UC Berkeley Computer Science

Subject: **A Description of a Fetch SHIP**
Date: **October 10, 2006**
From: **Ivan Sutherland**
UCIES **#2006-is37**

References:

UC# 2006-AK02: Fetch SHIP, Amir Kamil, 8 October 2006

INTRODUCTION

Our task is to define a Fetch SHIP. Amir Kamil reported in class about the form chosen by his group. In AK02 Amir has recorded the diagram of a Fetch ship, showing all of the possible inputs:

PreFetch::*codeBagDesc*
Activate::*boolean*
Execute::*codeBagDesc*

and one output:

Done::*token.*

In class, Amir's description of the SHIP considered two uses for the Fetch SHIP: either as an unconditional branch or as a conditional branch. Only the Execute and Done signals are relevant to the unconditional use. Only the PreFetch, Activate, and Done signals are relevant to the conditional use.

FOUR FETCH SHIPs

In this memo I shall separate the two uses. I will assume that the Fetch SHIP behaves as if it were four separate SHIPs, one unconditional SHIP and three conditional SHIPs. The two bits to be decoded by the SHIP and not the switch fabric will distinguish these four SHIPs. Therefore there will be four separate Done signals, distinguished inside the Fetch SHIP by the two bits of source address that the SHIP gets to decode. The program can access these separate done signals separately.

The two bits of destination address that the SHIP gets to decode will steer the code bag descriptor to the proper one of the four SHIPs. Therefore, we can combine the three PreFetch signals and the one Execute signal without loss of generality. I choose to call them CBD for short, with subscript 0-3 where necessary to distinguish them. The two bits of destination that the SHIP decodes will send the CBD to the Execute or the proper PreFetch input.

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.

Similarly, the two bits of destination address that the SHIP gets to decode will steer the Activate signal, a boolean. This signal is relevant only to the conditional units. Therefore one combination of the two destination bits is unused for that signal, namely the combination that would send it to the unconditional fetch unit. Activate[0] does not exist.

SHIPs AND THE SWITCH FABRIC

This memo has two purposes. One purpose is to provide a more detailed description of the behavior of a possible Fetch SHIP. The other purpose is to show how SHIPs interact with the switch fabric.

The switch fabric is an asynchronous pipeline. Each stage of the switch fabric interacts with its adjacent stages via handshake signals. Where the switch fabric and the SHIPs meet there must also be two-way handshake signals, although we often represent them as a single input or output arrow.

Destination = Input: Consider, for example, an already-occupied SHIP input. What happens if the switch fabric attempts to deliver another value to an occupied input? Obviously, the SHIP's input can't accept the new value yet and so the last stage of the switch fabric must retain it. Nevertheless, the switch fabric sends a signal to the SHIP input saying that the new value is available. This is the first part of the two-part handshake. I think of the switch fabric as "proffering" the new value to the SHIP.

When the SHIP is ready to accept the new value, the SHIP copies it from the switch fabric. Because the switch fabric can't actually detect when the SHIP copies the data value, the SHIP must also tell the switch fabric that the data are now safely aboard. That is the second part of the two-part handshake. After that "acknowledgement" signal, the data in the last stage of the switch fabric are no longer of value. We say that the last stage becomes "empty" although there's actually no change to the data latches.

Remember that there are two signals forming the complete "handshake" between the switch fabric and the SHIP's input. Although we normally show only one arrow from the switch fabric to the SHIP, the full handshake is nevertheless there.

Source = Output: A similar two-part handshake appears at a SHIP's output, a source for the switch fabric. The first stage of the switch fabric, which we have called the OutBox, combines an instruction and the data output from the SHIP. Ignoring for now the instruction part, consider only the interface between the SHIP's output and the OutBox. The SHIP must signal to the OutBox that data are available from the SHIP. This is the first part of the two-part handshake. I think of the SHIP as "proffering" the data to the switch fabric.

The SHIP must retain the data value until the switch fabric copies it. Because the SHIP can't actually detect when the OutBox copies the data value, the OutBox must

also tell the switch fabric that the data are now safely in the OutBox. That is the second part of the two-part handshake. After that “acknowledgement” signal, the data in the output stage of the SHIP are no longer of value. We say that the SHIP’s output becomes “empty” although there’s actually no change to the data latches.

Notice that for a COPY instruction the OutBox need merely delay sending the acknowledge part of the handshake. The acknowledge from the OutBox guarantees that the OutBox no longer needs the data value.

