Designing an Asynchronous Processor Using Petri Nets

Alex Semenov

Albert M. Koelmans

Lee Lloyd

Alexandre Yakovlev

University of Newcastle upon Tyne

Using a simple example, we demonstrate how to design and analyze asynchronous systems from labeled Petri net specifications, later refining, transforming, and translating them for implementations.

Petri nets are becoming increasingly attractive as a formal model for hardware system design. The graphical nature of the Petri net notation makes it more attractive to circuit designers than algebraic notations, which are much less intuitive.

The mathematically well-founded Petri nets are useful in exposing potential hazards in circuits. Designers can use them as a modeling language to perform formal synthesis and high-level analysis of complex processor designs and signal processing chips. They can translate the Petri nets to VHDL, and vice versa for subsets of VHDL, making it possible to integrate Petri net tools into existing design environments.

Many researchers have proposed extensions to the Petri net notation for the accurate modeling of circuit properties such as timing information. Because the event-driven nature of Petri nets mirrors the event-driven nature of asynchronous circuits, many asynchronous circuit designers use Petri nets and their closely related notation, the signal transition graph.[1]

Several design groups have completed asynchronous processor designs. Among the most recent are the Amulet1 from Manchester University,[2] an asynchronous microprocessor from the California Institute of Technology in Pasadena,[3] the Mayfly microprocessor from Hewlett-Packard Labs,[4] and the TITAC from Tokyo Institute of Technology.[5]

Each of these design groups used their own notation during the design process. For example, the microprocessor designed by Alain Martin's group at Caltech used a language called Communicating Hardware Processes. The group at HP Labs used algebraic models to verify the specifications and implementations of finite-state machines for the Mayfly distributed memory microprocessor. On the other hand, the Amulet group, led by Steve Furber, designed its first microprocessor virtually without using formal methods.

Attempts to model and analyze processors formally occurred as early as 1970.[6] In this work, Dennis describes the modeling of the CDC 6600 CPU using Petri nets. However, this and later examples of such work aimed at modeling existing (asynchronous) circuits, rather than designing new circuits from their initial specifications.

To our knowledge, the use of Petri nets and their related formalisms in actual synthesis of hardware has been scarce in the literature. The best known formal model, the signal transition graph, typically supports the synthesis of asynchronous interface circuits. However, these graphs are low-level models unsuitable for the 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 methods based on signal transition graphs, the complete design of an entire processor is a considerably more difficult task. We feel that the best way to approach large-circuit design with Petri nets would be to 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 (Advanced RISC Machine) did for the Amulet group. We decided upon Holton's[7] simple processor design, which demonstrates the fundamentals of processor operation, is clear, and is easy to understand. Then, we organized our asynchronous processor using the same operational modules and the same instruction set as Holton's processor.

Our design of a simple asynchronous processor is scalable and can be developed further into a fully operational version. Our aim was not to develop a complete hardware device, but to demonstrate design methods that use Petri nets and their modeling power. We also wanted to show how Petri net analysis tools can assist designers, so we point out particulars of circuit behavior here. In addition, the processor can serve as an ideal testbed for analyzing different properties, such as timing.

Petri nets and their analysis

The following briefly introduces Petri net theory. For a more comprehensive introduction, refer to Murata.[8]

Definitions. A marked Petri net is tuple $N = (P, T, F, m_0)$ in which P and T are nonempty sets of places and transitions, F is a flow relation that connects places to transitions and transitions to places. That is, $F \subseteq (P \times T) \cup (T \times P)$, and m_0 is the initial marking. We represent a Petri net as a graph with two types of nodes: Circles denote places, while bars or boxes indicate transitions. Tokens (bold dots) depict net markings.

A transition is enabled under a given marking when every input place contains 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 net's reachability set. A labeled net is a Petri net N along with its labeling function $L : T \rightarrow A$, which labels each transition with an action name from alphabet A.

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

A reachable marking m at which no transition is enabled is a deadlock. A Petri net is free of deadlocks if its reachability set includes no deadlocks. In a system that operates in cycles, the presence of deadlocks is regarded as an error.

A transition t of a Petri net is live if, for any reachable marking m, there exists a marking m' reachable from m at which this transition is enabled. A Petri net is live if every transition is live. This is often called a strong form of Petri net 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 the 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 that is not live usually indicates that some operation of the designed system can never be performed.

A marked Petri net is 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 a marking at which t' can disable t exists, then t and t' are in dynamic conflict. Clearly, to be in dynamic conflict, transitions t and t' must share at least one input place. This sharing is called a structural conflict. A Petri net is persistent if it is persistent for 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 Petri net properties. For example, if one transition may be disabled by another, 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 Petri net are concurrent if a marking m exists 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.

Analysis. There are several methods for analyzing Petri net dynamic behavior. One builds a reachability set that 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 Petri net.

Researchers have suggested several methods to overcome the state space explosion. Among them are Petri net symbolic traversal,[9] stubborn set methods,[10] and Petri net unfoldings.[11] Petri net symbolic traversal uses implicit representation of the reachability set in the form of binary decision diagrams. BDDs are canonical representations of Boolean functions in graphical form. Petri net symbolic traversal efficiently analyzes 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 deadlock freedom usually involves exploring other states.

