UC Berkeley Computer Science

Subject: An OutBox Control
Date: January 25, 2007
From: Ivan Sutherland
UCIES: #2007-is45

References:

UCAM13: Unified Boxes, Adam Megacz, 18 January 2007
UCAM14: Syntax, Adam Megacz, 18 January 2007
UCAM15:: Decommissioning and recycling Instructions, Adam Megacz, 18 January 2007

PURPOSE

This memo describes the general properties of an OutBox and proposes an asynchronous implementation for it. The memo seeks to define the structure of the OutBox clear at the block diagram level. A variety of implementations are possible.

INTRODUCTION

An OutBox attaches to each output of each SHIP. An OutBox gets Instructions that control its behavior as described in UCAM13. Before executing an instruction, an OutBox may wait for data from its SHIP or for a token from the switch fabric or both. When it executes the instruction, it may transfer data from the SHIP into its internal data register or leave the content of its register undisturbed. It may also send the data in its register into the switch fabric for delivery to the destination specified in the instruction. Instead of sending data it may send a token into the switch fabric.

The OutBox also has an instruction ring FIFO as described in Adam Megacz's memo, AM15. This FIFO permits the OutBox to repeat a few instructions until their counts run out, or until it receives a "kill" instruction to remove the instructions from the FIFO.

The main design challenge is to use a single arbiter. There are two tasks that might have required arbitration: A) entry of new instructions into the FIFO ring and B) orderly destruction of instructions already in the ring. This design combines these two functions.

ASYNCHRONOUS MODULES

This memo offers a "straw man" design of the OutBox in terms of separate asynchronous modules. The kind of module described acts when its input or inputs are "full" and its output or outputs are "empty." When it acts, it does three things as a single atomic action: A) It latches its input data and proffers the latched data as output to its successor or successors. B) It declares its input to be "empty." C) It declares its output to be "full."

Many implementations of this kind of behavior are possible. Among them are two asynchronous forms used at Sun that are known as asP* and GasP. Different asynchronous forms represent "full" and "empty" in different ways. GasP, for example, represents "full" and "empty" using "state wires" that run between modules. The module at one end of such a wire drives it HI to represent "full", and the module at the other end drives it LO to represent "empty." The asP* form uses an r-s latch between stages to represent

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.

"full" and "empty" and requires two wires between modules rather than one. Synchronous implementations are also possible using a flip flop to represent "full" and "empty."

BLOCK DIAGRAM – Figure 1 and Table 1

Figure 1 shows the set of asynchronous modules proposed here to implement an OutBox. The OutBox parts lie in the shaded part of the figure. Because we have yet to build such a device, I offer this figure for comment and correction. In the figure, each circle represents a control module named with a single upper case letter. Each control module delivers a latch pulse to a data register composed of latches and represented as a dark polygon. The data registers also carry names, a single upper case letter each. A control module and its associated register are collectively known as a "Stage." Table 1 below lists the control modules, their inputs and output state wires, and the registers that they control.

Table I: Asynchronous modules in Figure 1

module	input state wires	output state wires	register	comments
S	unnamed	s	S	An output of the SHIP. The SHIP fills s when the SHIP produces output.
T	unnamed	t	none	A token output from the switch fabric
D	s, ss, t, tt, i	d, u, j, o	D, J or O	D fires each time it executes the instruction in register I. D may copy all of I into register O, or it may pass (count – 1) from I into register J. D is mutually exclusive with R.
F	d	unnamed	F	The first stage of the data switch fabric
G	u	unnamed	none	The first stage of the token switch fabric
M	unnamed	m	M	A fresh instruction from the instruction horn
N	m	n, nk	N	An instruction waits here prior to entering the ring. Kill instructions kill and then die here.
H	p or later	h	H	The last stage of the ring FIFO
I	h	is, it, i	I	The instruction waits here until executed in D. Note that I makes i full and may make is and it (ignore t and ignore s) full for some instructions. Module E may also load register I.
see D			J	register controlled by D
see D or R			O	register controlled by D or R
R	i, n, nk	o	O	Fires to insert a new instruction or kill an old one. Mutually exclusive with D.
E	j	is, it	I	Fires between executions of a repeated instruction.
P	o	p	P	First stage of the ring instruction FIFO

In Figure 1 data flow generally upward, as suggested by the shape of the register symbols. Data from register I can flow down to register O along the curved data path. Although the figure omits the connection, the FIFO ring for instructions is closed by a

connection from the output of register P down to the input of to register H. The path from P to H might contain additional stages of FIFO omitted from the figure.

Two data registers where data paths may merge, namely I and O, have twin data inputs and twin latch pulses. The latch pulses are mutually exclusive, i.e. only one will occur at a time. Each latch pulse causes the register to capture and hold the data presented at its corresponding input. Thus control modules D and R can cause register O to latch instructions coming from registers I and N respectively. This is how fresh instructions enter the FIFO ring. Similarly, both control modules E and I can place instruction bits into register I. Control module E acts only when the OutBox uses a counting or standing instruction again. Control module E changes only those bits in register I necessary to decrease the count.

Each of the control modules has one or more state wires connecting it to control modules above and below it. These state wires carry names, generally a single lower case letter. Often the state wire above a control module carries a name related to the control module that sets it “full”. For example, near the bottom of the figure, state wire h is made full by control module H whenever H fires. State wire h, in effect, says that data from register H are valid.

Each of the control modules may wait for several conditions before firing, and each may produce several results depending on the instruction involved. Because flow is generally upward, each control module acts when the state wires below it are “full” and the state wires above it are “empty.” When it acts it generally reverses the condition of all state wires attached to it.

