

UC Berkeley Computer Science

Subject: Memory Read and Memory Write
Date: October 20, 2006
From: Ivan Sutherland
UCIES #2006-is39

References:

UCIES# 2006-is30: FLEET – A One-Instruction Computer, Ivan Sutherland, 24 August 2006
UCIES# 2006-is31: Some SHIPs, Adam Megacz and Ivan Sutherland, 24 August 2006
UCIES# 2006-is37: A Description of a Fetch SHIP, Ivan Sutherland, 10 October 2006
UCIES# 2006-is38: Description of a Fetch SHIP (slides), Ivan Sutherland, 18 October 2006

ABSTRACT

To describe the behavior of a SHIP with a Petri net, we represent inputs and outputs as a single event each. Treating inputs and outputs as a single event each omits the details of the handshake signals between the switch fabric and the SHIP. One can think of such a single input event as the earliest that the SHIP will consider a fresh input value. One can think of such a single output event as the earliest that the SHIP will produce a fresh output value. Appropriate behavior of the switch fabric is implied.

Such a view implies a sequence of events between the SHIP and the switch fabric. The SHIP cannot consider a fresh input until the switch fabric delivers it, and the switch fabric cannot deliver a fresh input value until the SHIP has acknowledged the previous input value. Similarly, the SHIP cannot produce a new output until the switch fabric acknowledges receipt of the previous output, and the switch fabric cannot even know about a new value until the SHIP announces that the new value is present.

Thus treatment of inputs and outputs as single events implies a cycle of three events at each input or output terminal. Two of these events are the “real” handshake signals between the switch fabric and the SHIP, and the third is an artificial event that appears only in the Petri net specification of the SHIP. Because the specification of the SHIP remains silent on the relationship between the handshake signals, such a specification for the SHIP’s behavior leaves ample room for designers to include or omit input or output storage registers. Programmers concerned only with moving data from one SHIP to another have little need to consider the details of the internal storage of a SHIP because the SHIP’s internal storage is only a very small part of the pipeline storage provided by the switch fabric itself.

PURPOSE

I offer this memo in response to the questions raised in class on 18 October about the precise meaning of the input and output events in a Petri net describing the behavior of a SHIP. I write this memo not only to present the possible Petri nets for the Memory Read SHIP and the Memory Write SHIP that we discussed in class but also to offer my understanding of the precise meaning of the events that appear in them. As often happens to me, wrapping English words around these ideas has forced me to think more deeply about them.

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley.. The ideas contained herein are freely available for any academic purpose.

INTRODUCTION

NASA might describe the schedule for a rocket launch as “no earlier than 10 AM on October 28.” While setting a target time for the launch, such a one-sided specification leaves room for possible delay. An event in a Petri net is like that; the specified conditions must all be met before the event can happen, but there may be other reasons for delay. Petri nets are permissive rather than compulsory. When all of the places that lead into an event are occupied, i.e. “have a dot” as we have chosen to say, the event may happen, or “fire” as we say. The Petri net, however, offers no compulsion that the event fire promptly or even at all. The event may have to wait for other factors not represented in the Petri net, just as NASA’s launch may have to wait for other conditions such as the weather.

The problem I discuss here is how to represent the essential behavior of a SHIP completely enough to serve the programmer who controls the movement of data between SHIPs but simply enough for the programmer to understand. I want to omit extraneous detail. The hard part, of course, is to identify which details are extraneous and thus OK to omit, and which are important enough to report.

In fact, there are two events associated with each communication between the SHIP and the switch fabric. This is true both at each input of the SHIP and at each output of the SHIP. However, I want to represent the behavior of the SHIP with only a single event for each of its inputs and only a single event for each of its outputs. I feel, intuitively, that a single event at each input and output should suffice to represent the “essence” of the SHIP’s behavior. But to be clear, I must offer a precise meaning for the input and output events that I choose to represent.