Petri net unfolding represents the full reachability graph using partial orders that preserve relations between transition occurrences. (A transition occurrence is a unique event associated with a single act of firing the transition.) Since all reachable markings are represented in the Petri net unfolding, we can easily obtain the concurrency relation for two transitions. While discussing the design steps in the next section, we will refer to the analysis techniques we used in checking the behavioral correctness of the microprocessor.

Unlike ordinary (untimed) Petri nets, 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 transition firing times, not their sum as in the case of sequential operation.

A more efficient design is one in which a certain major module is decoupled from the rest of the circuit. In the following, we introduce a pipelined operation in the system as an example of time efficiency. A Petri net description easily captures such an operation.

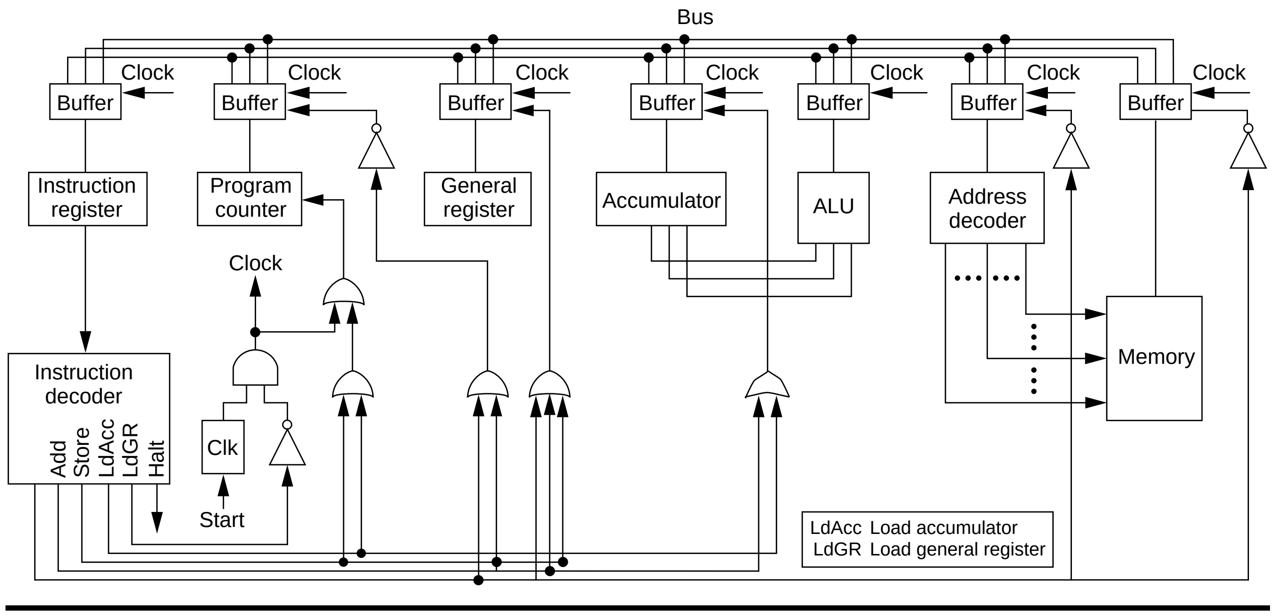


Figure 1. Synchronous implementation of a processor.

Synchronous implementation

Holton[7] describes a simple 3-bit processor design. Figure 1 reproduces its architectural organization. This synchronous design uses a common clock to synchronize data transfers between processor modules. It consists of the major operational modules: instruction register (IR), instruction decoder (ID), program counter (PC), general register (GR), accumulator (Acc), arithmetic and logic unit (ALU), address decoder (AD), and memory (Mem).

All modules connect to one shared bus through buffers. The instruction decoder serves as a processor manager by configuring the processor for execution of the current instruction.

The processor's operational cycle 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 program counter is incremented during the first clock period, and the new value of the program counter is presented to memory. During the second clock period, a word fetched from memory is latched in the instruction register. 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.

The instruction decoder determines which modules should be connected to the bus. If an arithmetic instruction is fetched, this decoder connects the ALU and the appropriate registers. If the instruction loads one of the registers, the instruction decoder connects the appropriate register and memory to the bus and signals memory to produce the data kept at the address decoded by the address decoder. A store instruction causes the general register to be connected to the bus together with the address decoder to load the address into memory; then the accumulator and memory are connected to write the data kept in the accumulator.

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 to drive the clock signal is wasted on those modules not involved in executing the current step. The delay of the longest execution cycle determines the clock period. 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 clock skew problem is increasingly becoming a major issue in chip designs with high clock rates. Asynchronous circuits do not have clocks, and thus avoid these problems.

By using Petri nets, we aim to ensure that the final design is functionally correct.

An asynchronous version

We follow the top-down design methodology in which, after deciding upon the top-level specification, we refine the specification until we reach the implementable level.

Basic design. To obtain a comparable asynchronous version of the processor, we use asynchronous equivalents of the modules used in the synchronous version. The main objective of the first design stage is to produce a labeled Petri net that has transitions labeled only with actions of the corresponding modules. During the second stage, we transform this high-level labeled Petri net into one that contains explicit transitions of control elements, and can therefore be translated into a circuit. We restrict ourselves to the instruction set specified in Holton,[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.

