# The Use of Petri Nets for Modeling Pipelined Processors

Rami R. Razouk

Department of Information and Computer Science
University of California, Irvine

## Abstract

This paper discusses the use of Petri Nets for modeling and analyzing pipelined processors. Petri Nets are particularly well-suited to modeling the synchronization, buffering, resource contention and delicate timing so common in pipelined processors. Tools for simulating, animating and analyzing the behavior of the models are described. The usefulness of the tools and the analysis methods they support in evaluating the performance and analyzing the detailed timing of pipelined microprocessors is illustrated through an example.

## Introduction

As single-chip processors become more powerful, the use of pipelining to speed up instruction fetching, decoding and execution has become more prevalent. Modern microprocessors such as the MC 680x0 and Intel x86 support features such as instruction pre-fetching, instruction and data caches, and pipelining of decoding and execution. Understanding the detailed timing of such processors is extremely difficult and therefore understanding the bottlenecks in systems which use them is also difficult. For example, memory speed and processor clock rate can have a strong yet difficult to predict impact on the performance of micro-processor-based computer systems. This paper addresses the needs for tools which can permit rapid construction of faithful models of pipelined processors.

The model proposed in this work is an extended Timed Petri-Net model. Petri Nets have been proposed as useful modeling tools for hardware systems by Misunas [Mis73], Ramchandani [Ram74], Agerwala [Age79], Ramamoorthy and Ho [RH80], Zubereck [Zub80], Razouk and Phelps [RP84], Holliday and Vernon [HV85], and many others. Invariably researchers interested in modelling hardware systems extend classical Petri Nets [Pet81] to include some notion of time. The model described in this paper includes extensions which are essential to making the construction of the models straightforward and ensuring faithful mimicing of timing behavior. While many researchers have proposed similar extension and discussed how Petri Nets can be used to model pipelining, the contribution of this paper is in describing a collection of tools (The P-NUT system) that support rapid construction and analysis of Petri Net models. The tools described herein support classical simulation capabilities as well as novel animation and timing analysis features.

This work has been supported in part by the Microelectronics Innovation and Computer Research Opportunities (MICRO) program cosponsored by Hughes Aircraft Corporation, and by the National Science Foundation under Grants DCR-8406756 and DCR-8521398.

Section 1 of the paper introduces our "flavor" of Petri Nets and highlights the features of the model which make the modeling of pipelining easy. Section 2 presents an example of a pipelined processor of moderate complexity. Extensions needed to model even more complex systems are discussed in Section 3. Section 4 of the paper discusses the types of analyses needed to evaluate pipelined processors effectively.

## 1 Petri Nets and Modeling Pipelining

A Petri Net model of a hardware system consists of a description of a set of possible events in the system. Each event has pre-conditions which must be true in order for the event to occur, and post-conditions which become true once the event has occurred. In pipelined processors typical events include: initiate instruction prefetch, issue instruction to functional unit, initiate operand fetch, initiate storing of result, etc... Typical preconditions for an event such as "initiate instruction prefetch" can be: bus is free, space is available in instruction buffer, no operand-fetch is pending.

In Petri Nets events correspond to transitions (typically drawn as lines or boxes) and conditions correspond to places (typically drawn as circles or hexagons). Constructing a Petri Net model of a system involves enumerating all events in the system and listing their pre- and post-conditions. The order in which the events are listed is irrelevant. Conditions which are true at some point in time are modeled by tokens on places. Boolean conditions are modeled by the presence or absence of tokens. More complex conditions can be modeled by having some number of tokens on a place. Events whose pre-conditions are met (tokens are present on their input places) may occur (the transition may fire) thereby removing tokens from inputs (disabling the pre-conditions) and producing new tokens on the outputs (enabling post-conditions). Since many pre-conditions may be satisfied at the same time, parallelism is a natural fall-out of the model. The modeler need not explicitly address parallelism. Similarly, events which share some common place can contend for the token on that place. Mutual exclusion and other forms of resource contention can be easily modeled with shared places.

