Innovative Systems Design and Engineering
ISSN 2222-1727 (Paper) ISSN 2222-2871 (Online)
Vol 2, No 5, 2011

www.iiste.org

Modelling of a Sequential Low-level Language Program Using Petri Nets

Ganiyu Rafiu Adesina (Corresponding author)
Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, P.M.B 4000,Ogbomoso, Nigeria.
Tel: +2348060596393 E-mail: ganiyurafiu@yahoo.com

Olabiyisi Stephen Olatunde
Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, P.M.B 4000,Ogbomoso, Nigeria.
Tel: +2348036669863 E-mail: tundeolabiyisi@hotmail.com

Omidiora Elijah Olusayo
Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, P.M.B 4000,Ogbomoso, Nigeria.
Tel: +2348030712446 E-mail: omidiorasayo@yahoo.co.uk

Arulogun Oladiran Tayo
Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, P.M.B 4000,Ogbomoso, Nigeria.
Tel: +2348033643606 E-mail: arulogundiran@yahoo.com

Okediran Oladotun Olusola
Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, P.M.B 4000,Ogbomoso, Nigeria.
Tel: +2348034466236 E-mail: dotunokediran@yahoo.com

Abstract

Petri nets were devised for use in the modelling of a specific class of problems. Typical situations that can be modelled by Petri nets are synchronization, sequentiality, concurrency and conflict. This paper focuses on a low-level language program representation by means of Petri nets. In particular, Petri net formalisms were explored with emphasis on the application of the methodology in the modelling of a sequential low-level language program using a Motorola MC68000 assembly language program as an example. In the Petri net representation of the sequential low-level language program under consideration, tokens denote the values of immediate data as well as availability of the data. Thus, the developed petri net model shows that Petri net formalism can be conveniently used to represent flows of control and not flows of data.

Keywords: Petri nets, model, low-level language, microprocessor, instructions.

1. Introduction

Computer program is a sequence of instructions written in a defined computer language given to a computer to have a problem solved. However, in solving a computational problem, microcomputer can be programmed using binary or hexadecimal number (machine language), Semi-English language statements (low-level language) or a more understandable human-oriented language called high-level language. Low-level language is a mnemonic representation of a natural or native language of a computer called machine code. The programmer finds it relatively more convenient to write the programs in assembly language than in machine language. Nevertheless, a translator called an *assembler* must be used to convert the low-level language programs into binary machine language programs (objects codes) for the microprocessor to execute (Arulogun *et al.*, 2005). In general, a low-level language instruction consists of the following fields:

- Label field
- Mnemonic (Op-code) field
- Operand field
- Comment field (optional)

In the same vein, a Petri net is an abstract, formal model of information flow. The properties, concepts, and techniques of Petri nets are being developed in a search for natural, simple, and powerful methods for describing and analyzing the flow of information and control in systems, particularly systems that may exhibit asynchronous and concurrent activities. The major use of Petri nets has been the modelling of systems of events in which it is possible for some events to occur concurrently but there are constraints on the concurrence, precedence, or frequency of these occurrences (Peterson, 1977). Petri nets were devised for use in the modelling of a specific class of problems. Typical situations that can be modelled by PN are synchronization, sequentiality, concurrency and conflict (Bobbio, 1990). Practically speaking, the Petri net represents the possible task execution sequence and it is similar to a task graph (Abdeddaim *et al.*, 2003; Saldhana *et al.*, 2001). The Petri net is both a visual and formal executable specification that is easy to understand (Staines, 2008). In particular, the Petri net graph models the static properties of a system, much as a flowchart represents the static properties of a computer program. Thus, in view of the foregoing, this paper explores the modelling of sequential low-level language programs using Petri Nets.

