DESIGN AND VALIDITY ANALYSIS OF ASYNCHRONOUS PROCESSORS USING HIGHER-ORDER PETRI NETS

MUSTSFA SAMI MAHMOUD
Professor of Computer Science
Helwan University

ALI MOHAMMED MELIGY
Associate Professor of Computer Science
Menoufya University

AZEZ SHAFIK
Lecturer of Mathematics
Menoufya University

MUSAAD WAGEH HASSAN
Ass. Lecturer of Computer Science
Tanta University

Abstract

Many researches look for new processor approaches to work with high-performance, battery-powered devices, such as cellular phones. In this paper we present an approach for designing an asynchronous processor using higher-order Petri nets (HOPN). HOPNs form a new class of Petri nets that exploits the properties of higher-order neural networks. The concepts of higher-order arcs will be applied in order to model accurate circuit properties, such as timing information, connection, and concurrency. We consider HOPNs with transitions that have indegree ≤ 1. As a prototype for our design approach we use Holton's Processor. We follow the top-down design, in which we refine the top-level specification until we reach to the implementable level. For analyzing the HOPN, a theorem on the relationship between the potential firability of the goal transition and T-invariant is proved. The circuit synthesis corresponding to the HOPN is discussed. We use the properties of the HOPNs to check the validity of our design.

Index words: *Petri nets (PN), higher-order Petri nets (HOPN), Processor design, Holton's processor model, Validity analysis.*

1. Introduction

Many ideas have been proposed by various researchers to make future microprocessors run at higher frequencies than current ones. New approaches are needed to deal with associated problems including heat elimination and power consumption. One promising approach is the asynchronous design, where no global circuit clock exists. This approach has implications for low power circuits such as those needed for cellular phones, PDAs, and other high-performance, battery-powered devices [GEE05]

Petri nets have been widely used for the modeling and analysis of concurrent systems [Rei85]. There are many factors that contribute to their success: The graphical nature, the ability to model parallel and distributed processes in natural manner, the simplicity of the model, and the firm mathematical foundations. Petri nets are also used in the real-time systems and logic applications. The firm mathematical foundations make Petri nets useful in expressing potential hazards in circuits. Also Petri nets can be used as a modeling language to perform formal synthesis and high-level analysis of complex processor, and signal processing chips design. Petri nets can translate into VHDL, and can be integrated into existing design environments [Sem97].

In some cases, the classical Petri nets can not model the behavior of the system accurately. To solve this problem, researchers proposed many extensions to the basic Petri nets. Examples are: Timed Petri nets [Van89] and [Ajm84]; colored Petri nets [Gen81] and [Van89].

In the same sense [Cho97] used the similarities between neural networks and Petri nets to exploit the properties of higher-order neural networks in a new class of Petri nets called Higher Order Petri Nets (HOPN). He uses the heuristic introduction of higher-order synaptic weights in neural networks to extend the concept of an arc in PN to a general higher-order arc. This extension makes the new class of Petri net more accurate in modeling asynchronous circuits properties. In section 2 the concepts and the structures of higher-order Petri nets are briefly reviewed.

The design of an asynchronous processor takes place in most researchers' work. Several design groups introduce asynchronous processor designs. Among them, Amulet1 from Manchester university [Fur93], an asynchronous microprocessor from the California Institute of Technology in Pasadena [Mar 90], the Mayfly microprocessor from Hewlett-

Packard Labs [Dav 92], and the TITAC from Tokyo Institute of Technology [Tak 94].

Each of these design groups used their own notation during the design process. For example, the design produced by [Fur93] represented virtually without using formal methods. The Amulet1 is an implementation or the ARM processor architecture using the micropipeline design style. The microprocessor designed by [Mar90] used a language called Communication Hardware processor. [Dav92] used algebraic models to verify the specifications and implementations of finite state machines for the Mayfly distributed memory microprocessor. The TITAC design based on quasi-delay-insensitive is a hardware chip designed and implemented as a CMOS gate array.

There was also early attempts to model and analyze processors formally. [Sem97] produce the asynchronous design of Holton's processor, which demonstrates the fundamentals of processor operation. This design is scalable and can be developed further into a fully operational version. His aim was not to develop a complete hardware device, but to demonstrate design methods that use Petri nets and their modeling power. In section 3 we present the Holton's processor, which demonstrates the fundamentals of processor operation. This processor will be the prototype of our asynchronous design.

In section 4 we introduce the basic design of Holton's processor using the same top-down specifications that used in [Sem97]. We produce a labeled higher-order Petri net that has only labeled transition with actions of the corresponding modules.

In section 5 the relation between the T-invariant and potential firing of the goal transition is discussed. A theorem has been stated and proved that can be used in analysing the labeled higher-order Petri nets properties such as safeness, liveness, and deadlocks free.

In section 6 we discuss the translating of the higher-order Petri nets into hardware elements. The asynchronous circuit synthesis corresponding to labeled higher-order Petri net is given. The conclusion is drawn in section 7.