Remember that there are two signals forming the complete “handshake” between the SHIP’s output and the OutBox. Although we normally show only one arrow from the SHIP to the OutBox, the full handshake is nevertheless there.

THE SIMPLE FETCH SHIP – Figure 1A

In Figure 1A we see a Fetch SHIP with three terminals: CBD, Done, and Fetch. The general idea is that the programmer delivers a CodeBagDescriptor to CBD, whereupon the Fetch SHIP gets some new code from memory and puts it into the instruction pool. When the Fetch SHIP is ready to receive a new CodeBagDescriptor, it issues a token on the Done signal.

To the left of Figure 1A appears a Petri diagram of this action. The circles are called “places” and the bars are called “events.” Each event carries a label. Some places are marked by a dot, called a token. In the Petri diagrams we’ll use there will be at most one token in any place. In Figure 1A only the topmost place has a token.

Here’s how Petri nets work. An event may occur only if there is a token in each and every place that leads to that event. Notice that I didn’t say must, I said may. It’s permissive not compulsory. When an event occurs, it consumes all of the tokens in the places that lead to it. It then produces a brand new set of tokens and puts them in all the places that lead away from the event. Tokens do NOT move – they are consumed and created; the number of tokens may change.

In Figure 1A the top place has a token, and therefore represents the starting condition. The only event that can occur is CBD. After CBD, the only event that can occur is Fetch, and after Fetch, the only event that can occur is Done. Thus the loop and the single token tell us that the three events occur in sequence: (CBD, Fetch, Done)*, where comma represents sequence and the ()* notation means repeat forever.

We can represent the same idea in CSP notation as shown in Courier type below the diagram. The CSP defines the behavior of the fetchShip as:

wait until a CBD event happens and then,
in sequence (double semicolons) do two things
do the Fetch and emit (the exclamation point) a token on Done.

THE SIMPLE FETCH SHIP – Figure 1B

The situation is a little more complicated if we consider also the double handshake signals that we omitted in Figure 1A. Here we see half arrows to represent each part of the double handshake on the CBD and Done terminals. The red half arrows with italic labels, CBD and DoneAck are generated by the switch fabric; the black half arrows are generated by the Fetch SHIP.

Now the Petri diagram is a bit more complex because it has to describe not only the behavior of the Fetch SHIP but also the details of its interaction with the switch fabric. It took me three or four tries to get the Petri diagram as simple as shown here. My previous tries implied that the Fetch SHIP would actually store the CBD internally. I wanted to represent the Fetch SHIP without internal storage because we can easily add pipelines to its input and output if we wish.

Initially, only the CBD event can happen. In other words the switch fabric has to start the action. Once the switch fabric proffers the CodeBagDescriptor, the Petri net consumes the top token and moves it down to the next place. Now the Fetch event can happen, because all of its input places have tokens. Here, however, I've shown the Fetch event as two separate events, one to start the Fetch process and the other to indicate that the Fetch is over. Note that when Fetch fires, it consumes two tokens and produces only one in the middle place. Now the memory is busy getting instructions and putting them into the instruction pool. When that's all over, the FetchOver event happens. Remember that I said that tokens permit, but don't require events to occur; in fact, FetchOver won't occur until some other things happen that are not represented in the figure.

When FetchOver fires, it consumes one token and produces two, thus enabling the Done and CBDack signals. The path split at FetchOver is characteristic of how Petri diagrams represent concurrent events. We say that Done and CBDack are concurrent, meaning that they can occur in either sequence or "simultaneously" whatever that may mean. Each consumes a token and places a new one at the top of the diagram, enabling DoneAck and CBD.

Notice the asymmetry in the diagram. Whereas the right side path for CBD and CBDack have an initial token at the very top, the left side path for DoneAck and Done have the initial token lower down. This indicates that although CBD and CBDack must alternate, CBD comes first, and although DoneAck and Done alternate, Done comes first. Interesting fine point. After the first CBD event, the situation is entirely symmetric.

We can represent the same ideas in CSP notation. Here we define detailFetchShip to be a sequence (represented by ;;) of two things. First, wait (represented by the ?->) for CBD and put its value into a variable called x. Second, do the repeat statement forever. The initial use of CBD is like moving the token on the right

of the Petri net down one place. After that startup action, the repeat statement is nice and symmetric.