2. Methodology

2.1 Basic Petri Net Notions

Mathematically, a Petri net (PN) is defined as a 5-tuple, $PN = (P,T,A,W,M_0)$ where:

$P = p_1, p_2,..., p_m$ is a finite set of places,

$T = t_1, t_2,..., t_n$ is a finite set of transitions,

$A \subseteq (P \times T) \bigcup (T \times P)$ is a set of arcs,

$W : A \rightarrow 1,2,3...$ is a weight function,

$M_0 : P \rightarrow 0,1,2,3...$ is the initial marking,

$P \bigcap T = \phi$ and $P \bigcup T \neq \phi$. (Murata, 1989)

Graphically, a PN consists of two types of nodes, called “places” (P) and “transitions” (T). Arcs (A) are either from a place to a transition $(P \times T)$ or from a transition to a place $(T \times P)$. Places are drawn as circles. Transitions are drawn as bars or boxes. Arcs are labelled with their weights (W), which take on positive integer values. The class of nets where we allow arc weightings greater than 1 are known as generalized Petri nets. When arc weightings are 1, the class is known as ordinary PNs. The ordinary PN is considered to be the common language linking various versions of PNs. Figure 2.1 depicts a typical Petri net (PN) while Table 2.1 gives a few possible interpretations of the places and transitions.

The marking at a certain time defines the state of the PN. The evolution of the state corresponds to an evolution of the marking, which is caused by the firing of transitions (David and Alla, 1994). A marking is denoted by *M*, an $m \times 1$ vector, where *m* is the total number of places. The p^{th} component of *M*, denoted by

M(p), is the number of tokens in the p^{th} place. The initial marking for the system represents the initial condition of the system and is denoted as M_0. The state of the PN evolves from an initial marking according to transition (firing) rule. In an ordinary Petri net, if all the places that are inputs to a transition have at least one token, then the transition is said to be enabled and it may fire. When an enabled transition fires, a token is removed from each of the input places and a token is placed in each of the output places.

Figure 2.2 gives an example of firing a Petri net. The initial marking is $M_0 = (1\ 1\ 0\ 1\ 0)^T$ as shown in Figure 2.2a. With a default arc weighting of one, transition t_1 is enabled by the tokens in its upstream places p_1 and p_2. The transition t_1 then fires, resulting in removal of one token from each of the places p_1 and p_2, and addition of one token into the place p_3 as shown in Figure 2.2b. The marking evolves to $M_1 = (0\ 0\ 1\ 1\ 0)^T$ after the firing of transition t_1. The tokens in places p_3 and p_4 then enable transition t_2, the firing of which results in a marking of $M_2 = (0\ 0\ 0\ 0\ 1)^T$, as shown in Figure 2.2c. Note that the number of the tokens is not necessarily conserved in a PN model. There are several behavioral properties of PNs (Murata, 1989; Lu, 2002). These include reachability, boundedness, liveness and reversibility.

2.1.1 Reachability

Reachability is a fundamental basis for studying the dynamic properties of any system. A marking M_n is said to be reachable from a marking M_0 if there exists a sequence of firings that transforms M_0 to M_n. The set of markings reachable from M_0 is denoted by $R(M_0)$.

2.1.2 Boundedness

A Petri net (P,T,A,W,M_0) is said to be *k*-bounded or simply bounded if the number of tokens in each place does not exceed a finite number *k* for any marking reachable from M_0, i.e. $k \geq M(p)$ for every place *p* and every marking $M \in R(M_0)$. A Petri net (P,T,A,W,M_0) is said to be safe if it is 1-bounded. By verifying that the net is bounded or safe, it is guaranteed that there will be no overflows in the buffers or registers, no matter what firing sequence is taken, and that the number of tokens in a place will not become unbounded.

2.1.3 Liveness

The concept of liveness is closely related to the complete absence of deadlocks in operating systems. A Petri net (P,T,A,W,M_0) is said to be live if no matter what marking has been reached from M_0, it is possible to ultimately fire any transition in the net by progressing through some further firing sequences. This means that a live Petri net guarantees deadlock-free operation, no matter what firing sequence is chosen.

2.1.4 Reversibility

A Petri net (P,T,A,W,M_0) is said to be reversible if, for every possible marking reachable from M_0, M_0 is reachable from it. Thus, in a reversible net one can always get back to the initial marking or state.