A PROPOSED MEANING FOR THE EVENTS

Recall that the Petri nets in this memo describe the behavior of a SHIP. They may also guide good behavior on the part of the programmer, but they don’t attempt to specify what a programmer must do or avoid, nor do they attempt to describe what programmers will or won’t do. These Petri nets describe only what a SHIP may do, or perhaps more important, they proscribe behaviors that the SHIP won’t do. If the programmer offers input values prematurely, the SHIP is free to ignore them, forcing the new values to queue up in the switch fabric until the SHIP is ready for them. If the programmer sends a MOVE instruction to a SHIP’s output before the SHIP puts a value there, the MOVE instruction must wait for the SHIP to produce the output value.

In the Petri nets included in this memo, an input event to a SHIP, a red bar in the figures, represents the earliest moment that the SHIP will consider a new input value. It means that until all the places leading to the red bar contain a dot the SHIP will avoid considering an input value, even if one is offered. Similarly, an output event from a SHIP, a green bar in the figures of this memo, represents the earliest moment that the SHIP will deliver an output value. Until all the places leading to the green bar contain a dot, the SHIP will not offer a new output value.

These earliest moments are permissive, not compulsory. The programmer may be aware that the SHIP may or may not wait until a handshake with the switch fabric is complete. Because such a handshake moves the input to or from a buffer in the switch fabric, the details should be of little importance to the programmer.

The SHIP may also wait for other events not explicitly represented in the Petri net. Such events could include the completion of an external memory cycle, or anything else. For example, a memory control SHIP might avoid starting a cycle if the memory

temperature is too high, or if the memory is otherwise occupied with backup or is serving another memory control SHIP. It might even wait for the passage of time to limit the data rate to or from the memory.

Consider the arcs that lead from the bottom of Figure 2a to the top. Such arcs are part of the description of the behavior of the SHIP, rather than a description of proper programmer behavior. They indicate that the SHIP won't consider a fresh input value until after it has produced the output value associated with a previous input value. Such an arc makes no guarantee that the SHIP will consider a fresh input value immediately after producing an output value. The SHIP may delay considering the new input value until other events take place, events that don't appear in the Petri net. For example, it may wait for the switch fabric to take responsibility for the output value just produced before considering new input value, or it may be willing to consider a new input value while still presenting the former output value to the switch fabric. The Petri net doesn't specify which.

A TRIVIAL SHIP – Figure 1

Consider the bit swap SHIP of Figure 1a. It has only one input and only one output. It swaps the order of the bits in a word, placing the most significant bit in the least significant bit position and so on. It can be made entirely of wire; it requires no logic. A Petri net describing the behavior of this SHIP appears in Figure 1c. What this Petri net says is that the SHIP will consider a new input value initially and then again only after it has proffered the processed version of the former input at its output. This Petri net is silent about the behavior of the handshakes at the input and output.

Embedding this SHIP in the switch fabric, as shown in Figure 1b, implies two more conditions, one at the input and one at the output, as shown in Figure 1d. The input condition states that INreq, IN, and INack must cycle, and the output condition states that OUT, OUTreq, and OUTack must also cycle. These are the conditions imposed by the switch fabric's asynchronous protocol. They are present for every SHIP by virtue of its input and output connections to the switch fabric.

The first of these two additional conditions says that the SHIP won't consider a new input value until after the switch fabric presents the new input value; moreover, the switch fabric can't present a fresh input value until the SHIP has acknowledged the previous input value. The second additional condition says that the SHIP won't produce a new output value until after the switch fabric has acknowledged receipt of the previous output value. That's all pretty obvious.

What's more important is what Figure 1d doesn't specify. In particular, it says nothing about relationship between OUTack and INack. Therefore, the Petri net of Figure 1d permits many different implementations of the bit swap SHIP. One possible implementation contains no internal storage. The implementation without internal storage avoids releasing its input until after its output has departed. In other words, it adds the condition shown in Figure 1e, forcing the sequence INreq, IN, act, OUT, OUTreq, OUTack, INack.