The repeat statement itself consists of a sequence (represented by ;;) of two things: sending (represented by !->) x to the Fetch device, and when it has done its job, taking on two concurrent (represented by ||) sequences. The first sends (represented by !->) a token to Done and then waits for DoneAck, putting its value into nowhere (represented by _). The second sends (represented by !->) a token to CDBAck and then waits for CBD again (represented by ?->) and puts its value into x.

The Petri diagram and the CSP differ ever so slightly. How? Does the diagram correspond to the Petri diagram or the CSP?

CONCURRENCY

I want you to get three messages from this memo so far. First, it is possible to describe how events relate to each other in sequence or concurrently. I have offered two description languages here: Petri diagrams and CSP. There are many others.

The second message I want you to get is that many different descriptions are possible. Figure 1A and Figure 1B describe the same behavior but at a different level of detail. Mostly we will ignore the details set forth in Figure 1B. When actually designing SHIPs, however, they may prove important.

The third message may not immediately be evident. These descriptions are hard to come by and even harder to check. Is the behavior that I have described what the Fetch SHIP group intended? Going from a general description of behavior to a precise description of behavior takes careful thought. Adam and I spent many hours on the descriptions in this memo.

CONDITIONAL FETCH SHIP – Figure 2

The conditional Fetch SHIP of Figure 2 has one more input, a boolean, that Amir called Activate and I have called "use." I like shorter names. The general idea is that after the SHIP gets both a CBD and a boolean it will either do the fetch job or not, and in any case return a Done token. Again I have used red italic arrows and names for signals that originate in the switch fabric. Hidden from view, of course, are the other halves of the two-part handshakes involved.

Consider the Petri diagram first. At the top left is a place with a token that leads to two events: use=T and use=F. Whichever event happens will consume the one and only available token, preventing the other event from happening. This is the idiom in Petri nets we use to represent mutually exclusive events, in this case use=T and use=F. Of course they are mutually exclusive because the boolean value is either TRUE or FALSE. These two events represent the two possible input values that can arrive on the use input.

At the top right is a place with a token that leads to CBD. That indicates that after initialization CBD may happen whenever the switch fabric is good and ready to cause it.

At the center of the Petri diagram are two events called XXXX and Fetch. In order for either to occur two tokens are required. For Fetch to happen, both CBD and use=T must have happened. For XXXX to happen, both CBD and use-F must have happened. I call the XXXX event “have a Foster’s beer”; it does nothing. I thought of calling it XX instead, but that’s Mexican beer and my Australian friends tell me that Australian beer is much better.

Whether it’s the XXXX event or Fetch that occurs, two tokens are consumed and one new one is created in the place just above Done. Note that XXXX and Fetch are mutually exclusive if only because the both require the token that was created by CBD. Finally Done consumes one token and creates two back at the top of the diagram to repeat the entire action.

Remember that details of the two-part handshakes have been omitted from these diagrams. Thus they offer only three essential ideas: First both “use” and “CBD” must come in before Fetch could possibly happen. Second, if use is FALSE, the value of CBD is ignored and Fetch doesn’t happen; instead the Fetch SHIP “pauses for a Foster’s.” Third, whether or not Fetch happens, the Fetch SHIP will return a Done token. That, I believe is the essence of the intent of this Fetch SHIP design.

Now consider the CSP description. It defines conditionalFetchShip as an endless repetition of a sequence (represented by ;;) of two actions. The first action in the sequence is a concurrent pair of guarded events (represented by I) one of which waits for both (represented by comma) CBD and use=T and the other waits for both (represented by comma) CBD and use=F. Only one of these actions can occur because use is either T or F but can’t be both. Thus the two guarded events either send (represented by !->) CBD to Fetch or drink a beer. Finally, the second action in the sequence sends (represented by !->) a token to Done.

These two representations are supposed to be equivalent. Are they? Which do you prefer? Does the meaning leap off the page into your mind? This is not easy stuff, but it’s very important.

FOUR SHIPs TOGETHER

Figure 3 shows my view of the complete Fetch SHIP. There are four parts in the SHIP designated by subscripts 0-3. Unit 0 is the simple unconditional branch. The other three are of the conditional branch type.

The only remaining thing to do is to indicate that at any one time only one of these units can actually Fetch instructions. In other words, I intend that the four Fetch[i] signals be mutually exclusive. The Petri diagram at the bottom of the figure represents

my desire for mutual exclusivity. By Fetch[i] I mean the complete fetch cycle involving both the start and completion of the Fetch cycle. I might, instead, have represented both the start and completion of the Fetch process as two separate events. All that would show is what I really intend, that a new Fetch process can't start until an older one is entirely over.