Figure 1 shows a brief example showing how pre-fetching of instructions into an instruction buffer can be modeled. The situation being modeled is a buffer pool of 6 words (16 bits each) that can be pre-fetched two-at-a-time. The buffer pool is modeled by place **Empty-I-buffers**. The fact that the buffers are used two-at-a-time is modeled by assigning a weight of 2 to the arc into transition **Start-prefetch**. Pre-fetching is initiated whenever the bus is free and there are no operands waiting to be fetched or re-

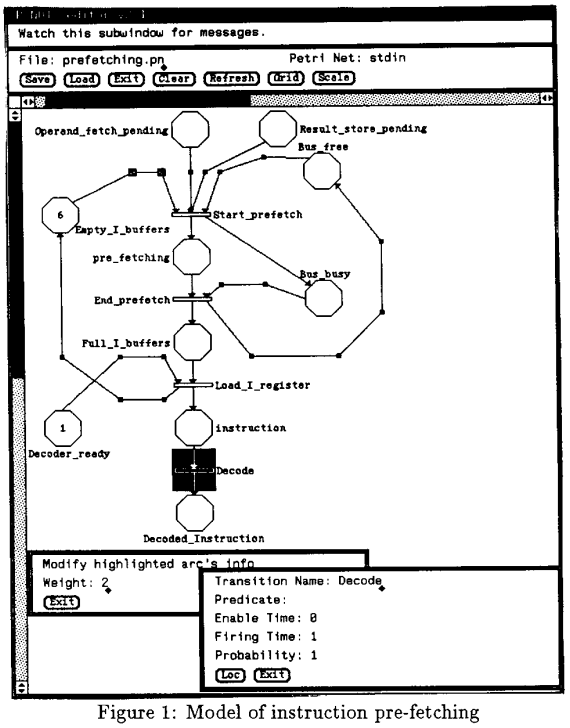

Figure 1: Model of instruction pre-fetching

sults waiting to be stored. These latter conditions are inhibiting conditions requiring inhibitor arcs, a common and convenient extension to Petri Nets. These arcs are depicted by dark bubbles in Figure 1. The instruction words are decoded one-at-a-time.

In addition to the simple primitives presented above, Timed Petri Net [Ram74,Zub80,RP84,HV85] support the modeling of time. Events which take time are modeled with transition which have *firing* times. During the firing of a transition tokens are neither on the inputs nor on the outputs. Another form of time is the *enabling* time. This is a delay during which the transition must be continuously enabled before it is allowed to fire. This form of time is particularly convenient for modeling timeouts in communications protocols. Figure 1 contains examples of both enabling and firing times. The fact that decoding an instruction requires one processor cycle is modeled by a firing time on transition **Decode**. The time required to complete a memory fetch is modeled by an enabling delay on transition **End-prefetch**. It should be pointed out that firing times can be easily simulated using enabling times but the opposite is not true. Firing times are therefore a convenience for modeling but are not a necessity. The introduction of time into Petri Nets enables the modeler to easily construct model of synchronous pipelines as well as the asynchronous pipelines discussed in this paper. For the sake of brevity we have omitted the discussion of synchronous pipelines although modeling them is also supported by the P-NUT system.

Another extension to Petri Nets which is necessary to support preformance evaluation is the introduction of probabilistic behavior. Each set of competing events are assigned (by the modeler) relative firing frequencies from which firing probabilities are dynamically computed during simulation (or analysis) [WPS86].

The final extension supported by new releases of the P-NUT system are predicates and actions associated with transitions. Predicates allow users to specify data-dependent pre-conditions which must be true for an event to occur. Actions are used to describe data transformations associated with particular events. These are powerful extensions which make the construction of complex models simple. The intent of the extensions is to allow the user to focus on modeling parallelism, resource contention and synchronization using Petri Nets, while supporting the construction of detailed models. Examples of the use of predicates and actions are discussed in Section 3.

## 2 An Example

In this section we present a small (yet interesting) model of a pipelined microprocessor. The purpose of this example is to illustrate the ease with which hardware concepts map onto Petri Net modeling concepts. The resulting high-level models are brief (generally less than a page) yet model a significant fraction of the detailed timing behavior of the system. The example pipeline has the following features:

1. 3-stage pipeline. The first stage is responsible for pre-fetching instructions, the second stage is responsible for decoding, address calculation and operand fetching, and the third stage is responsible for executing the instruction and storing the result (if necessary). Each instruction has a .2 probability of storing a result.

2. Pre-fetching is initiated whenever a free cycle on the bus is detected, there is enough room in the instruction buffer and there are no pending reads/writes from/to main memory.

3. The instruction buffer consists of 6 16-bit words. Each word contains a single instruction (a generalization of this is discussed later).