2.2 Low-level Language Programming

The most primitive language in which programs are written in native or host language of a computer is called low-level language. It uses mnemonic to represent various operations performed by the computer. Mnemonics are self-evident symbolic name that refers to an operation. For examples:

ADD denotes addition operation

SUB denotes subtraction operation

BRA denotes branching operation

MOVE denotes copy operation

By considering a typical low-level language instruction (*SUB.B Number*, *D3).* This instruction means subtract 8-bit number stored in memory location named *Number* from the contents of a data register *D3*. The *.B* following mnemonic indicates size of source data that the instruction will work on i.e. 8-bits. The data register *D3* referred to in the instruction is a special purpose data storage element within the microprocessor and *Number* refers to memory location of the source operand. Furthermore, an instruction in operation code field manipulates stored data and a sequence of instruction makes up a program. That is, the *OP-code* field specifies how data is to be manipulated. A data item may reside within a microprocessor register or in main memory. Thus, the purpose of the *Address field* is to indicate the location of a data item. For example, let us consider the low-level language instruction given below:

$$\underbrace{\text{ADD}}_{\text{Op-code}} \quad \underbrace{\text{R1, R0}}_{\text{Address field}}$$

Assume that the microcomputer under consideration uses RI as the source register and R0 as destination register. The Op-code i.e. the ADD part of the instruction means arithmetic addition operation. Therefore the instruction will add the contents of microprocessor register R1 to R0 and save the sum in register R0.

i.e. $[Destination\,\mathrm{Re}gister] \leftarrow [Destination\,\mathrm{Re}gister] + [Source\,\mathrm{Re}gister]$

2.2.1 Low-level Language Instruction Formats

The following instruction formats are identifiable in low-level language based upon the number of addresses specified in the instruction (Arulgun *et al.*, 2005):

- Zero-address instruction format
- One-address instruction format
- Two- address instruction format
- Three-address instruction format

Zero-address instruction format: An instruction that does not require any address is called a zero-address instruction format. Examples are STC (set carry flag), NOP (no operation), RAL (rotate accumulator left) and RET (return from exception).
One-address instruction format: An instruction with a single address is called one-address instruction format. It takes the following format:
<Op-code> Address1
e.g. ADD B ; $[Accumulator] \leftarrow [Accumulator] + [B]$
Two-address instruction format: An instruction containing two addresses is called two-address instruction format. It takes the following format:
<Op-code>Address1, Address2
e.g. MOVE R2, R1 ; $[R1] \leftarrow [R2]$

Three-address instruction format: An instruction with three addresses is called three-address instruction. It takes the following format:
<Op-code>Address1, Address2, Address3
e.g. MUL A, B, C ; $[C] \leftarrow [A]*[B]$

2.2.2 MC68000 Microprocessor Programming Instructions

The number and types of instructions supported by a microprocessor may vary from one microprocessor to another and primarily depends on the architecture of a particular machine. In writing low-level language programs, unlike high level language where compiler performs data allocation to registers automatically, programmer must decide what goes into any of the data registers and memory; address to a distinct address register; the type of data and address acquisition by the microprocessor for each of the program microinstructions. Programming in low-level language requires in-depth understanding of a particular microprocessor instruction set and its architecture. As a result, in this paper, MC68000 processor is chosen to explore the modelling of sequential low-level language programs using Petri Nets.

MC68000 microprocessor is the Motorola’s first 16-32bit-microprocessor chip. That is, it has 16-bit data path and capable of 32-bit internal operations. Other members of the former series are improved versions of MC68000 microprocessor, with many features added along the way. Its address and data register are all 32-bit wide. MC68000 supports five different data types. They are 1-bit, 4-bit BCD digits, 8-bits (byte), 16-bits (word), and 32-bits (long word). Its instruction set includes 56 basic instruction types, 14 addressing modes, and over 1000 *Op-codes*. It executes the fastest and slowest instructions at 500ns (i.e. the one that copies contents of one register into another register). It has no *input and output instruction*, hence, all input and output are *memory mapped*. The MC68000 is a general-purpose register microprocessor with many data registers which can be used either as an *"accumulator*" or as "scratchpad *register*". It has eight data registers (i.e. D0-D7) and nine address registers including the supervisor stack pointer (i.e. A0-A6; A7, A7ʹ). Any data or address register can be used as an index register for addressing purpose. Although, it has 32-bit internal registers; only the low-order 24 bits are used. It's also a byte addressable processor and can address up to 16MB of memory locations (Arulogun *et al.*, 2008).