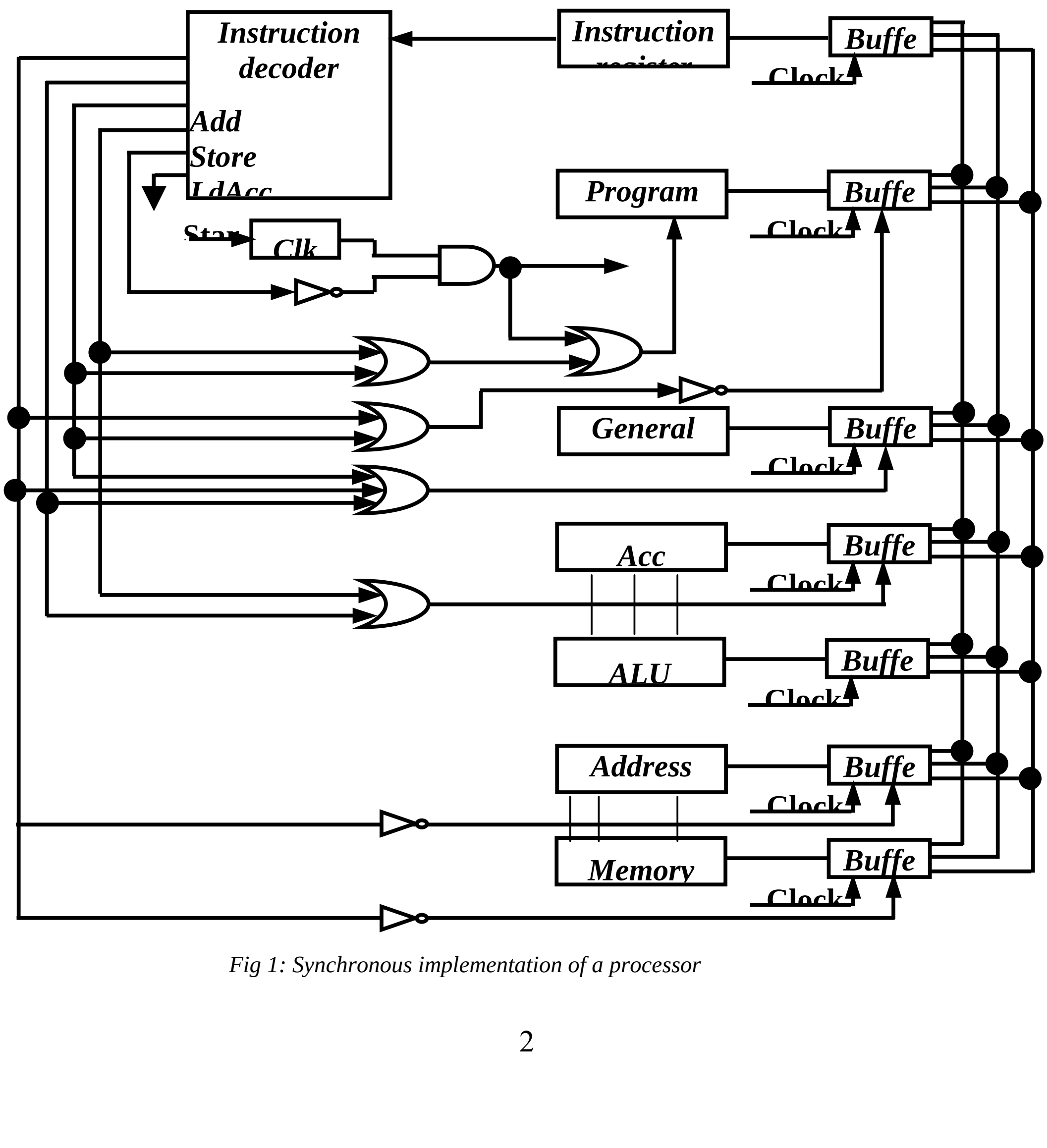

Fig 1: Synchronous implementation of a processor

2. Higher-Order Petri Nets (HOPN)

A classical marked Petri net is a 5-tuple PN=(P, T, F, W, M_o) , where P = { p_1 , p_2 , ……. , p_n } is a finite set of places, $P \neq Ø$; T = { t_1 , t_2 , ……. , t_m } is a finite set of transitions, $T \neq Ø$; $P \cap T = Ø$, $P \cup T \neq Ø$, m > 0, n > 0; $F \subseteq (T x P) \cup (P x T)$ is a set of arcs; $W:F \rightarrow N$ is the weight function; M_0: $P \rightarrow N$ is the initial marking.

Definition of a HOPN: A Higher_Order Petri Net is also a 5-tuple, HOPN= (P, T, F, W, M_0), where P, T and M_0 are defined as in the classical case; $F \subseteq (P x T) \cup (P^2 x T) \cup … \cup (P^n x T) \cup (T x P)$ is a set of arcs. IF $f\in (P^i x T)$, the f is called the ith-order arc; $W:F\rightarrow N$ is a weight function, where N represents the set of nonnegative integers. The major differences between an HOPN and a classical PN are the definitions of the arc and the weight function. In HOPN, we denote $f^{(i)}_{\{r(i)\}_k}$ where $f^{(i)}_{\{r(i)\}_k} \in (P^k x T)$, k= 1.2,… n and $W^{(j)}_{\{r(i)\}_k}$ as the kth order input arc of transition t, from places P_r (1), P_r (2),…, $P_{r(k)}$, and its corresponding weight respectively, where {r (I)} k= {r (I), r (2),… R (k)} $\subseteq$ {1, 2, …, n} and r (I) < r (2) <… < r (k). We also denote arc f_{tf} as the out put arc from transition t_I to place P_j, where $F_{tj} \in (T x P)$, and its corresponding weight as v_{ij}.

A transition t_j is enabled **iff**

$$\exists \; f^{(j)}_{\{r(i)\}_k} \in (P^k \; x \; T) \ni W^{(j)}_{\{r(i)\}_k} \leq Q^{(j)}_{\{r(i)\}_k} \; \forall \; P^{(j)}_{\{r(i)\}_k}$$

Where $P^{(j)}_{\{r(i)\}_k}$ = { $P_{r(1)}$, $P_{r(2)}$,…, $P_{r(k)}$} and { r (1), r(2),…, r (k)} $\subseteq$ {1, 2, …, n}, k= 1, 2, …, n,

Q ($P_{\{r(I)\}k}$) is the number of tokens in input places $P_{r(1)}$, $P_{r(2)}$, …, $P_{r(k)}$ connected on the kth order input arc $f^{(j)}_{\{r(i)\}_k}$ of transition t_j. This means that, the transition t_j is said to be enabled if at least one of its kth order input arcs has places that have at least many tokens as the weight of this kth order arc. Such an arc is called an enabled arc. This leads us to the fact that, it may be exist more than one enabled arc. But one enabled arc alone can make the transition enabled.

An enabled transition t_j may or may not fire. When t_j fires then one of its enabled arcs fires. Let the arc $f^{(j)}_{\{r(i)\}_k}$ fire Q (P $_{\{r(I)\}k}$) = Q(P $_{\{r(I)\}k}$) - $W^{(j)}_{\{r(i)\}_k}$, $\forall$ (P $_{\{r(I)\}k}$) $\in P_{IN}(t_j) \wedge Q(P_{out}(t_j)) = Q(P_{out}(t_j)) + W(P_{out}, t_j)$, $\forall P_{out}, (t_j) \in P_{out}(t_j), \in P_{out}(t_j), \forall P_{\{r(I)\}k} \in P_{IN}(t_j)$ Where Q (P $_{\{r}$ (I)} k) is the new number of tokens in input places $P_{r(1)}$, $P_{r(2)}$,…, $P_{r(k)}$ connected on the kth- order input arc $f^{(j)}_{\{r(i)\}_k}$ of transition t_j.

This means that when the transition t_j fires then the number of tokens in each of the input places $p_{\{r(I)\}k}$ related to the fired arc is reduced by the number that is equal to the weights assigned to the fired arc from $p_{\{r(I)\}k}$ to t_j, and the number of tokens in each of its output places increases by the number that is equal to the weights of the output arcs from the transition t_j.

The **reachability** set R (PN, M_0), for a higher- order petri net HOPN= (P, T, F, W, M_0), is the set of all markings that can be reached from its initial marking by all possible firings of transitions. R (PN, M_0)= { M_k: $M_k = \delta(M_0, \sigma)$, k= 0,1,2,……}, where σ is the sequence of all possible firing of enabled transitions initially with the initial marking M_0 until we reach to the marking M_k.

Safeness is one of the more important properties for higher- order petri net which is useful to model a real hardware device. A place P_I is safe iff $Q(P_I) \leq 1 \; \forall M_I \in R(PN, M_0)$, i= 0,1,2,….. A higher order petri net is said to be safe iff $Q(P_I) \leq 1 \; \forall M_I \in R(PN, M_0) \; \forall P_I \in P$, I= 0,1,2,…… That is, a higher order petri net is safe if all places in the net are safe at any marking.