4. There are three types of instructions: zero-memory-operand (register only) instructions, one-memory-operand instructions and two-memory-operand instructions. They occur with frequencies 70-20-10.

5. Decoding requires one processor cycle. Address calculation requires two processor cycles for each memory operand. Instruction execution requires 1-2-5-10-50 processor cycles with probabilities .5-.3-.1-.05-.05 respectively A memory access requires 5 processor cycles.

The resulting model is shown in Figures 1-3. Figure 1 shows the model of instruction pre-fetching and was discussed in Section 1 of the paper. Figure 2 shows the model for decoding, address calculation and operand fetching. The instruction mix is modeled by assigning firing frequencies to transitions **Type-1**, **Type-2** and **Type-3**. The load placed on the bus as a result of operand fetching and the fact that operand fetching inhibits instruction prefetching are explicitly modeled. The delays introduced by the calculation of the operand effective addresses are modeled by transition **calc-eaddr**. Place **Decoder-ready** is used to represent the second stage of the pipeline (a physical resource). Figure 3 shows the model of instruction execution and result storing. Place **Execution-unit** models the third stage of the pipeline. Five separate transitions, with appropriate firing frequencies and firing times, are used to model the five execution

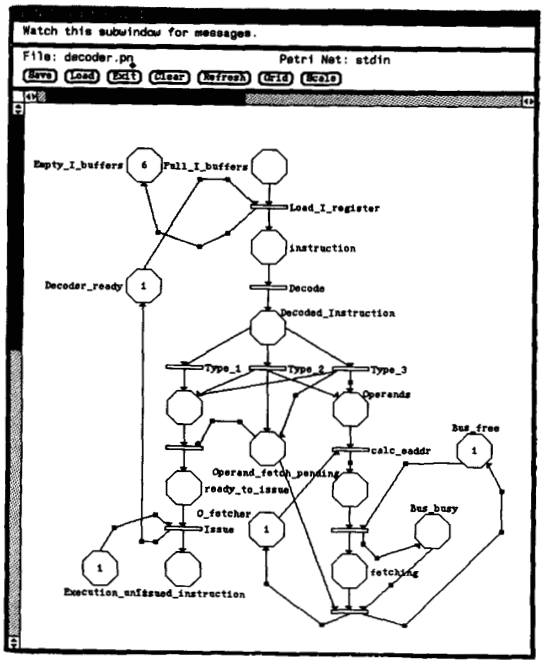


Figure 2: Model of decoding, address calculation and operand fetching

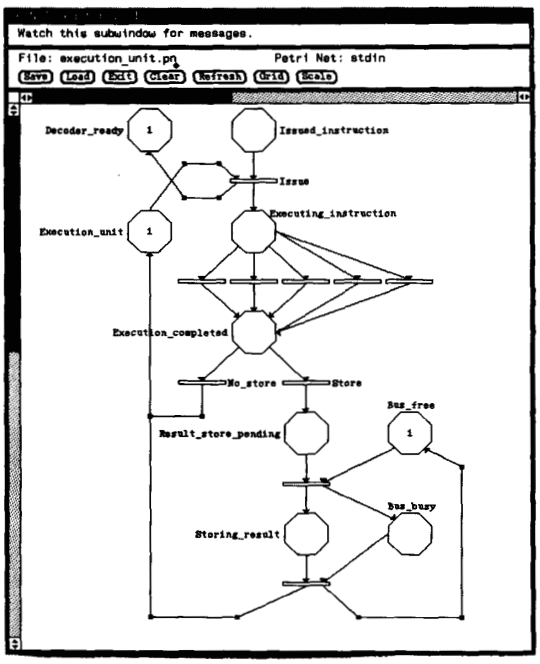


Figure 3: Model of instruction execution

delays. The contention for the bus which results from storing of results is explicitly modeled in the execution unit.

The resulting complete model can be expressed graphically in one or two pages and textually (for some of our textually based tools) in roughly 25 lines. Clearly the model is simplistic, but, with a few additional extensions, more complex models can be described nearly as tersely.