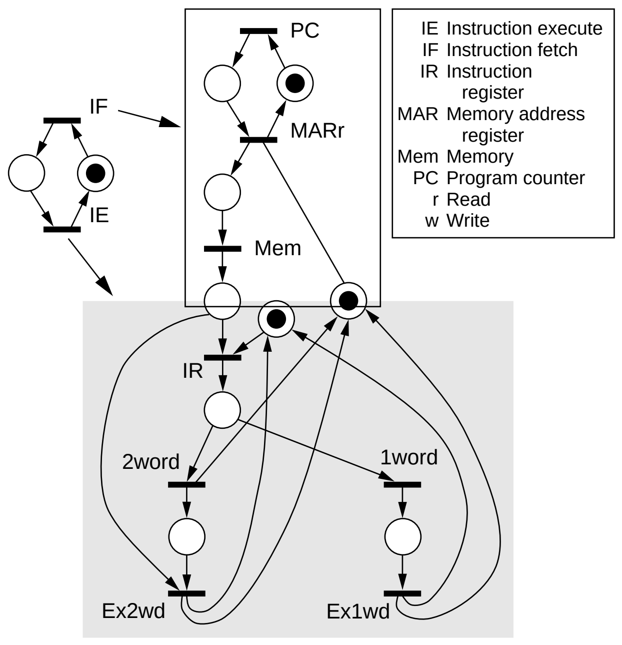


Figure 2. High-level specification of the asynchronous microprocessor (AMP). Circles indicate places; bars or boxes indicate transitions; bold dots depict net markings.

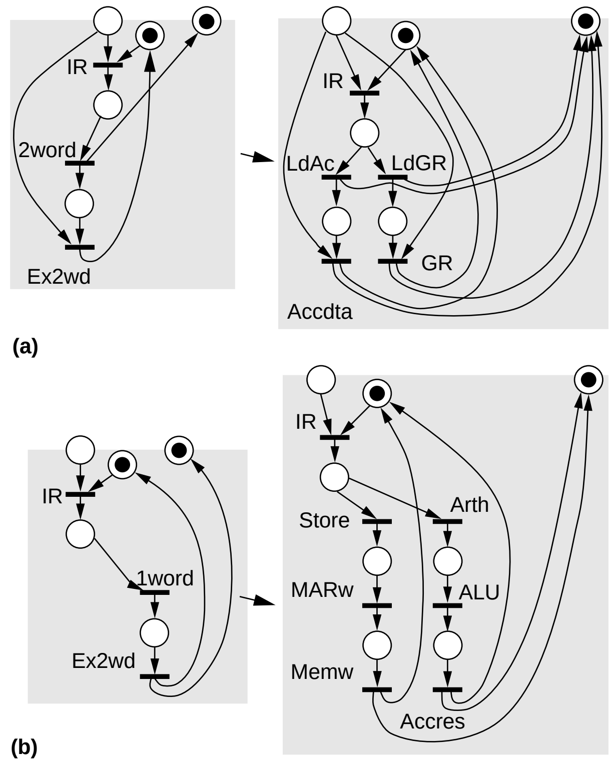


Figure 3. Model refinements for two-word (a) and one-word (b) instructions.

This follows the most abstract specification of the processor's operation: It alternates between the instruction fetch and instruction execute modes. Thus, the initial specification is simply a labeled Petri net with two transitions representing both modes.

Action refinement. We now refine these two transitions. First, we change the instruction fetch transition 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 memory at the address specified by MAR.

We assume that memory does not have output latching. It accepts an address along with the accompanying request-for-read signal, and produces an acknowledgment 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 memory module.

Memory uses another set of inputs for a write operation. Whenever a write request arrives, the processor stores data from the write bus at the location specified by MAR. This is acknowledged on a separate wire. The MAR and memory modules can therefore be activated in two modes: instruction fetch and instruction execute. To avoid confusion and indicate the mode in which they operate, we label the memory module operation with an "r" for read and a "w" for write.

Instruction execute needs careful consideration. The instruction set has two types of instructions: one-word (1wd) and two-word (2wd). 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 the instruction register using the address in the program counter. If a two-word instruction (such as load accumulator) executes, the next word fetched from memory contains data. Though the instruction word remains in the instruction register, the processor sends an acknowledgment 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.

When an instruction is latched in the instruction register, the instruction decoder decodes and executes it. While the instruction executes, the contents of the instruction register must not change. The load accumulator instruction is refined into LdAcc, representing the decoding of the instruction, and Accdta, which represents the actual latching of the second word in the register. Instruction Arth is decomposed into ALU and Accres, which corresponds to the activation of the ALU and latching of the result in the accumulator. Store is refined into MARw, which loads MAR with an address at which the data from the general register is to be stored, and Memw, which represents storing of the data in memory. Figure 3 shows these refinements.

Transitions labeled with Accdta and Accres correspond to the accumulator being used in two modes: register loading 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.

Table 1. Temporal relations between transitions (Tr.) for one labeled Petri net, first processor version.