A transition t_j is deadlock **iff**

$$Q(P_{in}(t_j)) < W^{(j)}_{\{r(i)\}_k}$$

$$\forall M_s \in R(PN, M_0) \; I, j, s, k \in K.$$

For all or some ($P_{in}(t_j) \in P_{IN}(t_j)$).

A transition t_j is deadlock (blocked) if there is no any reachable marking can make this transition enabled. Also a reachable marking is deadlock if it can not make any transition enabled. A higher-order petri net is deadlock at marking MK $\in$ R (PN, M_0) iff

$$Q(P_{in}(t_j)) < W^{(j)}_{\{r(i)\}_k} \quad \forall t_j \in T \quad I, j, s, k \in K.$$

For all or some (P_{in} , (t_j) $\in P_{IN}(t_j)$).

That is, a higher- order petri net is deadlock if all transitions are not enabled at certain marking. So a higher- order Petri net is free of deadlocks if its reachability set includes no deadlocks.

A transition t_j is live **iff**

$\exists M_I \in R(PN, M_0)$ s.t $Q(P_{in}(t_j)) \geq W(P_{in}, t_j)$ at M_I

$\forall$ P_{in} $(t_j)) \in P_{IN}$ (t_j) and $M_k = \delta$ (M_I, t_j), $M_k \in R$ (PN, M_0)

That is, a transition is live if it not deadlock. This does not mean that the transition is enabled at any marking, but rather than it can be enabled. In other word there exist marking $M_I \in R$ (PN, M_0) can make t_j enabled.

(i.e. $\exists$ $M_I \in R$ (PN, M_0) s.t $M_I \rightarrow M_k \in R$ (PN, M_0))

A higher- order petri net is live if $M_I \in R$ (PN, M_0) $\exists$ $M_k \in R$ (PN, M_0) s.t $M_k = \delta$ (M_I, t_j) $\forall$ $t_j \in$ T, I, j, k, $\in$ K. That is, a higher- order petri net is live its all transitions are live.

Transition t_i and t_j are in structural conflict if

$\exists$ $p_k \in P_{IN}(t_i) \wedge p_k \in P_{IN}(t_j)$

This means that transitions t_i and t_j are in structural conflict if they share at least one input place.

Transition t_i and t_j are in dynamic conflict if

i. t_I and t_j are in structural conflict.

ii. Q ($P_{in\,k}$ $(t_j)) \geq$ W ($P_{in\,k}$, t_j), Q ($P_{in\,s}$ $(t_I)) \geq$ W ($P_{in\,s}$, t_I)

$\forall$ ($P_{in\,k}$ $(t_j)) \in P_{IN}$ (t_j), $\forall$ ($P_{in\,s}$ $(t_I)) \in$ ($P_{in\,s}$, t_I) at marking M_1, where I, j, k, s, 1 $\in$ K.

iii. If t_I fires then Q (P $_{in\,k}$ $(t_j))$ < W ($P_{in\,k}$, t_j) for some or for all $P_{in\,k}$ $(t_j) \in P_{IN}$ (t_j).

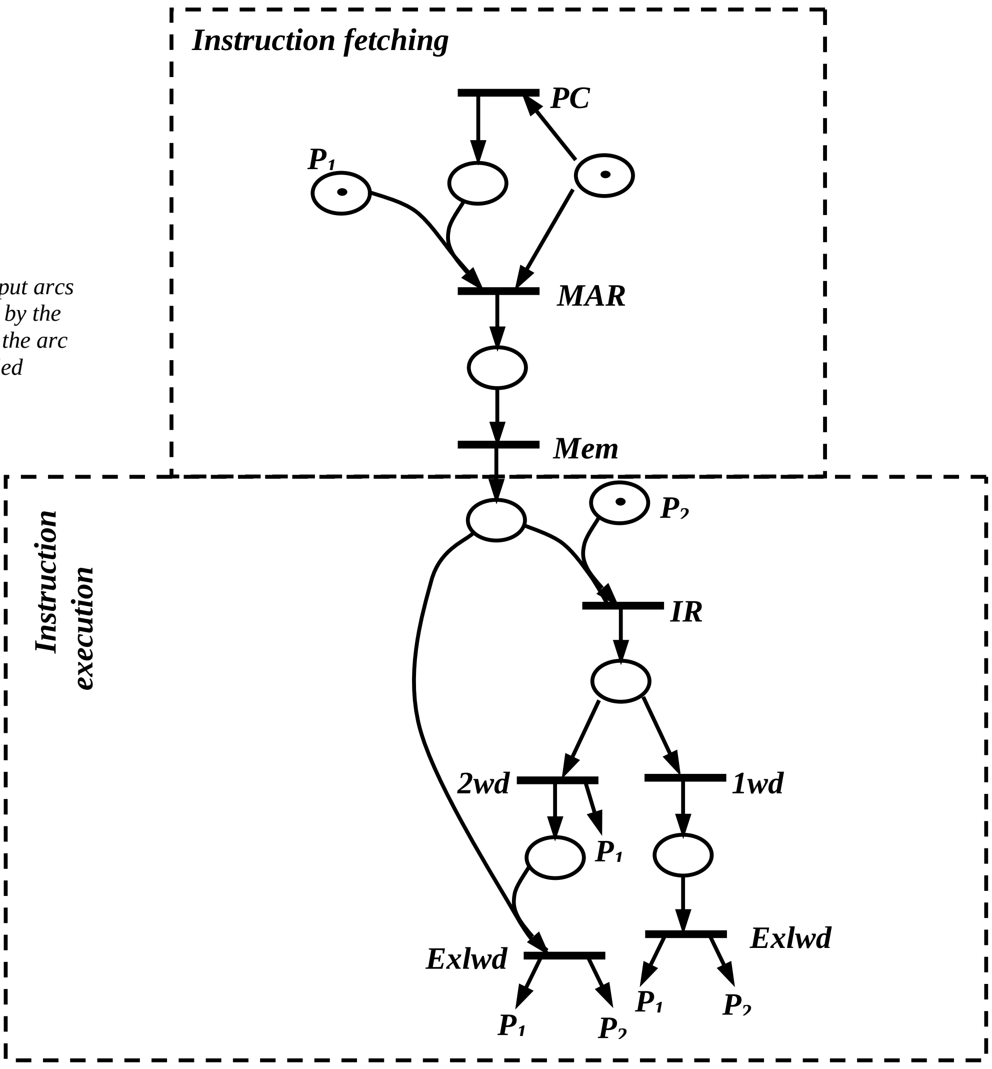

Fig 2: Refinement for top- level specification with simplified output arcs and labeled their desired places by the same label associated with arcs the arc goes to the place that is its labeled pointed

3. Synchronous processor model

[Hol77] introduces the description of a simple 3- bit processor design. This processor contains the major operational modules; IR, ID, PC, GR, Acc, ALU, AD, and Mem. All these modules contain the following instruction set; load accumulator (LdAcc), load general register (LdGR), arithmetic operation (Arth), and store. The processor has no jump instruction. Fig. 1 illustrates the architecture of Holton processor model. This design uses a common clock to synchronize data transfer between processor modules.