Dictionary

prof•fer |ˈpräfər|

verb [trans.]
hold out (something) to someone for acceptance; offer : *he proffered his resignation.*

noun poetic/literary
an offer or proposal.

ORIGIN Middle English : from Anglo-Norman French ***proffrir***, from Latin ***pro-*** ***'before'*** + ***offerre*** ***'to offer.'***

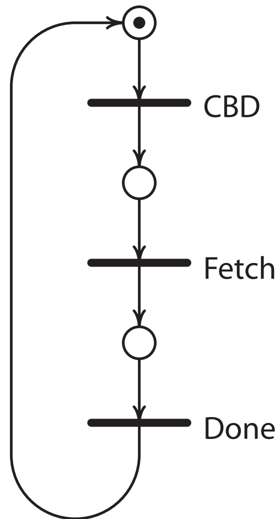
Thesaurus

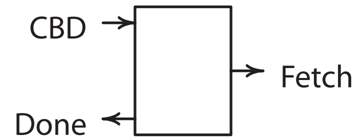
proffer

verb

Coleman proffered his resignation OFFER, tender, submit, extend, volunteer, suggest, propose, put forward; hold out. ANTONYM refuse, withdraw.

UCIES# 2006-ies37
10 October 2006

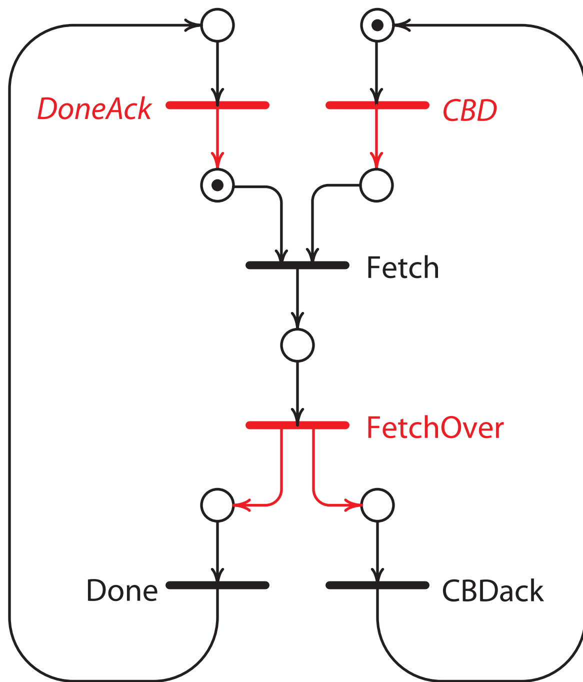


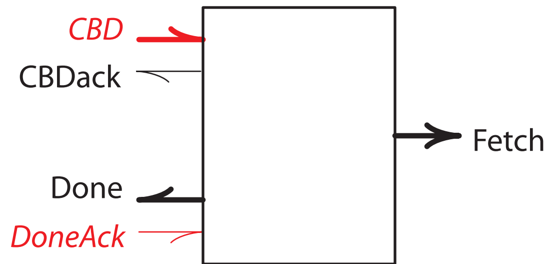



```
fetchShip =
|CBD => ;; Fetch
        ;; token !-> Done
```