Another implementation might have two internal registers, one at the input to swap wires and one at the output of its swap wires. Having produced a first output, it can not only consider a new input but also release the switch fabric at its input port.

Does the programmer need to know all this detail? I assert not. The programmer need only know that for every input value the program sends to the bit swap SHIP, the SHIP will produce a swapped value. That's what the Petri nets say.

One might think that without at least one stage of storage inside the SHIP, programs would have to avoid MOVE instructions that send its output directly back to its input. But the switch fabric itself has many stages storage. Even a bit swap SHIP without any storage will operate correctly with such a MOVE instruction from output directly back to input. Because the SHIP has no storage, it forces the input value to linger in the switch fabric at the input until after the switch fabric takes responsibility for the output value. Only after the Nth output value is safely launched into the switch fabric will such a storage-less bit swap release the Nth input value. One or two stages of storage more or less in the bit swap SHIP itself make no difference to correct operation of the program.

MY TARGET

I want the programmer to think of a single event at the input of a SHIP and to think of a single event at the output of a SHIP. These input and output events are simplifications of the twin events of the switch fabric's handshakes. Think of such input events as the soonest that the SHIP can consider a fresh input and the soonest that the SHIP can offer a fresh output.

Although we don't explicitly represent the rest of the input handshake, we can be sure that the SHIP won't consider a fresh input until it has acknowledged receipt of the previous input from the switch fabric. Indeed, the switch fabric won't present the new input until after the SHIP acknowledges the previous one. Similarly, although we don't explicitly represent the rest of the output handshake, we can be sure that the SHIP won't produce a fresh output until the switch fabric has taken full responsibility for delivery of the previous one. The input and output handshake events are included in every description of a SHIP by virtue of its connection to the switch fabric.

MEMORY WRITE SHIP – Figure 2

The Memory Write SHIP does "jobs" described by a set of three inputs: an address, a stride, and a count. It writes successive values in memory starting at "address" and separated by "stride", and it asks for a new job when it has written "count" values. For each write operation it requires a data value to write and returns a token when it is ready to receive a new data value. When the entire job is done, the SHIP produces an output called J which has the value of the next address that would have been written had the count been one greater.

The Petri net of Figure 2a specifies that the first write operation will occur only after the SHIP has been given a data value, an address, a stride, and a count. Thereafter, it specifies that each additional write requires a new data value and will produce a fresh token. After the last write has been done and the count has run out, the write SHIP will produce the output called J.

The Petri nets of Figure 2 are silent on when the Memory Write SHIP releases the switch fabric to deliver the next data value. Just as we saw in Figure 1, various implementations of the Memory Write SHIP might offer different behaviors if the programmer fails to provide a MOVE instruction to deliver the token output, N. One implementation might refuse to acknowledge the old data value and consider a new data value until after the previous token is safely on its way. Another implementation might accept the new data value and acknowledge its receipt at the input, but avoid writing it into memory until the output token is safely on its way. A third implementation might accept, acknowledge, and write the new data value while still offering the hung token at the output. The Petri net of Figure 2 applies equally well to all of these implementations.

What happens if the programmer fails to provide a MOVE instruction for the output called J? Having finished a first job and produced an output J, the Petri net says that the SHIP may consider the values for the next job. The Petri net says nothing about whether or not the SHIP actually does accept the next set of parameters. Nor does the Petri net say whether or not the SHIP will begin the next job right away or later. Indeed, the specification permits an implementation that will complete two jobs before the program takes away the J value from the first job!