The processor's operational cycle is subdivided into two phases; Fetch cycle that performed by the program control unit, which must obtain the instruction from main memory. Instruction cycle that performed by the data processing unit, which must execute the instruction fetched from memory. Each phase requires two clock cycles. At the first cycle, the PC is increment and the new value of the PC is presented to memory. At the second cycle, MAR specify the address at which the fetched word is read , and reading it from memory at the specified address and latch the fetched word in the IR. Now the fetch phase is complete and the processor enters the execution phase. At the third cycle ID decodes the instruction, and then the appropriate modules are activated and connected to the common bus. At the last cycle, complete the execution of instruction fetched from memory.This model is synchronous and has some problems:

1- Loss of power. Since each module is clocked at each clock period, this will cause loss of power.

2- Inefficiency. 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.

We use HOPNs to construct the asychronous version of this synchronous processor. Especially, after producing an asynchronous one using PN [Sem 97]. In chapter 5 we give the representation of the asynchronous version of this processor using HOPN.

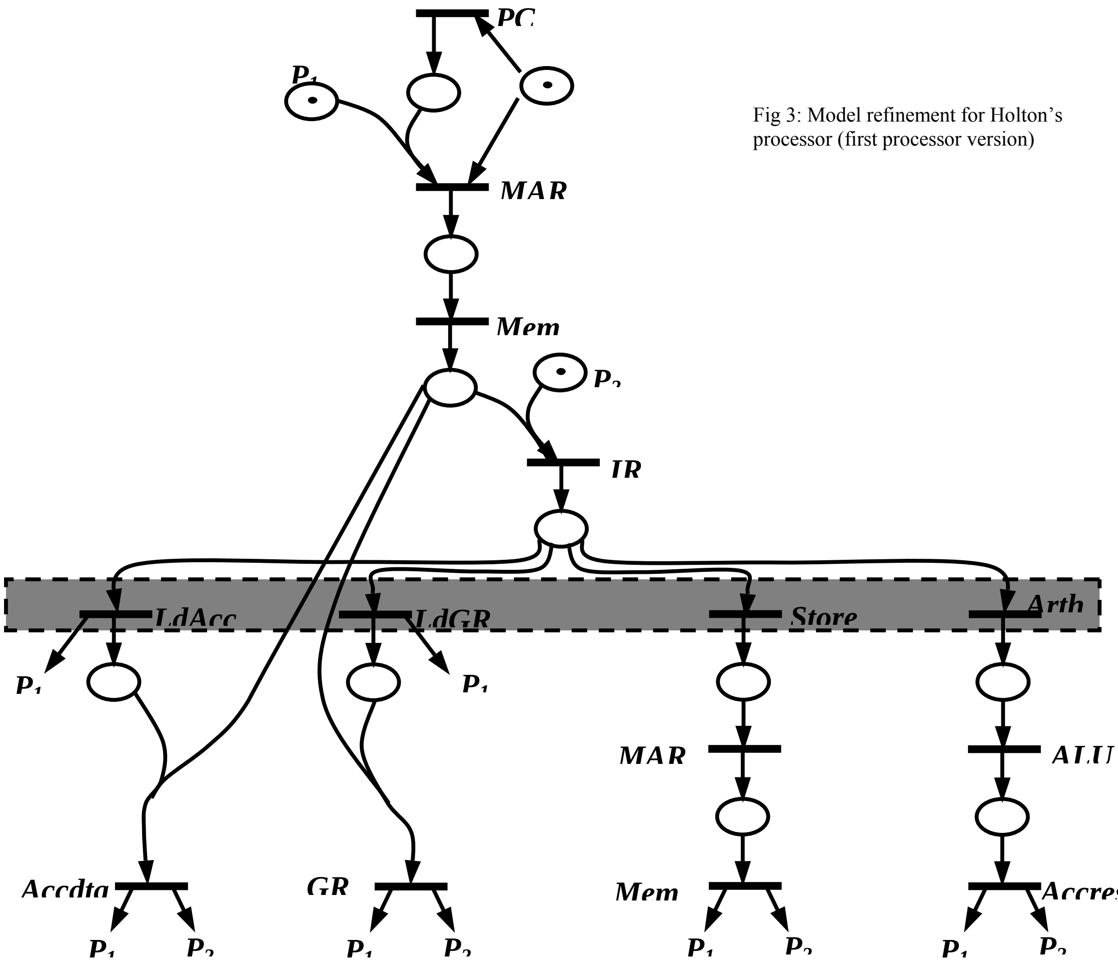

Fig 3: Model refinement for Holton's processor (first processor version)

4. Asynchronous design using HOPN

In this section we use the same refinement used in [Sem 97] to give labeled HOPN that contain labeled transitions (each one correspond to one module in Holton's processor). Transition labels are action of corresponding modules.

When we consider the initial specification (general consideration of the processor, such as the function of the processor, how the processor works, or the strategies of the work) we observe that, the processor has two stages instruction fetch (IF) and instruction execution (IE), in which the processor operates mutually (one follow the other, but not simultaneously). IF transition can be decomposed into PC, MAR_r, and Mem_r. IE transition can be subdivided into IR, 2 wd, Iwd, Ex1wd, and Ex2wd, IR, which decode instruction and types it into one word (1wd) or two word (2wd). This provides some facts. First, when the instruction typed into two words, ID must inform MAR for reading the second word. Second, after executing any instruction MAR must informed to provide to read a new word. Third, after executing any instruction IR must informed to ensure that the processor is idle and ready for latching new fetched instruction and sending to ID for decoding and execution. These refinement observations are illustrated in Fig 2. When an instruction is reached in IR, the ID decodes and executes this instruction. During the execution of an instruction, the IR must not change its content. So the condition "the execution complete and the processor is idle "must connect on input of IR to prevent IR from receiving new instruction, until the execution complete.

The two-word instruction is decoded into load accumulator or load general register. The load accumulator instruction is refined into decoded instruction (LdAcc) and execution action (Accdta). The execution action (Accdta) can not complete until the second word fetched from memory. The load general register instruction is also refined into decode instruction (LdGR) and execution action (GR). Also the GR wait for second word. One word instruction is decoded into Arithmetic or store instruction. The arithmetic instruction is suddivided into Arth, and execution action (Accres). The store instruction also subdivided into store, MAR $_w$, and execution action (MeM_w). Fig. 3 shows these refinements.

When we analyze the HOPN presented in Fig. 3, we find that, this HOPN is live (since any transition can be enabled at some reachable marking), free of deadlocks (any reachable marking make some transition enables), and safe (any place contain only one or zero token at any reachable marking). But the concurrency between the transitions is low. The operation of presenting data in all register never traverses with other operation. Therefore, any arithmetic instruction can be executed concurrently with fetching the next word from memory. When instruction is presented in the instruction register and decoded in the instruction decoder, then an acknowledgement can be sent to MAR to proceed (for reading new word). But the acknowledgment "the execution complete" is sent to IR (since the new fetched word wait at IR for proceeding). This improvement is shown in Fig. 4. Analyzing the behavior of the processor shows that, the degree of concurrency is still low. So we can observe that the instruction decoding may take a long time, and can proceed concurrently with fetching the next word from memory. The previous version could only allow fetching after the instruction was decoded. If the MAR receive an acknowledgment from the instruction register at earlier stage (before decoding instruction) to fetch the next word, while the instruction decoder decodes the instruction.