## 3 More Complex Pipelined Processors

The model described above omits many common aspects of modern microprocessors. Instruction and data caches are quite common and can be easily modeled probabilistically, assuming some given hit ratio. A more difficult issue to handle is the fact that the instruction set of a modern microprocessor is quite complex and normally involves variable length instructions. The use of individual transitions to model each instruction type results in extremely complex nets. It can be argued that the net complexity approach that of other simulation models. This is an area where the selective use of predicates and actions can result in a Petri net model which is no more complex than the one described above yet which accurately models the detailed processing of the instruction set.

The three areas where the instruction set affects the model are in decoding variable length instructions, fetching operands, and executing instructions. Typically modern microprocessors may support as many as 30 addressing modes, each of which requires different length instructions, and places a different load on the bus to main memory. Rather than using a separate subnet for each addressing mode it is possible to construct a table-driven model of the instruction set. One transition in the net can randomly select the instruction type (according to some distribution) and the remaining parts of the net use the instruction type to remove additional words from the instruction buffer, and to calculate firing times, enabling times and the number of times to iterate through loops (to fetch operands for example). The Petri net itself would be used to model what Petri nets model best: the contention for the bus and the synchronization between different portions of the pipeline. Figure 4 shows the skeleton of such a model. The interaction with the instruction buffer has been omitted in this example. The transition **Decode** has the action:

```
[[][type] type = irand[1, max-type];
        number-of-operands-needed = operands[type]; ]
```

where **irand** randomly selects the instruction type and **operands** is a table containing the number of operands for each instruction type.

The predicate for transition **operand-fetching-done** is

```
[[][] number-of-operands-needed == 0 ]
```

and the predicate for transition **fetch-operand** is

```
[[][] number-of-operands-needed > 0 ]
```

Finally, the action for transition **end-fetch** is

```
[[][] number-of-operands-needed =
  number-of-operands-needed-1 ]
```

Instruction execution can be modeled similarly. Execution delays can be calculated based on instruction type as can the number of required reads/writes from/to memory. Again the Petri net focuses exclusively on modeling contention for the bus.

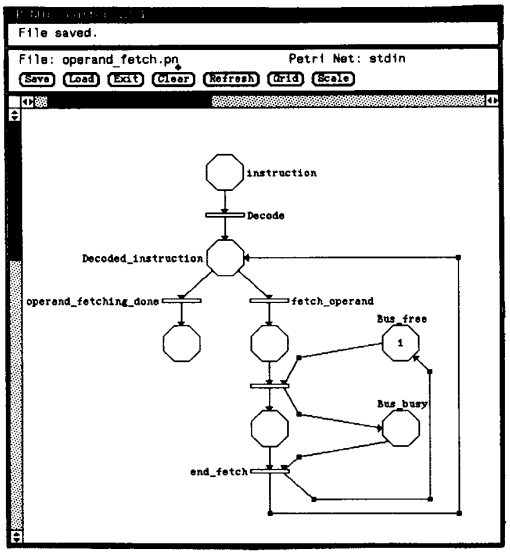

Figure 4: Interpreted net for operand fetching

# 4 Analysis Tools for Pipelined Processors

The previous section discussed the reasons that make Petri Nets particularly well-suited to modeling pipelining. Some of the extensions discussed above have been used by other researchers [Bae80,Zub80,HV85] The contribution of this paper is in presenting a set of tools which effectively support these extensions and form an effective environment for modeling and analysis of computer systems. The set of tools is called P-NUT (for Petri-Net Utility Tools). The P-NUT system include tools for constructing and analyzing complete reachability graphs (timed [RP84], and untimed [MR87]) and for performing simulation experiments from which detailed timing analyses can be derived and from which performance estimates can be obtained. In the sections below the P-NUT tools that support simulation, performance evaluation, animation, and timing analysis are described.

## 4.1 Simulation

The P-NUT simulator is a simple simulation engine which "pushes" tokens around a Timed Petri Net. The input to the simulator is a Petri Net and a few simulation commands that allow a user to control the duration of one or more simulation experiments. Unlike most simulation tools, the simulator knows little about how to analyze the simulation results. That task is left up to specialized tools. The simulator simply generates a **trace**. A trace is simply the description of the initial state of the system, followed by a series of state deltas describing how the state of the system changes over time. The decoupling of simulation engine from analysis tool is extremely useful since the intermediate trace representation need not be made specific to a particular modeling technique. Traces can be easily generated from SIMSCRIPT simulations as well as any other simulation language.