If we wish further to limit the range of permitted behavior, we will have to expand the specification. For example, we might wish explicitly to specify that by the green bars we mean not only that the SHIP produces an output, but also that the output is safely in the hands of the switch fabric. Because there are two green bars in Figure 2a, such a statement would limit the SHIP in two ways. First, the SHIP would consider a fresh data value to write only after the MOVE instruction associated with the N token output executes. Second, the SHIP would consider a fresh set of parameters only after a MOVE instruction associated with the J value executes. I suspect, however, that we will need more experience to know whether this kind of subtitle distinction really matters.

Figure 2b specifies slightly different behavior for the Memory Write SHIP. It specifies that the SHIP will offer the J output before beginning to write data. It permits, but does not require, the SHIP to accept a new set of job parameters before completing the previous job. Remember that the red bars at the top of the picture are permissive and not compulsive.

I find remarkable that the specification of Figure 2b can be more restrictive than the specification of Figure 2a. Consider what might happen if the programmer fails to provide a MOVE instruction to take away the J output. The specification of Figure 2a, as we have seen, permits an implementation that will complete two jobs. The specification of Figure 2b, on the other hand, requires that each job begin only after the SHIP produces a J output. Because the SHIP can't produce a second J output until the first one has been taken away by a MOVE instruction, the second job can't start until such a MOVE instruction has executed.

MEMORY READ SHIP – Figures 3 and 4

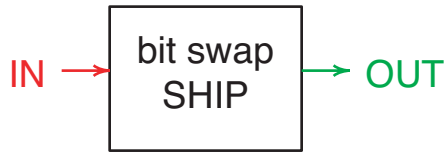
The Memory Read ship shown in Figure 3 is much like the Memory Write SHIP. It accepts "jobs" described by a starting address, a stride, and a count. Thereafter, for each token given to it on its T input it delivers a corresponding data value on its next data output, D. Figure 3a offers a simple Petri net. Figure 3b offers a Petri net with two concurrent processes.

The Petri nets of Figures 2 and 3 are remarkably similar. In fact, I copied one to make the other, substituting only the names of the data and token inputs and outputs. Thus all of the comments about the range of behavior permitted to the Memory Write SHIP apply equally to the Memory Read SHIP.

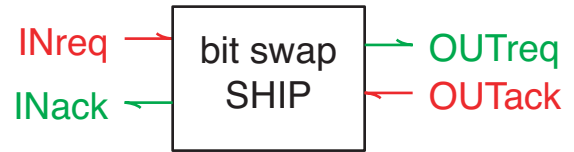
But Memory Read is fundamentally different from Memory Write. In order to write in memory one must have the data to write. In order to read, one need not have the token that permits output of the value read. The Petri nets of Figure 3 fail to make this distinction. Figure 4 offers a slightly different specification for the Memory Read SHIP. Here we see that input of the token, T, is concurrent with the read operation.

How might one test a Memory Read SHIP to detect this difference in specification? On the very last read of a job, Figure 3a requires input of the token, T, prior

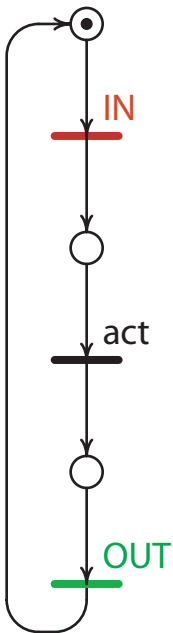
to producing the job request J, but Figure 4a does not. Thus we could withhold the last token and see if the SHIP still produces an output at J. If it does, we can rule out Figure 3a. However, if it does not, we have learned nothing. For one thing, how long do we wait before announcing the failure of J to appear? Ten seconds, ten minutes, ten hours, or ten years? We don't know what additional constraints have been imposed on Figure 4a to prevent appearance of the J output. Figure 4a specifies some permitted behaviors, not required behaviors. Thus the distinction between the Petri net of Figure 3a and Figure 4a is at best one-sided.