This means that, the fetching of the next word can be done simultaneously with instruction decoding. So when the execution of any instruction be complete, it must tell IR to end the current fetch, by sending new fetched instruction to ID and start new fetching. ID then begins the execution of the fetched instruction, which received from IR, and so on. If the instruction is classified into arithmetic/ store instruction, then arithmetic/ store proceed with fetching (which may be faster than the execution of instruction). The IR may receive new word, while the current instruction still in execution. So we must put additional place on input of both arithmetic and store transitions, to stop execution of any new instruction until the current one complete. If the store instruction is decoded, then it needs to access MAR (which may be busy with fetching). So we must guarantee that, if the MAR is used in fetching, then store must wait until the fetching is complete. Then both MAR $_w$ and MAR $_r$ transitions must share one place "condition". This place will represent the conditions "completion of execution", fetching second word", or "completion of fetching". This shared place will resolve the mutual excursion problem between a pair of requests to MAR. then this place will act as a semaphore for the actions involving MAR. independent requests to MAR have to complete for one token in this place, thus resolving the mutual exclusion problem. This place is shown as a dashed place Fig 5.

Unfortunately, if the instruction is decoded into store instruction, and sent a request to access MAR, simultaneously with the program counter's increment loop, then store may lose its request for the mutual exclusion token (since, if MAR $_r$ fires then MAR $_w$

must wait for the next time). At this instant, the HOPN will deadlock (no transition be enabled). The new fetched word will not be able to advance because it is waiting for the instruction register to be cleared, at the same time, the instruction register waits for completing store. Now, for solving this problem, we need an additional register to store the new fetched word temporarily, and allow MSR to accept the request from store (to balance between the two requests for MAR). Analysis with this modification leads to the fact that, IR must restrict PC's increment if and only if the additional register is pipeline. This is introduced in the form of an additional dependency constraint (new place) which is dashed in Fig 5.

Verification of this HOPN shows that it is live, safe, free of deadlocks, and the conflict between MAR's requests have been resolved. Also the degree of concurrency will increase. Now, if we want to increase the degree of concurrency we introduce a second additional register. As shown in Fig 6, this second additional register gives us higher degree of concurrency between transitions. Analysing this HOPN shows that this HOPN is live, safe, and deadlocks free. Fig 6 gives us the fourth version of the processor. We will stop at this version as an end to the analysis. For the concurrency degree see [Sem 97].

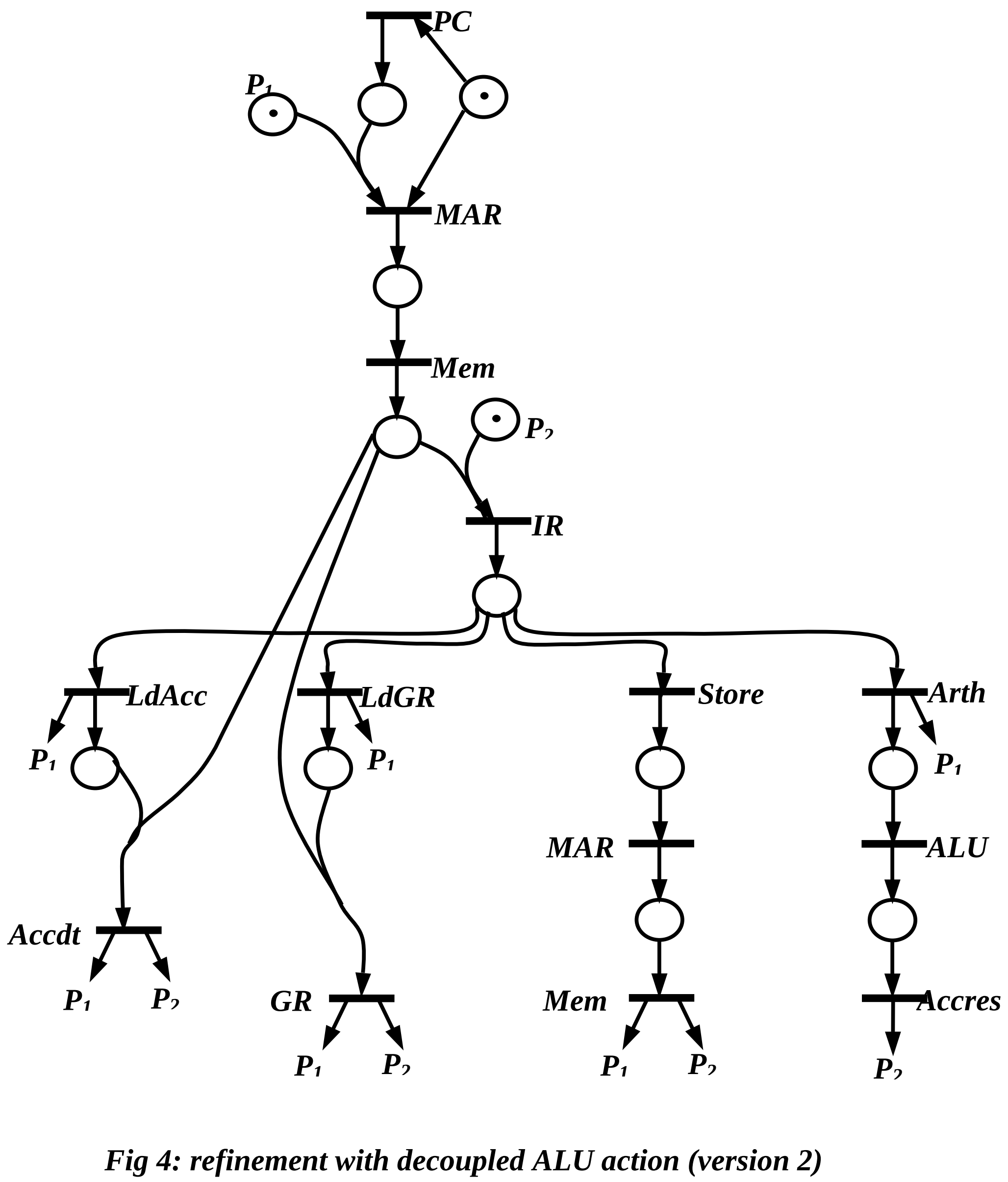

Fig 4: refinement with decoupled ALU action (version 2)