One problem with simulation traces is that they can be extremely large. The size of a simulation trace can become unwieldy if the length of the trace is large or if the information retained about the system is too detailed. By default the P-NUT simulator retains all information about all places and transitions in the net. Yet, usually only a handful of places and transitions are of interest in performing a particular analysis. The P-NUT system therefore provides a **filter**ing tool from which significantly smaller traces can be obtained. While filtering solves the problem of too much detail it doesn't solve the problem of very long traces. In order to support long simulation experiments the simulator was designed so that its output can be directly "plugged" into the input of analysis tools, thereby eliminating the need for storing large files.

## 4.2 Performance Evaluation

Traditionally, simulation experiments are performed to obtain accurate performance estimates. P-NUT supports such classical uses of simulation by providing a statistical analysis tool whose purpose is to extract from simulation traces performance related information. It is important to note that the data provided by the statistical analysis tool is provided in terms of places and transitions. The mapping between this information and higher-level concepts such as processor utilization is left up to the user. This mapping, however, is usually straightforward. The following section illustrates some of the information which can be obtained by carefully analyzing the model described in section 2.

In order to properly interpret simulation statistics a careful mapping must be done from the modeling primitives back to some higher level concept. This mapping is, of course, needed for any type of model. The problem is discussed here to point out some subtle aspects of modeling using Timed Petri Nets. The statistical analysis package in P-NUT (**stat**) reports information about places and transitions. The information about places includes the average (over time) number of tokens in a place. The significance of this number depends on how places are used to model various aspects of the computer system. For example, the availability of the bus was modeled in Section 1 (See Figure 1) by using two mutually-exclusive places: **Bus-free** and **Bus-busy**. The semantics of these two places is that the sum of the tokens on these two places should always equal one: the bus is either free or busy. This requires that all events which move tokens from Bus-free to Bus-busy (and back) be instantaneous (no firing times). If that is the case then the average number of tokens on Bus-busy describes the utilization of the bus. A more detailed breakdown of the activity on the bus can be obtained by looking at places **pre-fetching**, **fetching** and **Storing-result**. These will provide the usage of the bus for pre-fetching instructions, fetching operands and storing results. Similarly, the utilization of instruction buffers, of the decoder and execution units can be easily obtained.

The statistical analysis package also provides information about the average number of concurrent firings of a transition. Typically only one instance of a transition is firing at any point in time, therefore the average number of concurrent firings refers to the utilization of that transition (percent time it is busy). Theoretically, it possible for many instances of a transition to be firing concurrently, modeling the fact that a particular event (an instruction entering a queue) may be ocuring several times simultaneously. This is particularly useful in modeling servers in queueing networks. In the model in Section 1 (See Figure 3) this type of information can be used to uncover the percentage of time the execution unit spends executing each type of instruction. Another statistic provided by the package is the "throughput" of a transition: the number of times it finished firing divided by the

| PLACE STATISTICS | | |
|---|---|---|
| Place Number (name) | Min/Max Concurent Tokens | Avg Concurent Tokens |
| Full_I_buffers | 0/6 | 4.621 |
| Empty_I_buffers | 0/6 | 0.7576 |
| pre_fetching | 0/1 | 0.3107 |
| fetching | 0/1 | 0.2275 |
| storing | 0/1 | 0.12 |
| Bus_busy | 0/1 | 0.6582 |
| Decoder_ready | 0/1 | 0.0014 |
| Execution_unit | 0/1 | 0.2739 |
| ready_to_issue_instruction | 0/1 | 0.5022 |

Figure 5: Performance statistics report

simulation time. This information can be used to measure processing rate. In our example the sum of the throughputs of all the execution transitions describes the instruction processing rate of the machine.

As discussed above, detailed information about the performance of the modeled system can be easily obtained by carefully interpreting the statistics provided by the P-NUT package. The reports are produced from the **stat** tool in format suitable for processing by standard text processing tools. An abbreviated sample report is shown in Figure 5.

### 4.3 Animation

One of the advantages offered by Petri nets is the graphical representation of the nets. This graphical representation is claimed to be easier to understand than textual representations. We are attempting to evaluate such claims, so the tools we have constructed support the graphical input of nets, and the animation of simulation results. Simulation traces can be processed by an animation tools which allows the user to single-step through the trace of to animate the entire trace. The animation is done carefully so as to give the user enough visual feedback to understand the relationship between events. A common deficiency of Petri Net animations is that the animation consists of tokens disappearing and reappearing from places. The P-NUT animator deliberately animates the flow of tokens over arcs in order to give the user time to understand the effect of state transitions. The enabling and firing of transitions are also animated so as to provide users with some understanding of the causes of various events.