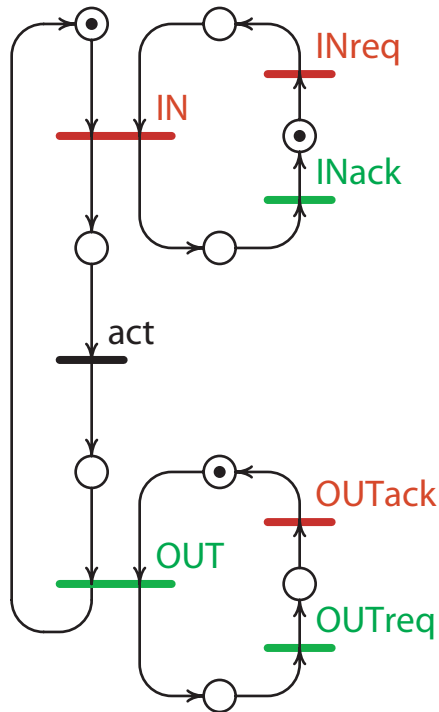
a) Block diagram



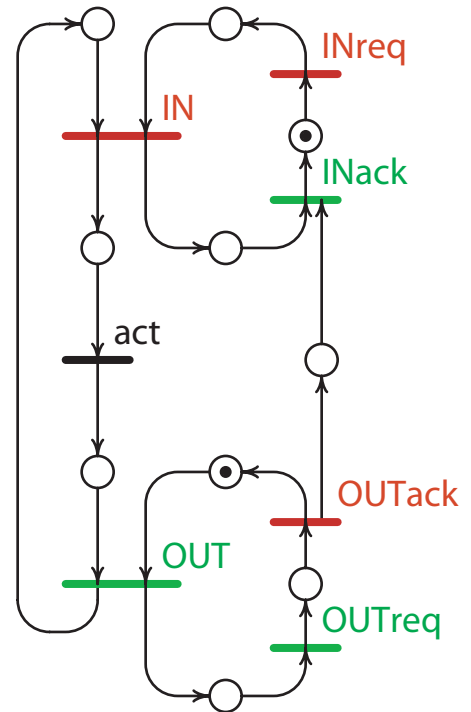
b) Signal diagram



c) simple view

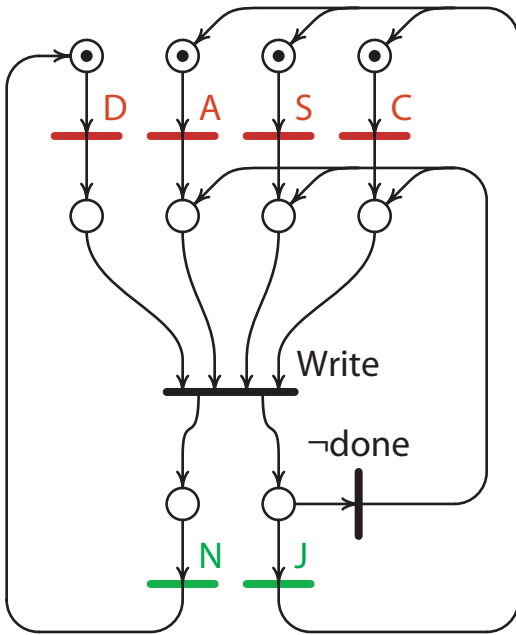


d) with handshakes

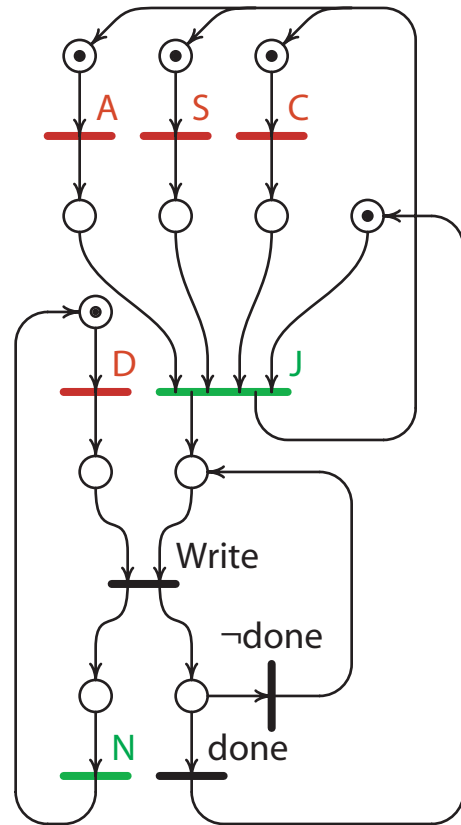