No.	Tr. name	1	2	3	4	5	6	7	8	9	10	11
1	PC	-		||	||	||	||	||	||	||	||	||
2	MARr		-									
3	Memr	||		-								
4	MARw	||			-							
5	Memw	||				-						
6	IR	||					-					
7	ID	||						-				
8	ALU	||							-			
9	Accres	||								-		
10	Accdta	||									-	
11	GR	||										-

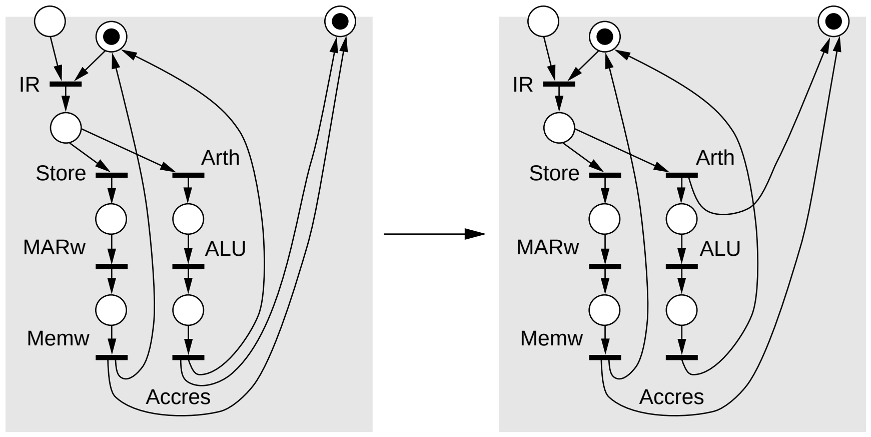


Figure 4. Labeled Petri net refinement with decoupled ALU action.

Table 2. Temporal relations between transitions for processor version 2.

No.	Tr. name	1	2	3	4	5	6	7	8	9	10	11
1	PC	-		||	||	||	||	||	||	||	||	||
2	MARr		-						||	||		
3	Memr	||		-					||	||		
4	MARw	||			-							
5	Memw	||				-						
6	IR	||					-					
7	ID	||						-				
8	ALU	||	||	||					-			
9	Accres	||	||	||						-		
10	Accdta	||									-	
11	GR	||										-

Analysis and improvement. We now have a labeled Petri net that contains only transitions labeled with actions of modules. Verification of this labeled Petri net using the characteristic segment of its unfolding shows that the labeled Petri net is live, safe, and free of deadlocks. Reported non-persistent transitions are those representing a data-dependent choice of the type of instruction in the instruction decoder.

We now derive temporal relations between the transitions of the labeled Petri net. Table 1 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 a program counter 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 labeled Petri net therefore represents a behavior that can be implemented as an asynchronous circuit, and its functionality meets the design specification. In contrast with the synchronous version, the delay in executing any particular instruction depends only on the actual speed of the modules.

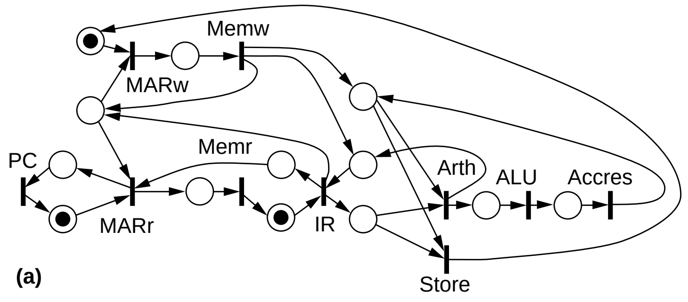
Analysis of the relations between the transitions in this design shows that the program counter increment is the only operation concurrent with the execution of the current instruction. However, any arithmetic instruction can be executed concurrently with fetching the next word from memory. The instruction does not require data from memory. Once the Arth instruction is latched in the instruction register and decoded in the instruction decoder, an acknowledgment to proceed can be sent to MAR. Completion of the instruction is acknowledged to the instruction register to allow Arth to complete. This observation results in a different labeled Petri net refinement, which is shown in Figure 4.

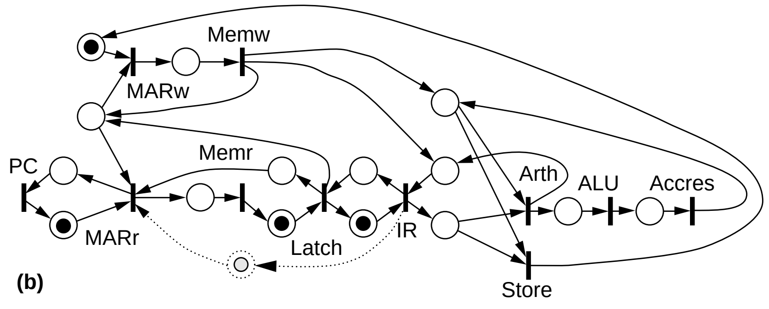
Behavioral analysis of this labeled Petri net shows that it holds the same properties as the initial labeled Petri net. Analysis of the relations between transitions (Table 2) reveals that the execution of an arithmetical operation, including writing to the accumulator, may happen concurrently with loading MAR with a new address and reading the next word from memory. This reduces the average execution time of a program containing arithmetic operations.

Pipelining

Because the second processor design has a low degree of concurrency, we need to decouple the modules further. For example, instruction decoding, which may take a relatively long time, could proceed concurrently with fetching the next word from memory. We now elaborate on the design to allow a higher degree of concurrency between its modules.

The previous design could only allow fetching after the result of instruction decoding was known. If an acknowledgment is sent to MAR to enable the next fetch at an earlier stage, say from the instruction register, 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 program counter's 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 then have to compete for one token in this place, thus resolving the mutual exclusion problem. Figure 5a illustrates this, showing only the decoding of store and Arth for the sake of simplicity.