The animation supported by the P-NUT system is better referred to as a visual discrete event simulation. It is not a true animation since there is no constant relationship between real time and simulation time. Since the simulation is inherently a discrete-event simulation, the simulation clock can change dramatically in a brief instant in real-time. One area currently being explored by the P-NUT group is the use of true animation in giving users feedback about bottlenecks in the system.

### 4.4 Timing Analysis and Verification

A great deficiency in current simulation languages and tools is that they provide only the most primitive aids in analyzing the simulation results for correctness and in examining the dynamic behavior of the system being modeled. This weakness results from the fact that simulation models and tools are typically intended to provide performance data. The underlying assumption behind many of the current tools is that errors in simulation models must be uncovered using traditional program-debugging methods or by analyzing the performance data. Unfortunately, neither of these approaches is effective. First, because simulation models mimic concurrency, debugging techniques and tools for sequential programs are inadequate. Secondly, many incorrect simulation models produce performance data which appears on the surface to be quite reasonable. What are needed are timing analysis tools and verification aids. P-NUT supports both through a tool call **tracertool**

The timing analysis supported by **tracertool** is inspired by examining how systems engineers test real computer hardware. If given the task of understanding the timing behavior of a modern micro-processor, a systems engineer is likely to resort to a Logic State Analyzer. Probes are places at relevant inputs to the processor and on important lines on the bus and the resulting timing traces are examined to understand how the resources in question (the bus, for example) are being used. This same concept has been adopted for analyzing timing in Petri Net models. Part of **Tracertool** is nothing more than a software logic state analyzer. A user may select any places or transitions to be plotted over time and may define arbitrary functions (using a simple programming language) on places and transitions. Figure 6 shows an example of a timing analysis of the model described in section 1. The first line shows activity in place Bus-busy. Bus activity caused by instruction pre-fetching and operand fetching is shown in the next two lines. The next line shows a a user-defined function which sums the activities on all the execution transitions, and therefore shows the activity of the execution unit. The final display shows how the number of empty instruction buffer slots varies over time. Markers can be position in the trace to identify critical events. The **tracertool** can then assist the user in timing these events.

The verification supported by **tracertool** is inspired by a similar tool used in analyzing reachability graphs [MR87]. The P-NUT reachability graph analyzer allows users to enter high-level specification of the expected behavior of a system in first-order predicate calculus and in temporal logic. The analyzer then determines if all possible behaviors of the system meet the high level specification. **Tracertool** uses the same concept to "test" (rather than prove) the correctness of a simulation trace. The correct behavior of the system is specified and the tool automatically checks if the particular traces is correct. A detailed description of the capabilities of the reachability graph analyzer (and therefore tracertool) can be found in [MR87]. Below we illustrate some of the questions that could be asked of the behavior of the pipelined processor described above.

In order to use **tracertool** effectively, a deep understanding of the model is required. For example, we discussed earlier that in the model presented in this paper, the sum of the tokens on the places modeling the state of the bus should always be 1. An error in the model (for example a non-zero timing in a transition) may cause a token to be removed from both places at the same time. Checking for such an error can be done by asking:

```
forall s in S [ Bus_busy(s) + Bus_free(s) == 1 ]
```

The variable S implicitly refers to all the states in the simulation trace. This example tests for a bug in the model. Similar tests can also be used to learn new information about the system being

