *VLSI'93, Grenoble, September 1993.* Conference best paper award.

[7] W.C. Holton. The large scale integration of mcroelectronic circuits. *Scientific American*, pages 82–94, 1977.

[8] A.J. Martin. Collected papers on asynchronous vlsi design. Technical Report Caltech-CS-TR-90-09, California Insitute of Technology, 1990.

[9] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

[10] R. Milner. *A Calculus for Communication Systems.* Springer-Verlag, 1980.

[11] T. Murata. Petri nets: Properties, analysis and application. *Proceedings of IEEE*, 77(4):541–574, April 1989.

[12] T. Nanya. A quasi-delay-insensitive microprocessor: Titac-i. In *Proceedings of 1995 Israel Workshop on Asynchronous VLSI*, March 1995.

[13] S. Nicklin. Petri-net modelling of the amulet1 address interface. in [1].

[14] S.S. Patil. Cellular arrays for asynchronous control. In *Proceedings of the ACM 7th Annual Workshop on Microprogramming*, 1974. (CSG Tech. Memo. 122, Project MAC, MIT, April 1975).

[15] N.C. Paver. *The design and implementation of an asynchronous microprocessor.* PhD thesis, University of Manchester, 1994.

[16] N.C. Paver. Private communications, 1995.

[17] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood Cliffs, 1981.

[18] O. Roig, E. Pastor, and J. Cortadella. Symbolic model checking for speed independent circuits. In *Proceedings of ACiD Workshop on Asynchronous Low Power VLSI, Lyngby, Denmark*, April 1993.

[19] L.Ya. Rosenblum and A.V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets, Torino, Italy, July 1985*, pages 199–207. IEEE Computer Society, 1985.

[20] K.S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems.* PhD thesis, The University of Calgary, September 1994.

[21] I.E. Sutherland. Micropipelines. *Communications of ACM*, 32(6):720–738, 1989.

[22] A. Valmari. Stubborn Sets for Reduced State Space Generation. *Advances in Petri Nets 1990, ed.G.Rozenberg, LNCS 483*, pages 491–515, 1991.

[23] A.V. Yakovlev, A.M. Koelmans, A. Semenov, and D.J. Kinniment. Modelling, analysis and synthesis of asynchronous control circuits using petri nets. Technical Report 514, University of Newcastle upon Tyne, 1995.