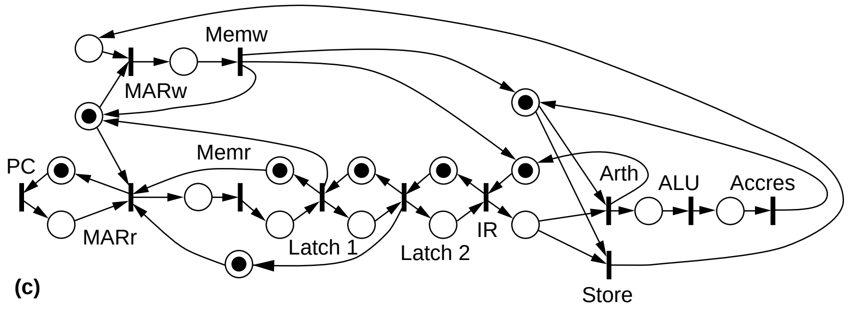


Figure 5. Pipelined processor models: deadlocking (a), with one latch (b), and with two latches (c).

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 the program counter, the labeled Petri net will deadlock. The newly fetched word will not be able to advance because it is waiting for the instruction register to be cleared, and at the same time the instruction register will be waiting for store to complete. This corresponds to the marking in Figure 5a.

Now, we need an extra register to store the newly fetched word and allow MAR to accept the request from store. Figure 5b shows the modified labeled Petri net model.

Yet, this modification is still insufficient for avoiding a deadlock. The processor will stall if the pipeline fills with prefetched program counter values waiting to be decoded; however, a store instruction occupies the instruction register. The request from the program counter should only be allowed to bid for access to MAR when there is room in the pipeline. This is introduced in the form of an additional dependency constraint, a place, shown as a dashed line in Figure 5b.

Analysis of this labeled Petri net 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 lists of relations in Tables 3 and 4, next page). An additional benefit is that when a two-word instruction executes, the second word is fetched in parallel with the instruction decoding. After decoding completes, the appropriate register can start to latch the data earlier.

It is still possible to increase concurrency between the modules. Note that we introduce an additional latch, which decouples the instruction and memory registers. Data latching into the instruction register can also occur concurrently with the fetching of new words from memory. Analysis of the labeled Petri net in Figure 5c shows that this is true. We obtain an even more concurrent implementation, which gives us the fourth design version of the processor. See Table 4.

Performance estimation

The framework we've presented demonstrates techniques for designing asynchronous circuits using Petri nets. Now we can use the four versions of the processor design to demonstrate how we analyze the performance of designs expressed in the form of labeled Petri nets. The technique analyzes design specification performance, that is, before specifications are implemented in actual physical elements. Since labeled Petri nets have transitions labeled with actions

Table 3. Temporal relations between transitions for processor version 3.

No.	Tr. name	1	2	3	4	5	6	7	8	9	10	11
1	PC	-		||	||	||	||	||	||	||	||	||
2	MARr		-					||	||	||		
3	Memr	||		-				||	||	||		
4	MARw	||			-							
5	Memw	||				-						
6	IR	||					-		||	||		
7	ID	||	||	||				-				
8	ALU	||	||	||			||		-			
9	Accres	||	||	||			||			-		
10	Accdta	||									-	
11	GR	||										-

Table 4. Temporal relations between transitions for processor version 4.

No.	Tr. name	1	2	3	4	5	6	7	8	9	10	11
1	PC	-		||	||	||	||	||	||	||	||	||
2	MARr		-				||	||	||	||		
3	Memr	||		-			||	||	||	||		
4	MARw	||			-							
5	Memw	||				-						
6	IR	||	||	||			-		||	||		
7	ID	||	||	||				-				
8	ALU	||	||	||			||		-			
9	Accres	||	||	||			||			-		
10	Accdta	||	||	||							-	
11	GR	||	||	||								-

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 program counter firing circle.

Delay assumptions. As in Holton,[7] we assume that the processor has a 3-bit word length. (Additional study can be done to examine AMP performance with different word lengths.) We get a reasonable estimate of the delays associated with asynchronous modules from the data obtained from the Amulet group. Table 5 lists these delays.

The delay associated with latching data in a register (effectively, one stage in a micropipeline) is 20 ns. Most of this is used to convert the two-phase control between the pipeline stages into the four-phase control of the latches.

The delay of the program counter incrementor depends on the highest changing bit *n*. In our example, the incrementor can be modeled by eight separate, mutually exclusive transitions (one for each combination of 3-bit values) with appropriate associated delays. According to Amulet1 data, the ALU delay in any arithmetic operation is 17 ns for carry chains of less than 4 bits. Since in our case the word length is 3, we can use this figure.

We chose the delay of the instruction decoder to be equal to the corresponding figure of Amulet1. Of course, for our simple microprocessor this is a pessimistic assumption. However, it allows us to illustrate how pipelining affects performance. We estimated processor performance while executing a test program (line 1 in Table 6). For simplicity, we assumed that instructions LdAc, LdGR, Arth, and store execute in arbitrary order and no other instructions are involved. On average, this would correspond to Holton's example program. We assumed the processor would operate in cyclic mode, that is, after the program counter reaches 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 Petri nets—UltraSAN from the University of Illinois at Urbana-Champaign.[13] We also measured the cycle times for different designs executing only one particular instruction (lines 2 to 4 in Table 6). Since LdAc and LdGR are similar, we present only one measurement.