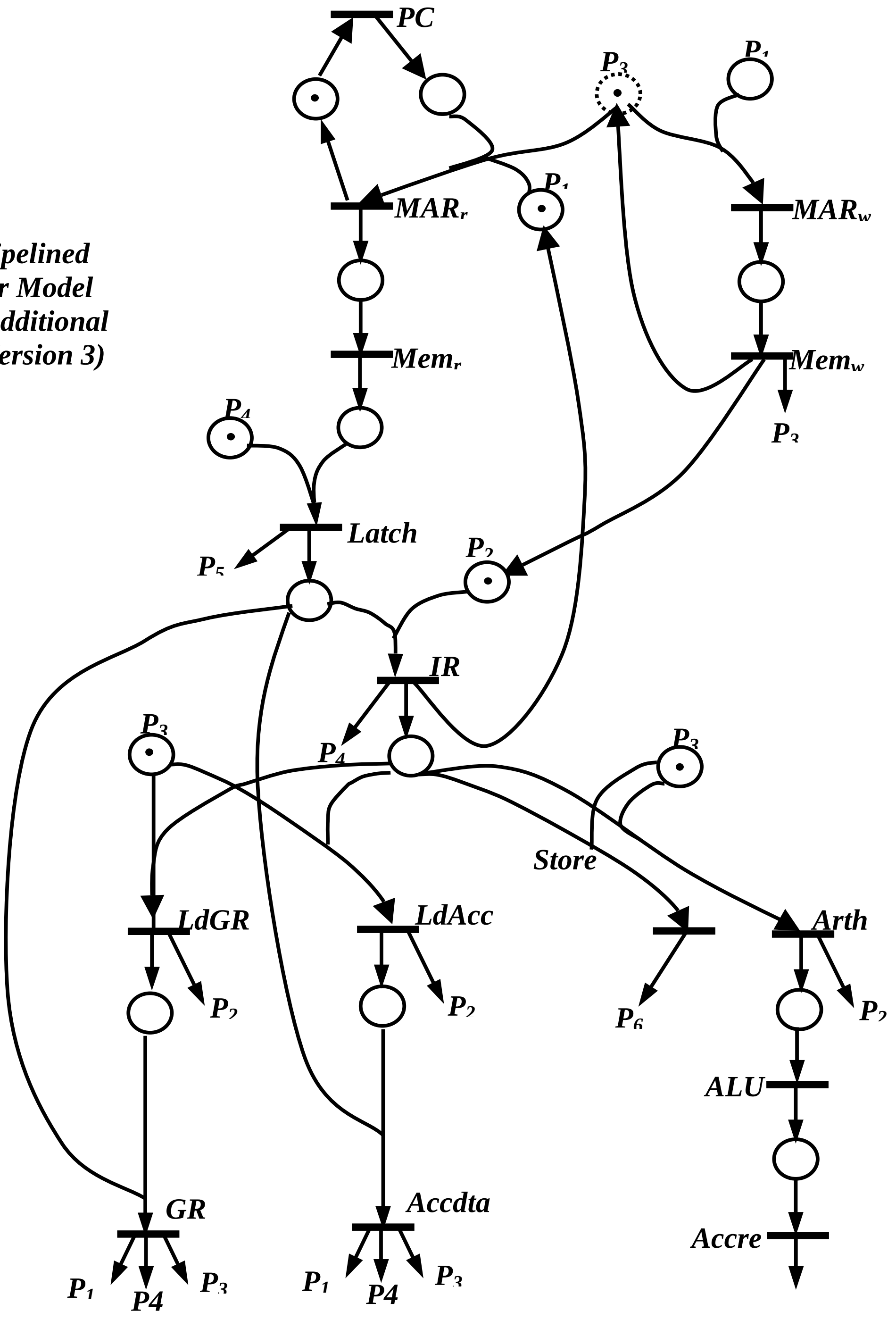

Fig 5: Pipelined Processor Model with one additional register(Version 3)

5. Validity Analysis of processor (version 4)

We use the relation between the T- invariant and the potential firability of the goal transition to check the validity of our design. This depends on finding valid sequence to execute the instruction and return to the initial marking after completion of execution (firing the goal transition). A sequence σ must be found including the goal transition (the transition that completes the execution). There exists a relationship between σ and T- invariant of the HOPN. The following theorem gives this relation

Theorem. Let HOPN= (P, T, E, W, M_0) be a higher order Petri net that has all transitions with indegree $\leq$ 1. Let t_g be a goal transition in T. there exists a firing sequence σ to reproduce the initial marking M_0 and to fire the goal transition t_g iff HOPN has a T- invariant X such that $X \geq 0$ and $X(t_g) \neq 0$.

Proof. First we prove the necessity.

If there exists a sequence σ, from the relation between markings $M= M_0 + f(\sigma). A$, where $f(\sigma)$ is the count vector of σ whose entry j denotes the occurrence of t_j in σ, and A is the incidence matrix. This leads to $M= M_0$, thus the product $f(\sigma). A$ must be equal zero. Let $f(\sigma)= X', X'$ is m- vector (row vector). Then $X'. A= 0$ $A^{tr} = 0$, Where $X' = X'^{tr}$ (i.e. X is column vector). Then HOPN has T- invariant X such that $X \geq 0$ and $X(t_g) \neq 0$.

The sufficiency can be proved as follows:

Let he HOPN has T- invariant X such that $X \geq 0$ and $X(t_g) \neq 0$.From the definition of T- invariant, we find that $A^{tr} \circ X= 0$, X is column vector. This leads to $X'. A= 0$, X' is the transition of X, since $X \geq 0$ then $X' \geq 0$, also $X'(t_g) \neq 0$. From the relation between markings, we find that $M= M_0 + X'. A$ $M= M_0$, which means that, there exists a sequence σ (which is count vector X') includes the goal transition and reproduces the initial marking.

We apply this theorem to the HOPN in Fig 6. It is clear that there exists a sequence corresponding to executing each instruction. For add instruction there exist the sequence σ_1= (PC, MAR_r, Mem_r, $Latch_1$, $Latch_2$, IR, Arth, ALU, Accres), where the goal transition is “Accres”. For the store instruction there exist the sequence σ_2= (PC, MAR_r, Mem_r, $Latch_2$, IR, Store, MAR_w, Mem_w), where the goal transitions is “Mem_w”. For Load General Register (LdGR) instruction there exist the sequence σ_3= (PC, MAR_r, Mem_r, $Latch_1$, $Latch_2$, IR, LdGR, PC, MAR_r, Mem_r, $Latch_1$, $Latch_2$, GR), where the goal transition is “GR”. For load Accumulator (LdAcc) instruction there exist the sequence σ_4= (PC, MAR_r, Mem_r, $Latch_1$, $Latch_2$, IR, Ldacc, PC, MAR_r, Mem_r, $Latch_1$, $Latch_2$, Accdta), where the goal transition is “Accdta”. For these four sequences σ_1, σ_2, σ_3, and σ_4 there are vectors X_1, X_2, X_3, and X_4 respectively. The entry j in any vector X_I represents the occurrence of the corresponding transition the transitions are ordered as they arranged in the incidence matrix). These vectors X_1, X_2, X_3, and X_4 can be interpreted as they arranged in the incidence matrix). These vectors X_1, X_2, X_3, and X_4 can be interpreted as re- T- invariant as they satisfy the condition $A^{tr} \circ X_I= 0$, I= 1,2,3,4 and $^\circ$ is the matrix product, A is incidence matrix [Tom 97]. From this we can ensure that, the implementation is accurate. Also the connections in HOPN are well defined, and each instruction can execute without interrupts (deadlock free). After execution of each instruction the marking returns to initial marking M_0 (liveness), and the HOPN is safe. To show the hardware equivalent of a HOPN, we can perform two operations:

1. Net-level transformation. As an example a kth-order arc can be transformed into a sequence of 2nd-order arcs, as in Fig. 7(a).