In furtherance, MC68000 instruction set repertoire is very versatile and allows an efficient means to handle high-level language structures like linked lists and array. The notation 'B', 'W', 'L' is placed after each MC68000 mnemonic to depict the operand size whether it is byte, word or long word. All MC68000 instructions may be classified into eight groups as follows:

- Data Movement Instructions
- Arithmetic Instructions
- Logical Instructions
- Shift and Rotate Instructions
- Bit Manipulation Instructions
- Binary Coded Decimal Instructions
- Program Control Instructions
- System Control Instruction

2.2.3 Sequential Low-level Language Programs

In a conventional microcomputer, instructions are always executed in the same order (sequence) in which they are presented to the computer, irrespective of the programming language being employed. In this situation, a program can select a particular sequence of instructions to execute based on the results of computation. In a low-level language programming, the instructions that could be used to realize this idea are called *program control instructions* (i.e. Unconditional Branch Instruction, Conditional Branch Instruction, Subroutine Call and Return Instruction). Thus, low-level language programs without program control instructions are called *sequential low-level language programs*. They are extensively used to program simple arithmetic operations that do not require iteration or branching. To explore the petri net modelling of a sequential low-level language program, Figure 2.3 depicts the MC68000 program under consideration. The low-level program is characterized by nothing but sequential instructions. Figure 2.4 shows the developed petri net model of the MC68000-based sequential low-level language program depicted in Figure 2.3.

3. Conclusions and Future Work

In this paper, we have been able to develop a Petri net model for a low-level language program. Precisely, Petri net formalisms were explored with emphasis on the application of the methodology in the modelling of an MC68000-based sequential low-level language program. In the Petri net representation of the sequential low-level language program under consideration, tokens denote the values of immediate data as well as availability of the data. Thus, the developed petri net model depicts that Petri net formalism can be conveniently used to represent flows of control and not flows of data. Nevertheless, analysis of the developed Petri net model could be carried out using reachability tree method in a bid to gain insights into the behavioural properties of the modelled phenomenon. Besides, future research may be geared towards developing Petri net models for low-level language programs of a named microprocessor, which are characterized by program control instructions such as unconditional branch instruction, conditional branch instruction, subroutine call and return instruction.

References

Abdeddaim, Y., Kerbaa, A. and Maler, O. (2003). Task Graph Scheduling using Timed Automata. *IEEE Parallel and Distributed Processing Symposium*.

Arulogun, O. T., Fakolujo, O. A., Omidiora, E. O. and Ajayi, A. O. (2005). *Assembly Language Programming Using MC68000.* Johnny Printing Works, Ogbomoso, Nigeria, (Chapter 2).

Arulogun, O. T., Fakolujo, O. A., Omidiora, E. O. and Ganiyu, R. A. (2008). *Introduction to Microprocessor System*, Johnny Printing Works, Ogbomoso, Nigeria, (Chapter 2).

Bobbio, A. (1990). System Modelling with Petri Nets. A.G. Colombo and A. Saiz de Bustamante (eds.), *System Reliability Assessment*, Kluwer p.c., 102-143.

David, R. and Alla, H. (1994). Petri Nets for Modeling of Dynamic Systems - A Survey. *Automatica*, 30, 175-205.

Desrochers, A. A. (1992). Performance Analysis Using Petri Nets. *Journal of Intelligent and Robotics Systems*, 6, 65-79.

Lu, N. (2002). *Power System Modelling Using Petri Nets.* PhD thesis, Rensselaer Polytechnic Institute, Troy, New York.

Murata, T. (1989). Petri nets: properties, analysis and application. *Proceedings of the IEEE*, 77, 4, 541-580.

Peterson, J. L. (1977). Petri nets. *Computing Surveys*, 9, 223‑252

Saldhana, J. A., Shatz, S. M. and Hu, Z. (2001). Formalization of Object Behavior and Interactions From UML Models. *International Journal of Software Engineering and Knowledge Engineering IJSEKE*, 11, 6, 643-673.

Staines, A. S. (2008). Modeling and Analysis of a Cruise Control System. *World Academy of Science, Engineering and Technology* 38, 173-177.