In the first design, only the program counter increment could happen concurrently with any instruction's execution. The average delay of instruction execution is simply an average of the execution times of all instructions.

In version 2, with a decoupled ALU, arithmetic instruc-

Table 5. Average module execution times.

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

Table 6. Performance of different processor versions (ns).

Measure	Ver. 1	Ver. 2	Ver. 3	Ver. 4
PC cycle	141.8	137.6	112.4	109.0
LdAcc	240.6	240.6	200.3	200.0
Store	220.5	220.5	175.2	179.3
Arth	183.0	145.3	100.3	100.0

tions can execute concurrently with fetching the next word from memory. Observe the reduction of the value in line 4 in Table 6 for the mode when only arithmetic operations execute. 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 the table shows.

The remaining two versions are in fact three- and four-stage micropipelines with some extra feedback. Introducing pipelining in version 3 allows concurrent data fetching from the memory and instruction decoding. Since instruction decoding is included in the execution cycle of each instruction, the average time required for instruction execution is reduced (see Table 6).

The last version has the instruction register decoupled to enable its latching to take place concurrently with instruction fetching. As can be observed, introducing an additional register only slightly affects the program counter increment cycle. This register allows decoupling of the instruction register, but it also introduces extra latency in the execution of store. Therefore, a new program counter 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 the instruction decoder has decoded an instruction. The program counter cycle time of this version is close to the previous one.

Let us compare the synchronous version of the processor with its asynchronous counterparts. Execution of each instruction in the synchronous version takes two clock cycles (four periods). Usually, the period involving computations in the ALU dictates the clock period length for the whole processor. However, when the ALU is small, as 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. This brings the average instruction execution time to 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 time while dealing with faster instructions results in a reduction of the average instruction execution time of the asynchronous processor.

Hardware synthesis

The process of obtaining the implementation for the processor's model consists of first transforming the specification and then translating it into the circuit implementation.

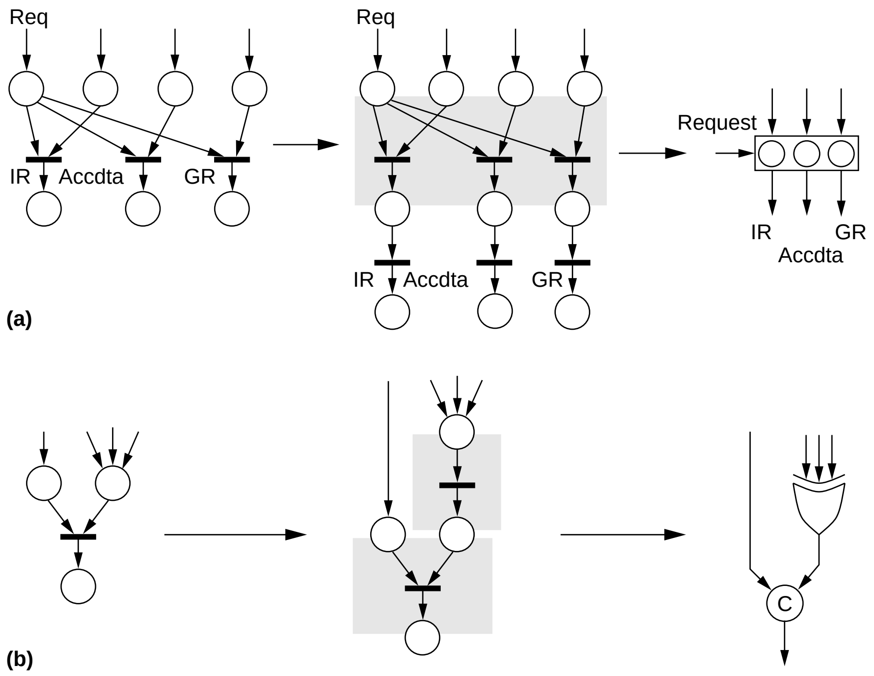


Figure 6. Translations of Petri net segments into asynchronous circuit elements: 3×1 decision-wait (a) and XOR and C, a Muller C (b).

Net-level transformation. In this step we transform a labeled Petri net with transitions labeled with module actions into a labeled Petri net for translation into a circuit. Each place in the high-level labeled Petri net is considered to be an input of a module. There are two transformation types, one to be applied to places with multiple input arcs and the other for synchronizing transitions.

The first type 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, as dictated by the hazard-free condition, an XOR element can implement the merging operation.

In terms of the labeled Petri net, we introduce an explicit auxiliary transition that separates the merging operation from the modules' inputs. All inputs will arrive mutually exclusive in time, and the output will signal each such event. 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 treelike segment of a labeled Petri net. Then, 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 cannot synchronize requests. Modules have only one request input for each operation. As a result, all synchronizations need additional control logic. In this case, we introduce additional transitions that correspond to extra elements that synchronize the inputs.