A) Ignoring details

Switch fabric actions in red italics

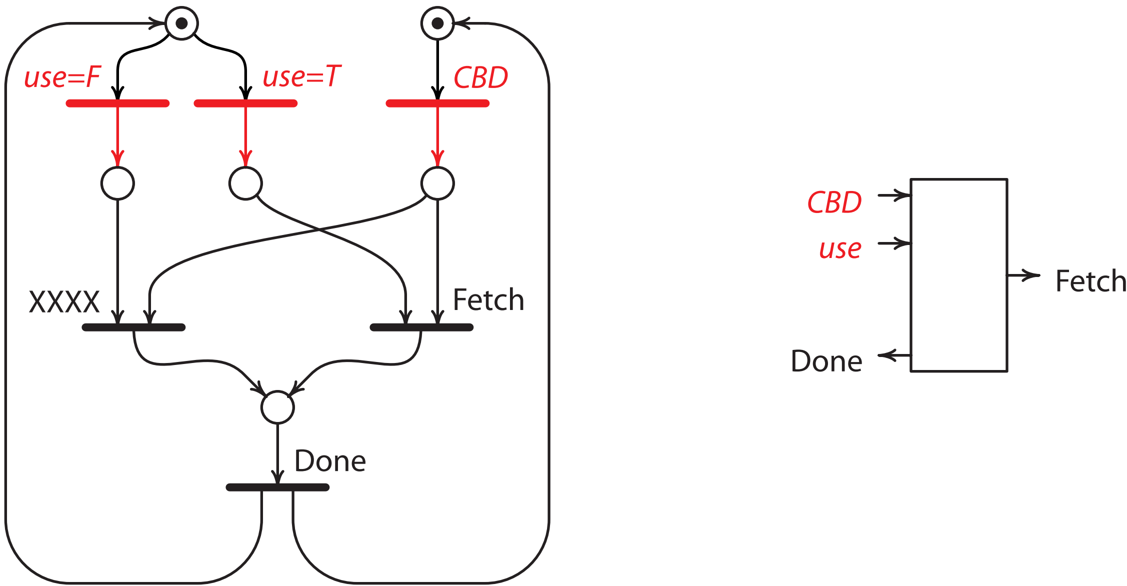





```
detailFetchShip =
;; CBD ?-> x
;; repeat
  ;; x !-> Fetch
  ;;
    || token!->Done   ;; DoneAck?->_
    || token!->CDBAck ;; CBD?->x
```

B) Showing details

Figure 1: Simple Fetch SHIP




```
conditionalFetchShip =
repeat
  ;; | (CBD,use=T) => CBD !-> Fetch
     | (CBD,use=F) => DrinkFostersBeer
  ;; token !-> Done
```

Figure 2: Conditional Fetch SHIP

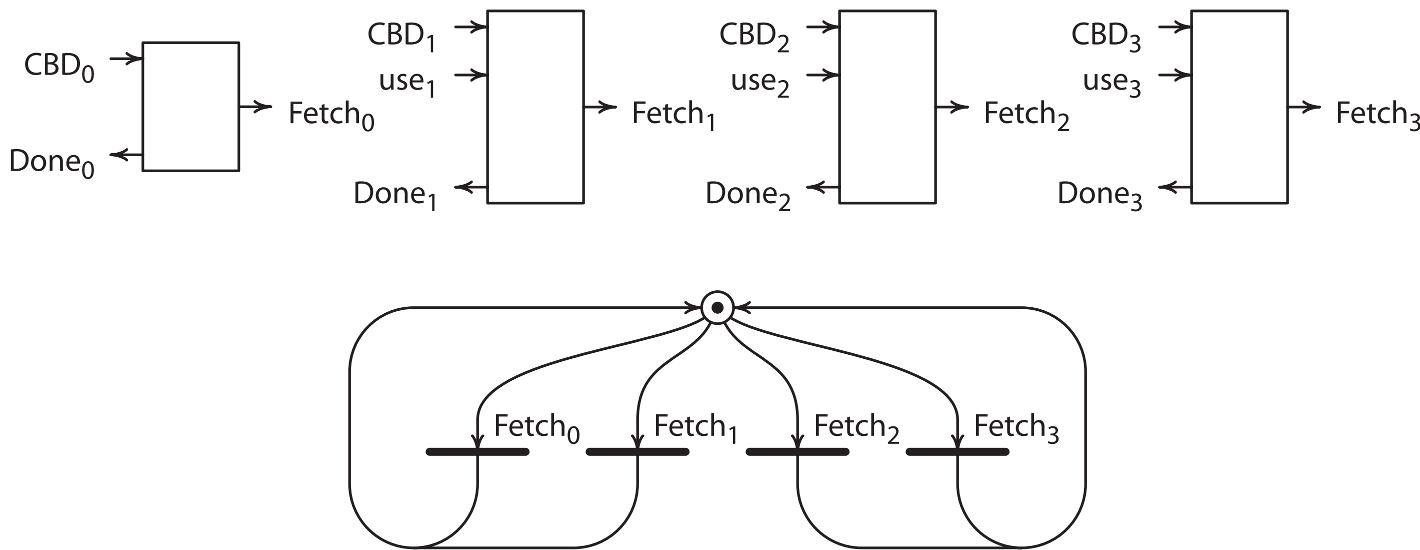


Figure 3: Four-part Fetch SHIP