e) with no registers

Figure 1: Simple bit swap SHIP



a) without input buffer



b) with input buffer

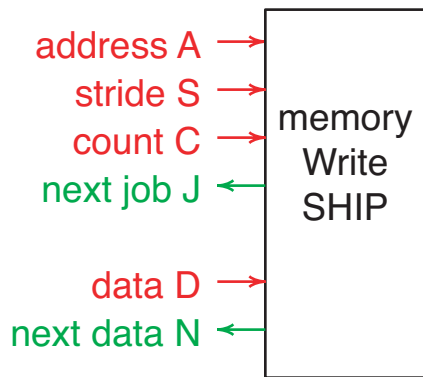
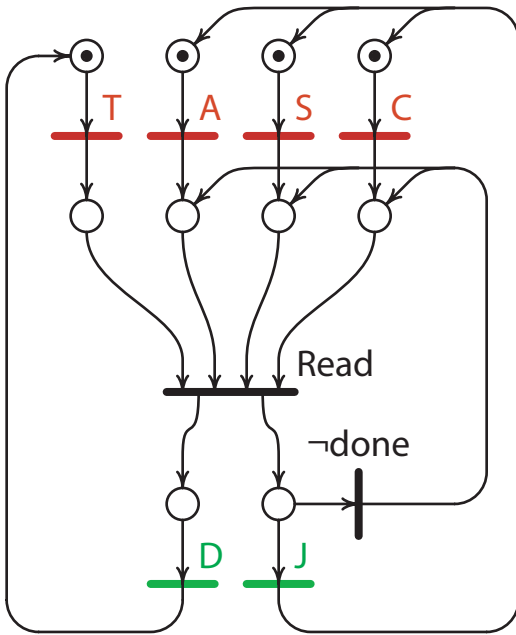
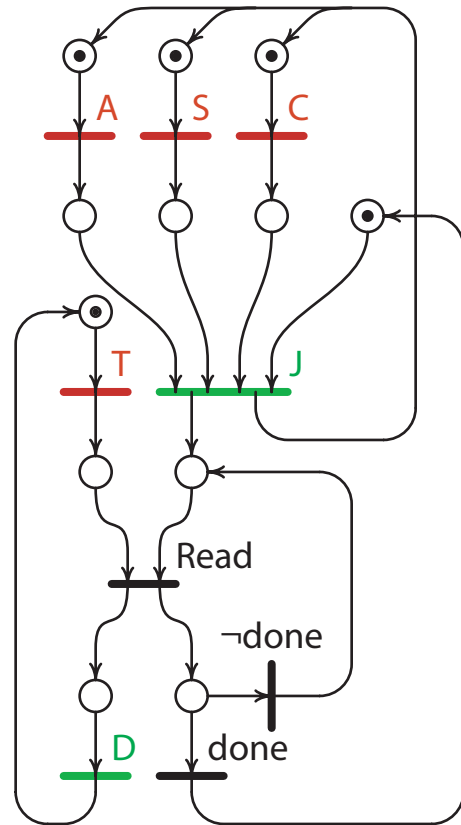