Figure 6 shows examples of both types of labeled Petri

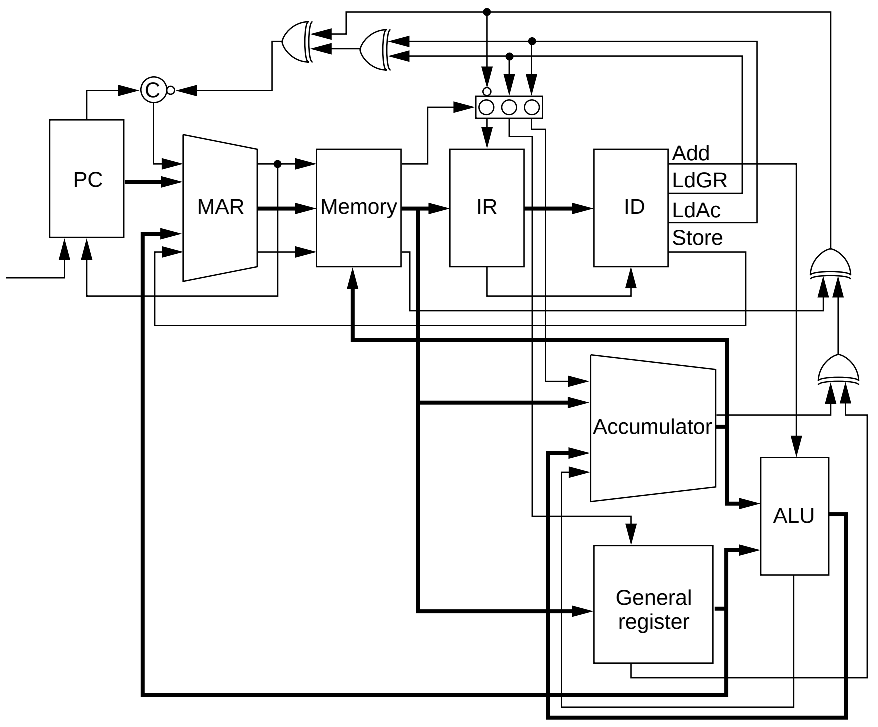


Figure 7. Straightforward AMP implementation. (C indicates the Muller C element.)

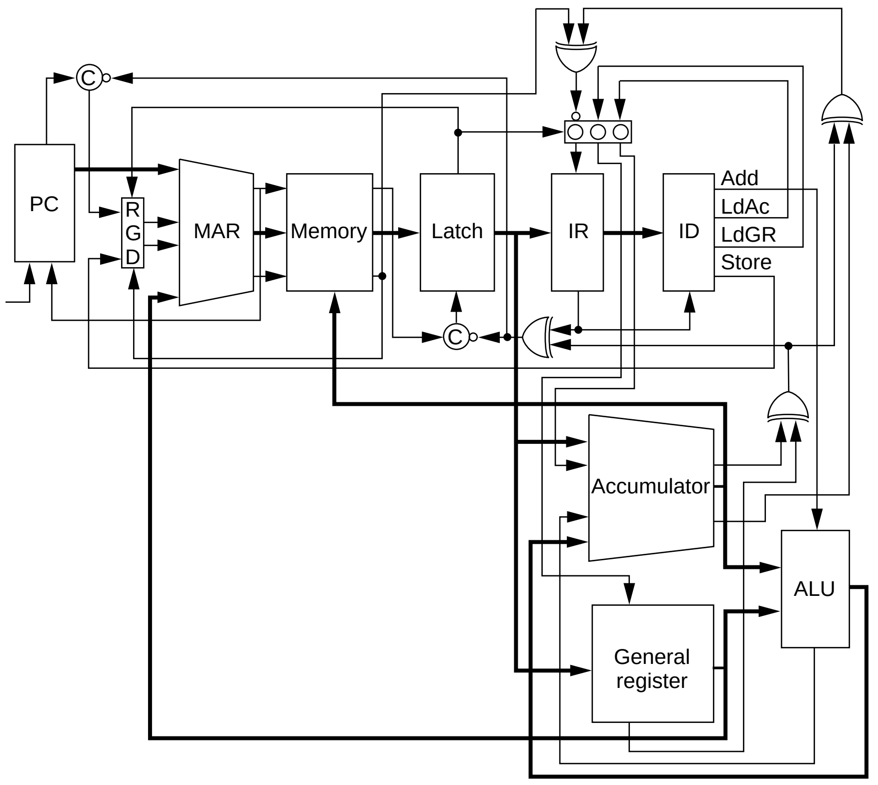


Figure 8. Implementation of decoupled versions of a processor with an arbiter.

net transformations. Each place of the labeled Petri net corresponding to a module has only one input arc.

Circuit synthesis. The labeled Petri net is now translated into an asynchronous circuit with a translation method based on Patil's work.[12] It uses the close correspondence between the event-driven semantics of a two-phase micropipeline control logic[13] and that of labeled Petri nets.

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 translate into XOR elements. Synchronizing transitions that are not in conflict with any other transitions translate into Muller C elements,[12] also called join elements.[10] By using the persistency relations between transitions, we can identify a unique choice (as structural conflict with no dynamic conflict). A unique-choice structure translates into a decision-wait element, which functions as a generalized C element. The use of a decision-wait element, as opposed to a collection of C elements, is necessary because there is no guarantee that the signal phases being synchronized will be the same. This may happen, for example, when synchronizing a request for latching a new instruction in the instruction register and an acknowledgment from another module. If a two-word instruction executes before or after a one-word instruction, one phase of synchronization is skipped for the instruction register and used to activate another register.

All other choice structures between transitions that represent the control logic translate into arbitration modules for resolving the conflict. Figure 7 shows the resulting circuit. Note that the accumulator uses one signal to acknowledge the latching of data from memory and the ALU. The heavy lines in the figure show the data path.