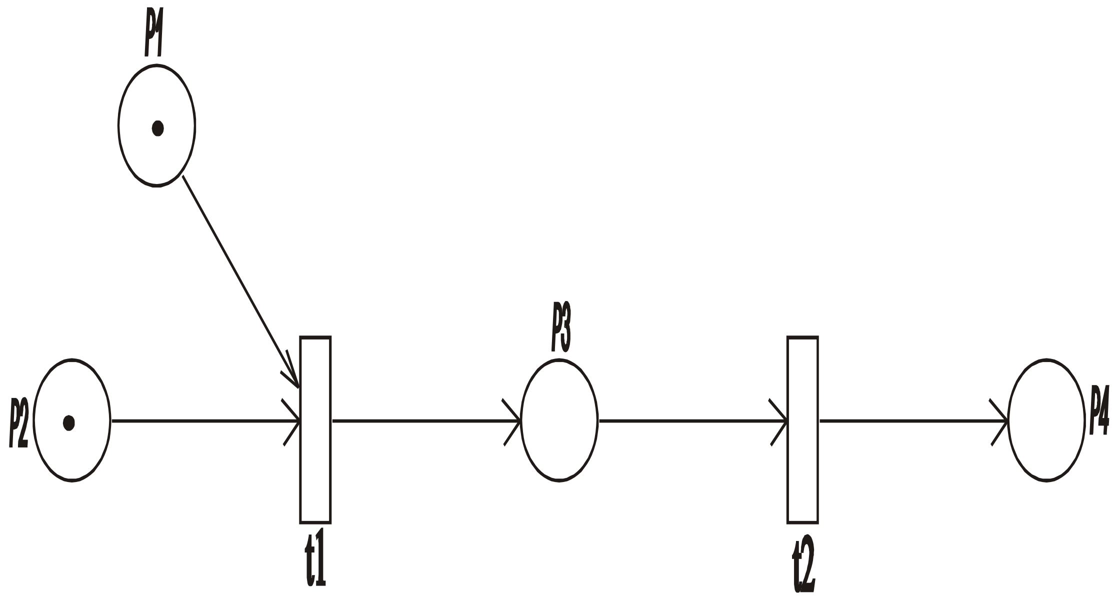

Figure 2.1. A typical Petri net example

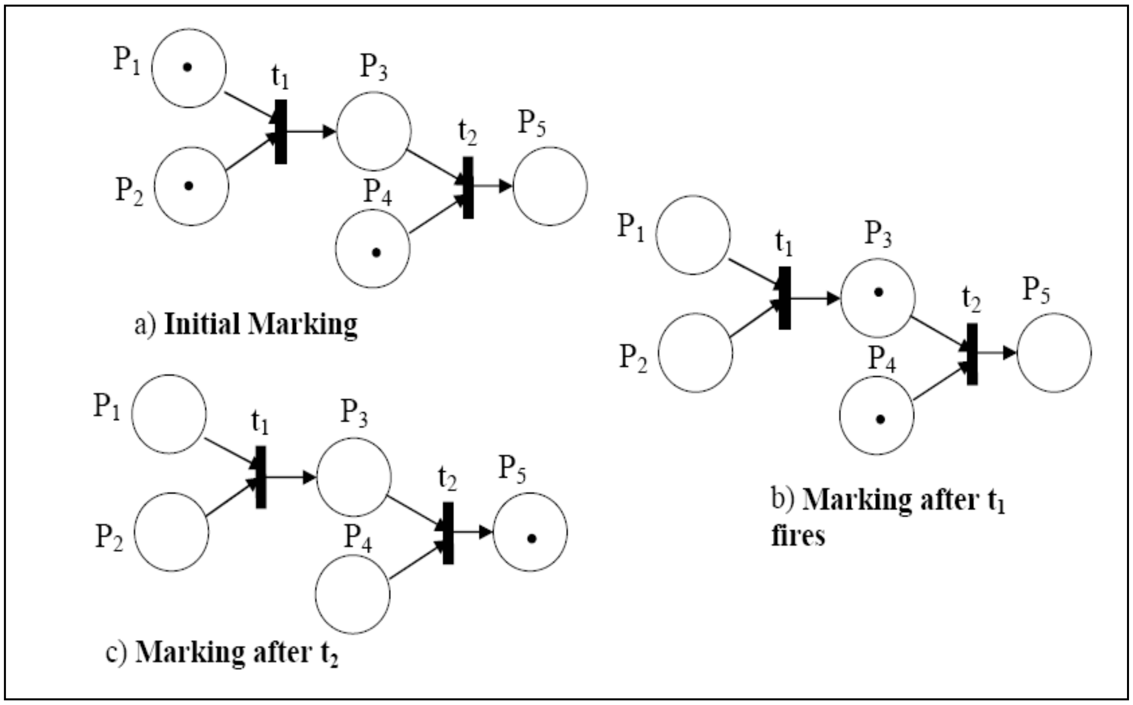

Figure 2.2. Firing of a Petri net (Desrochers, 1992)

```
MOVE #P, D0        ; move data P into D0
MOVE #Q, D1        ; move data Q into D1
SUB D1, D0         ; D0 ← P − Q
ADD #P, D1         ; D1 ← Q + P
DIVU D1, D0        ; D0 ← (P − Q)/(P + Q)
MOVE D0, (A0)      ; (A0) ← D0
```

Figure 2.3. The sequential sample program written in MC68000 low-level language