The complete set of these transformations is given in [Has01]. The corresponding asynchronous circuit is presented in Fig 8. The heavy lines represent the data transfer between modules.

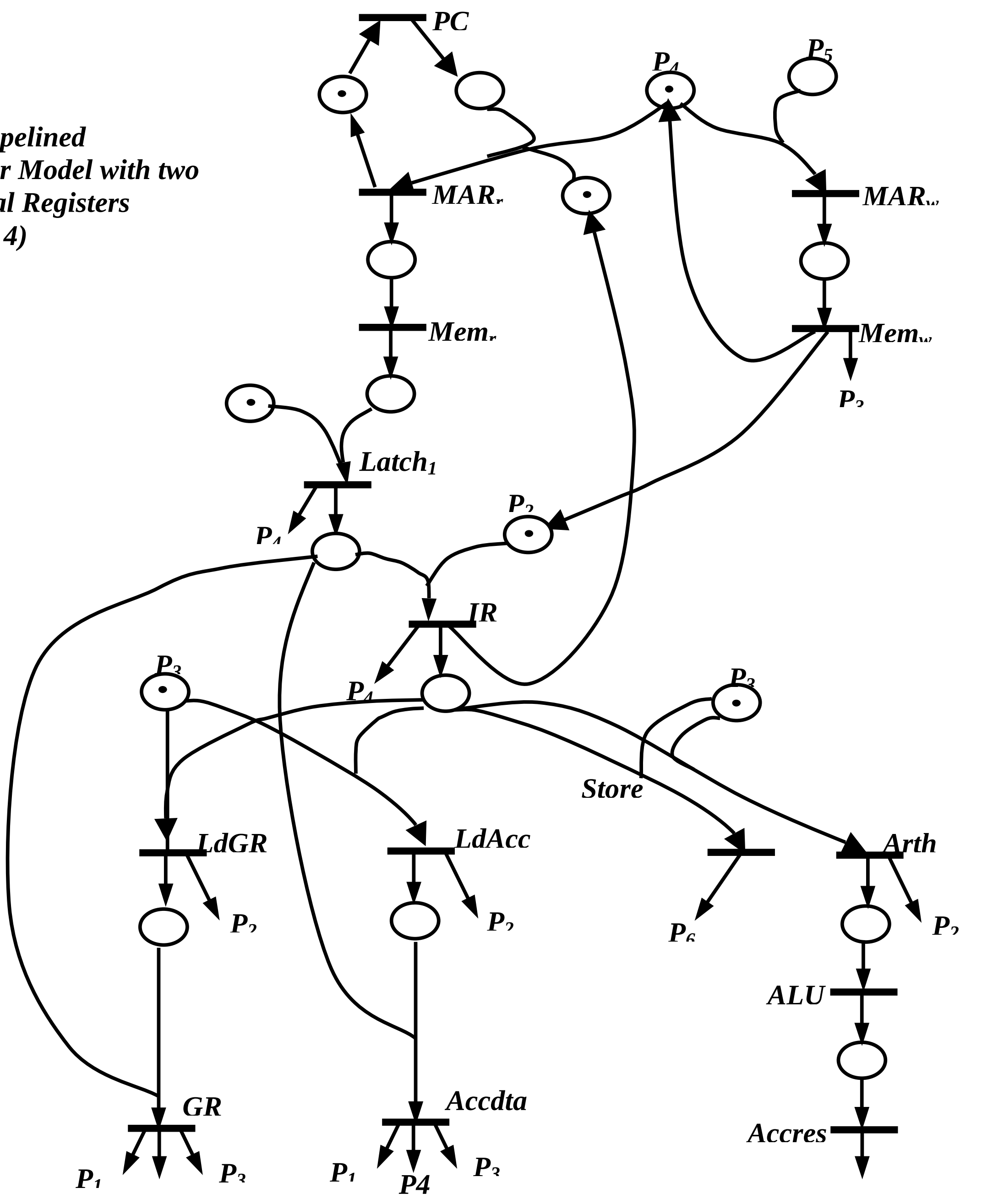

Fig 6: Pipelined Processor Model with two additional Registers (Version 4)

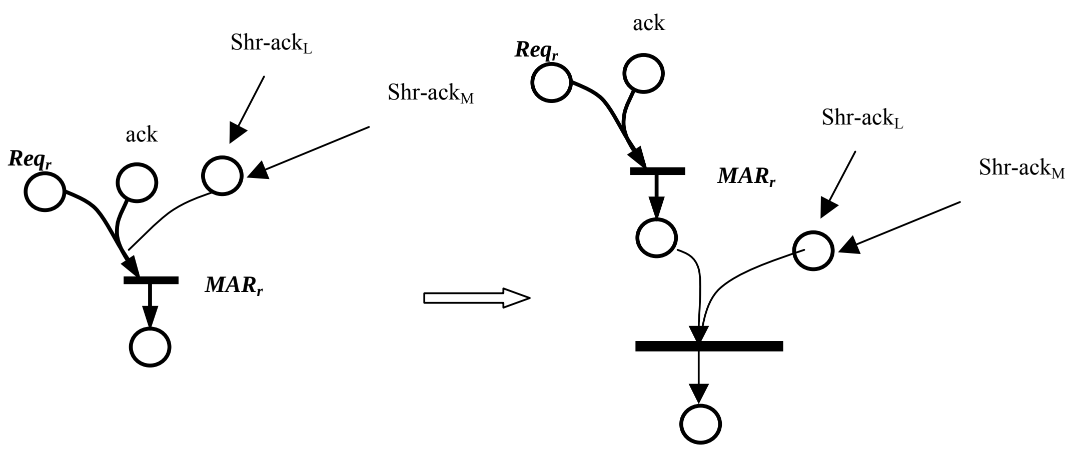

Fig 7(a): Dividing a 3rd order arc into two 2nd order arcs and an additional transition