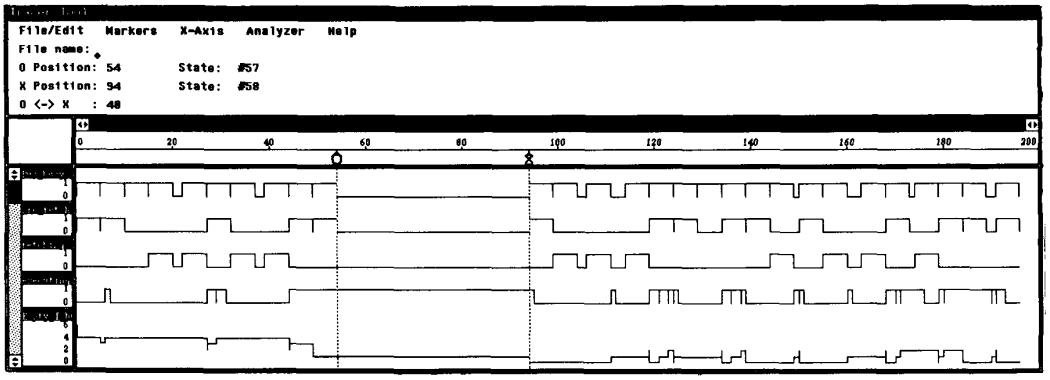


Figure 6: Timing analysis using Tracertool

modeled. Pipelining hazards may be detected by determining if certain places ever received more than one token or if certain transitions were ever fired several times concurrently. Similarly, examining the number of tokens in places can yield interesting information about buffering requirements. In our example we may want to ask if the instruction buffer ever becomes empty after the initial state (where it starts out empty). This question can be formulated as

```
exists s in (S-{#0}) [ Empty_I_buffers(s) == 6 ]
```

In this expression #0 refers to the initial state of the system. It is also possible to ask questions to determine whether particular situations were simulated during a particular run. For example, the question

```
exists s in S [ exec_type_5(s) > 0 ]
```

determines if type 5 instructions (whose execution is modeled by transition **exec_type_5**) were ever encountered during a simulation run. Another particularly powerful feature of the reachability graph analyzer and the tracertool is the use of temporal logic. Using temporal logic it is possible to ask if something eventually becomes true. For example, the test below tests to see if the bus is freed every time it is used. This is not a proof of any kind but it does allow the user to ask if, **during this particular simulation run**, the bus is always freed.

```
forall s in {s' in S | Bus-busy(s')}
          [ inev(s, Bus-free(C), true) ]
```

This expression can be interpreted as: "from every state where the bus is busy, inevitably we reached a state where the bus was free".

## 5 Conclusion

This paper describes a collection of tools which support the construction and analysis of Petri Nets models. These models are particularly well-suited to modeling hardware systems such as pipelined processors. The paper has focused on a subset of the tools which comprise the P-NUT system. Other tools support analytical (as opposed to simulation) performance evaluation, and reachability analysis (for verification). The long term objective of this project is to support the design and evaluation of computer systems through the use of models. The work on new modeling and analysis tools which can apply to both hardware and software continues.

An effort is currently under way to test the usefulness of the analysis methods being developed. Empirical evaluations, using student subjects, has begun and preliminary results should be available soon. The objective of the evaluation is not necessarily to show that some specific tool is useful but to uncover the areas (in terms of uncovering design flaws through models) where certain analysis techniques are particularly effective and where they fail. Such experiments should provide useful insights into fruitful areas of future research.

## References


[Age79] T. Agerwala. Putting Petri nets to work. *IEEE Computer*, 85–94, December 1979.

[Bae80] J-L. Baer. *Computer Systems Architecture.* Compuer Science Press, Potomac, MD, 1980.

[HV85] M. Holliday and M. Vernon. A generalized timed Petri net model for performance analysis of pipelined architectures. In *International Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[Mis73] D. Misunas. Petri nets and speed independent design. *Communications of the ACM*, 16(8):474–481, August 1973.

[MR87] E. Timothy Morgan and Rami R. Razouk. Interactive state-space analysis of concurrent systems. *IEEE Transactions on Software Engineering*, SE-13(10):1080–1091, October 1987.

[Pet81] J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[Ram74] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* Technical Report MAC-TR-120, MIT, 1974.

[RH80] C.V. Ramamoorthy and G.S. Ho. Performance evaluation of asynchronous concurrency systems using Petri nets. *IEEE Transactions on Software Engineering*, SE-6(5):440–449, September 1980.

[RP84] R.R. Razouk and C. Phelps. Performance analysis using timed Petri nets. In Y. Yemini and S. Yemini, editors. *Protocol Specification, Testing, and Verification IV*, North-Holland Pub. Co., 1984.

[WPS86] D.L. Woo, C.V. Phelps, and R.D. Sidwell. Timed Petri net probability semantics. In *Proccedings of the Seventh European Workshop on Application and Theroy of Petri Nets*, pages 131–149, Oxford, England, June 1986.

[Zub80] W.M. Zuberek. Timed Petri nets and preliminary performance evaluation. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 88–96, 1980.