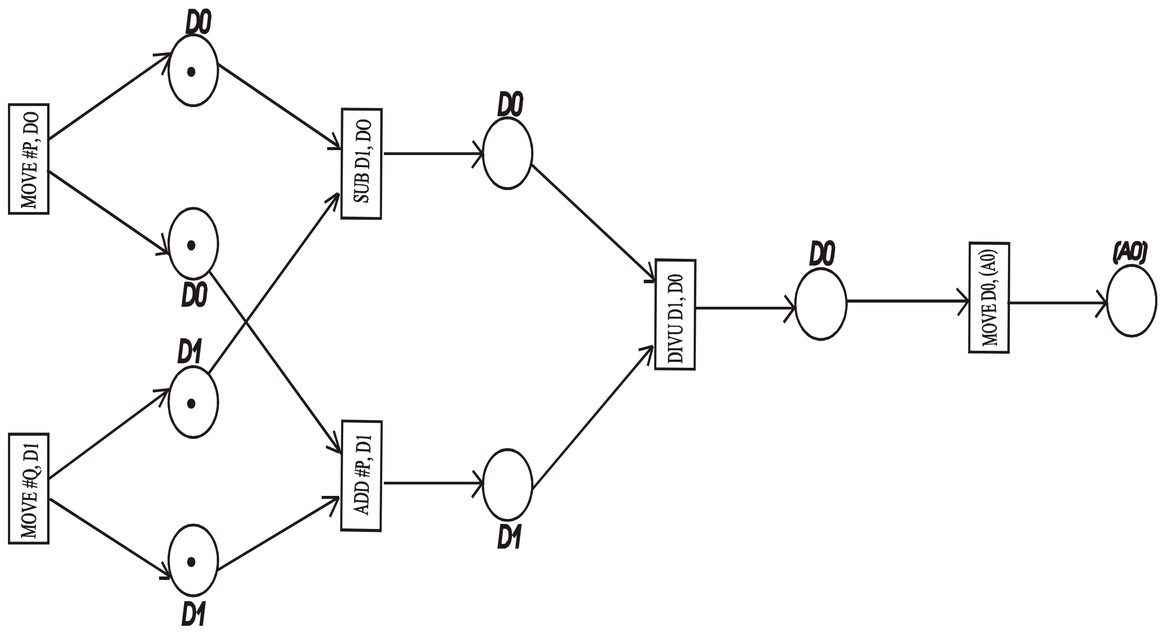

Figure 2.4. The developed petri net model of the MC68000-based sequential low-level language program shown in Figure 2.3

Table 2.1. Some Typical Interpretations of Transitions and Places (Murata, 1989)

Input Places	**Transitions**	**Ouput Places**
Preconditions	Event	Postconditions
Input data	Computation step	Output data
Input signals	Signal processor	Output signals
Resources needed	Task or job	Resources released
Conditions	Clause in logic	Conclusion(s)
Buffers	Processor	Buffers