2. Circuit synthesis. The second order arc can be translated into a Muller C element, as in Fig. 7(b)

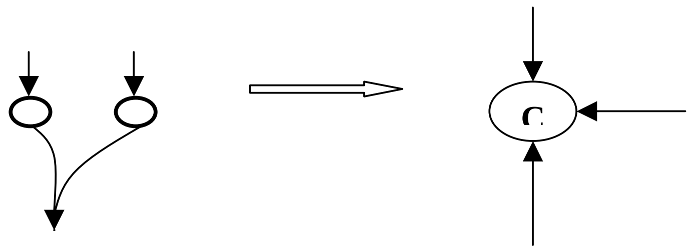

Fig. 7(b) A second order arc translates into a Muller C element

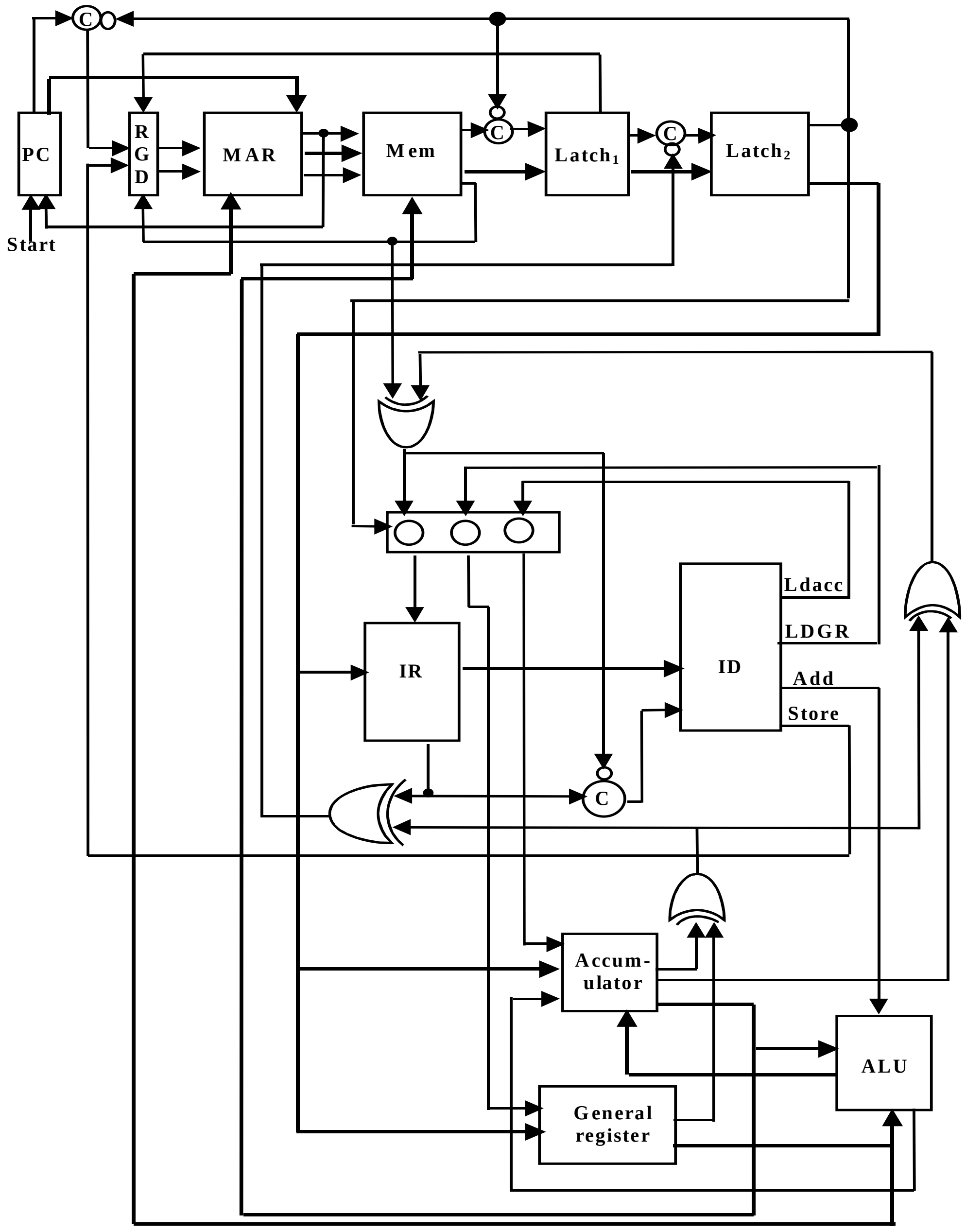

Fig 8: Circuit Synthesis of Holton's Processor (Version 4)

6. Conclusion

Higher-Order Petri nets have been successfully used to model an asynchronous processor. This proves that this new class of Petri nets is able to model general asynchronous systems. The fact that we have considered a special processor does not affect the generality of the approach. It was shown that the behavior of an event in HOPN is similar to that of an asynchronous circuit. Using top-down approach has enabled us to refine different versions of the processor. In each case we have ensured that the liveness, safeness, and deadlock-free properties. From the analysis point of view, the existence of an executable sequence is guaranteed. A theorem concerning the relationship between the potential friability of the goal transition and the T-invariant has been proved. When the goal transition fires, it completes the instruction and returns to the initial marking. The practical applications of the theorem cover other fields. We have found that HOPNs are capable to describe the actions of asynchronous circuits. HOPNs can better describe the control that is necessary for transforming the nets into asynchronous circuits than the classical PNs.

References

[Ajm84] M. Ajmone et al.
A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems.
ACM Transactions on Computer Systems, 2(1984), pp. 93-122.

[Cho97] Tommy W.S. Chow, Jin-Yan Li
Higher-Order Petri Net Models Based on Artificial Neural Networks.
Artificial Intelligence 92(1997), pp. 289-300.

[Dav92] Davis, A. L.
Mayfly: A General-Purpose, scalable, parallel processing architecture.
LISP and Symbolic Computations Vol 5, May 1992, pp. 7-47.

[Fur 93] S.B. Furber et al.
A Micropipelined ARM. Proc. VLSI 93, North Holland, Amsterdam, 1993, pp. 4.4.1-5.4.10.

[GEE 05] David Geer
Is It Time for Clockless Chips?
Computer(IEEE), March 2005 , pp. 18-21

[Gen81] Genrich, H. J. and Lautenbach, K.
System Modeling with High Level Petri Nets
Thepretical Computer Science, 13(1981), pp.109-136.

[Has01] Mosaad W. S. A. Hassan
Asynchronous Processor Design Using Higher-Order Petri Nets.
Master Thesis, Faculty of Science, Menoufya University, 2001.

[Hol 77] W.C.Holton
The Large-Scale Integration of Microelectronic Circuits
Scientific American, 1977, pp. 82—94.

[Mar90]A.J. Martin
Collected Papers on Asynchronous VLSI Design.
Tech. Report CS-TR-90-09, Calif. Inst. Of Technology, Pasadena, 1990.

[Rei85] Reisig, W.
Petri nets: An introduction. Prentice-Hall Englewood Cliffs. 1985.

[Sem97] Semenove, A et al.
Designing an Asynchronous Processor Using Petri nets. IEEE Micro, 1997, pp. 54-63.

[Tak94] T. Nanya et al.
TITAC: Design of Quasi-Delay-Insensitive Microprocessor", IEEE
Design & Test of Computers, Vol. 11, No. 2, Summer 1994, pp. 50-63.

[Van89] Van Hee, K. M. et el.
Executable Specifications for Distributed Information Systems.
Proc. IFIP Conference on Information Systems Concepts, Elsevier Science Publ., Amsterdam 1989, pp. 139-156.