Figure 2: Memory Write Petri Nets



a) without input buffer



b) with input buffer

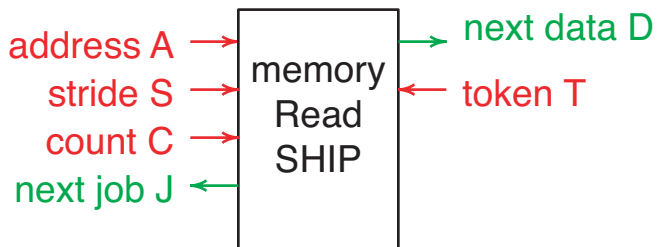
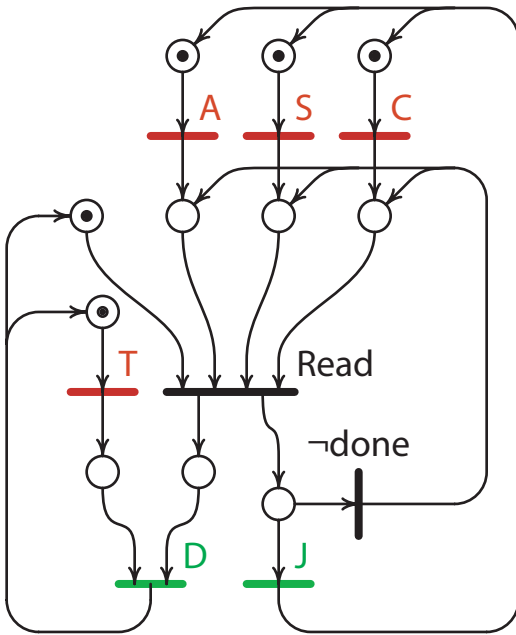
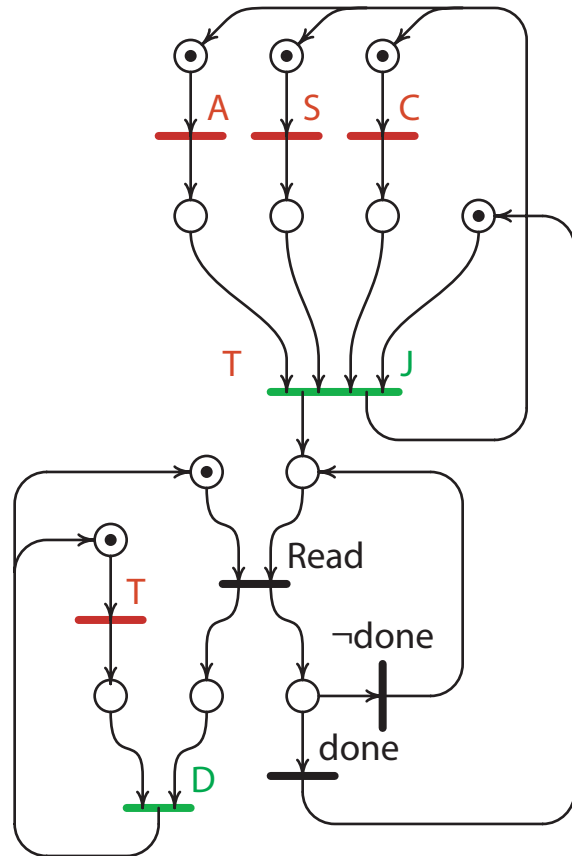


Figure 3: Memory Read Petri Nets



a) without input buffer



b) with input buffer

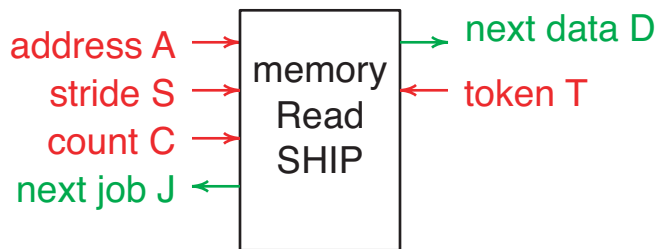


Figure 4: Memory Read Petri Nets