Another example is the circuit for

the third version, as shown in Figure 8. Transitions preceding MARr and MARw are in dynamic conflict; in the implementation we translate this structure into an RGD arbiter.[14]

At this point, we arrive at an implementation for each particular design. We can more accurately estimate the performance of each implementation by taking into account the control logic delays. We can also estimate other properties such as area and power consumption. However, these issues are outside the scope of this article.

OUR METHOD OF DESIGNING an asynchronous processor using Petri nets leads to an implementation by refining a labeled Petri net specification, initially in very abstract terms. It allows analysis of the behavior specified by the labeled Petri net and the relations between transitions, which makes this approach even more flexible.

Performance estimates take place at the specification level, well before reaching the circuit implementation stage. This allows designers to address certain bottlenecks at an earlier design stage, and thus improve the resulting circuit.

We plan to continue investigation of our transformation technique for converting the specification of the labeled Petri net into a circuit by means of a mechanical process.

Acknowledgments

Grants from the Research Committee of the University of Newcastle upon Tyne and CVCP and EPSRC grant GR/J52327 partially supported our work.

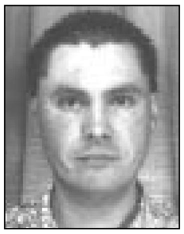
References

1. L.Y. Rosenblum and A.V. Yakovlev, "Signal Graphs: From Self-Timed to Timed Ones," *Proc. Int'l Workshop Timed Petri Nets,* IEEE Computer Society Press, Los Alamitos, Calif., 1985, pp. 199-207.
2. S.B. Furber et al., "A Micropipelined ARM," *Proc. VLSI 93,* North Holland, Amsterdam, 1993, pp. 4.4.1-5.4.10.
3. A.J. Martin, "Collected Papers on Asynchronous VLSI Design," Tech. Report CS-TR-90-09, Calif. Inst. of Technology, Pasadena, 1990.
4. A.L. Davis, "Mayfly: A General-Purpose, Scaleable, Parallel Processing Architecture," *Lisp and Symbolic Computation,* Vol. 5(1/2), May 1992, pp. 7-47.
5. T. Nanya et al., "TITAC: Design of a Quasi-Delay-Insensitive Microprocessor," *IEEE Design & Test of Computers,* Vol. 11, No. 2, Summer 1994, pp. 50-63.
6. J.B. Dennis, "Modular, Asynchronous Control Structures for a High-Performance Processor," *Proc. Project ACM Conf. Concurrent Systems and Parallel Computation,* ACM, New York, June 1970, pp. 55-92.
7. W.C. Holton, "The Large-Scale Integration of Microelectronic Circuits," *Scientific American,* 1977, pp. 82-94.
8. T. Murata, "Petri Nets: Properties, Analysis and Application," *Proc. IEEE,* Vol. 77, No. 4, Apr. 1989, pp. 541-574.
9. E. Pastor et al., "Petri Net Analysis Using Boolean Manipulation," *Proc. Int'l Conf. Applications and Theory of Petri Nets,* LNCS 815, Springer-Verlag, Berlin, 1994, pp. 416-435.
10. A. Valmari, "Stubborn Sets for Reduced State Space Generation," *Advances in Petri Nets 1990,* Lecture Notes in Computer Science 483, G. Rozenberg, ed., Springer-Verlag, Berlin, 1991, pp. 491-515.
11. K.L. McMillan, *Symbolic Model Checking,* Kluwer Academic Publishers, Boston, 1993.
12. S.S. Patil, "Cellular Arrays for Asynchronous Control," *Proc. ACM Seventh Ann. Workshop Microprogramming,* 1974, pp. 178-185; also CSG Tech. Memo 122, Project MAC, Mass. Inst. of Technology, Cambridge, Apr. 1975.
13. *Ultra San User's Manual,* Center for Reliable and High-Performance Computing, Univ. of Illinois at Urbana-Champaign, 1994.
13. I.E. Sutherland, "Micropipelines," *Comm. ACM,* Vol. 32, No. 6, 1989, pp. 720-738.

Alex Semenov is a PhD student in the Department of Computing Science at the University of Newcastle upon Tyne in the UK. His research interests include modeling, formal verification, and synthesis of asynchronous circuits and systems.

Semenov received his MSc from the St. Petersburg State Electrical Engineering University in Russia.

Albert M. Koelmans lectures for the University of Newcastle's Department of Computing Science. His research interests include computer-aided design for VLSI, functional programming applications, and design of asynchronous and fault-tolerant VLSI systems. Koelmans graduated from Groningen University in the Netherlands with a *Doktoraal* degree in mathematics and computer science.



Lee Lloyd is a doctoral student in the same university department. His research interests include design of asynchronous VLSI systems and EMC and low-power design issues. Lloyd obtained his MSc at the University of Newcastle.

Alexandre Yakovlev is a lecturer in the Department of Computing Science, University of Newcastle upon Tyne. His research interests include the field of modeling and synthesis of asynchronous, concurrent, and fault-tolerant systems.

Yakovlev received MSc and PhD degrees in computing science from the St. Petersburg Electrical Engineering Institute, Russia.

Direct comments about this article to Alex Semenov, Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK; alex.semenov@newcastle.ac.uk.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 168 Medium 169 High 170