In the case of particular instructions, however, a control module may not wait for some condition. For example, control module D waits for data from the SHIP by watching state wire s unless the instruction indicates otherwise. The instruction in register I may set state wires is and it to tell control module D to ignore data from the SHIP or a token from the switch fabric respectively.

Some state wires connect more than two modules. For example, state wire i connects control module I to control modules D and R. Control module I fills state wire i when it loads an instruction into register I. Either control module D or control module R may change state wire i back to the empty state. Control module D drains i when it executes the instruction, possibly copying the instruction into register O to travel around the instruction ring and return via stage H to execute again. Control module R drains state wire i to kill an existing instruction.

The flow of instructions from the FIFO ring through execution and back into the FIFO ring follows registers H, I, O, and P. The ring closes through a connection, omitted from the figure, between stage P and stage H. This connection may include additional FIFO stages. New instructions enter from the instruction horn via stages M and N. Data from the SHIP flows into the switch fabric along the route S, D, F.

FUNCTIONAL OVERVIEW

An instruction reaching stage I is ready to execute. If stage D wins the arbitration and fires, the instruction in I controls what stage D does. State wires is and it carry information to D about whether or not it should wait for s or t respectively or ignore them. When D fires, register D may capture data from S, and state wire d may indicate that data are available for output to the switch fabric, F. This completes the data part of the instruction.

In addition, when D fires, all or parts of the instruction pass into register J or O. The notation " -1" near the output of register I shows that the count is reduced by 1 upon leaving register I. If the instruction recirculates, the instruction, with reduced count, passes into register O.

On the other hand, if the instruction repeats, parts of it, especially the count, require modification. The decremented value of the count goes into register J and returns later to register I. State wire j indicates that register J and the data path from J to I are "full", enabling control module E to fire. When module E fires it reloads the instruction register I with an appropriately updated version of the instruction. For a repeating instruction, D fills state wire j but avoids draining state wire i. Therefore, control module E is mutually exclusive with control module I.

On the other hand, there may be a new instruction at stage N. If this is a "kill" instruction it will fill state wire nk. State wire nk was erroneously called np in some early versions of the figure. If it is an ordinary instruction it will fill state wire n. Let us consider these two cases separately.

First, a kill instruction cannot act until there is an instruction in stage I for it to kill. Thus stage R waits for both np and i to be full before firing. Second, for an ordinary instruction destined for the ring FIFO, module R can act whenever it gets control of the arbitration. In this case, module R waits only for n to be full and o to be empty.

The rest of this memo offers more detail about each of the control modules in turn. Each control module will fire only when certain of its state wires are "full" or "empty". Each will take certain actions to fill or drain its state wires and to latch data. The conditions for taking these actions control the overall behavior of the OutBox. The rest of this memo describes those conditions in detail.

STAGE N

Stage N fires when both n and np are empty and m is full. When it fires it always loads register N. If N is an ordinary instruction it fills state wire n. If N is a kill instruction, it fills state wire nk. Note that delay considerations may force stage N make n or nk full before the bits of the new instruction have actually settled in the register N. To do this, stage N may use the input to register N, rather than its output, to condition state wires n and nk at the same time as it loads the instruction into the latches of register N.

STAGE R

Because stage R is an arbitrated stage, we must distinguish its request to the arbiter from its firing conditions. I assume that stage P will promptly drain state wire o, and so I shall avoid including the state of o in the request for arbitration.

In the case of an ordinary instruction, state wire n becomes full and Stage R requests service from the arbiter. Moreover, if service is granted, stage R will latch the instruction into register O, fill state wire o, and drain state wire n.

In the case of a kill instruction, state wire nk fills. In this case Stage R must wait until the instruction it is to kill reaches stage I. Therefore, module R requests service from the arbiter only when both state wires nk and i are full. If service is granted, module R drains state wire i to kill the instruction in I. If the kill instruction is done, stage R also drains state wire nk to let a subsequent instruction enter stage M. A kill instruction never loads O, nor fills state wire o.

There is, however, one difficult case. Suppose that when state wire nk becomes full, stage I is empty, as indicated by state wire i being in the empty state. In this case, both stage R and stage D are waiting for state wire i to become full. When state wire i becomes full, both control modules D and R might simultaneously request service from the arbiter.

The outcome of the arbitration matters. Suppose that stage I got service first. Stage I would recirculate the instruction via register O and drain state wire i, leaving the system back in the condition where both D and R wait for i to fill. Thus we must somehow bias the arbiter so that it reliably grants service to the request to R in the case that state wire i is the last to become full. We do this by ensuring that the full state of i reaches control module R slightly before it reaches control module D. For asynchronous arbiters a small time bias is equivalent to priority.

STAGE D

This is the most complex control module. Notice that five state wires precede stage D, namely s, is, t, it, and i. The state wires ss and tt are set by bits in the instruction that indicate whether or not to ignore data from the SHIP or to ignore a token from the switch fabric or both. The wait conditions for the SHIP and the token are state wire s and t respectively. Thus, if we treat the meaning of is and it as “ignore the corresponding state wire,” the input condition for stage D’s predecessors is:

i & (s v is) & (t v it).

As with stage R, I’m going to assume that the output stages from D, namely J and O, will drain rapidly. Therefore, I omit the need for state wires j and o to be empty as a criterion for requesting service from the arbiter.

Table 2 below describes three different kinds of instructions for which the behavior of stage D differs. The first column is for an instruction that doesn’t recirculate but is in its last execution. It may have had an initial count of 1 or its count may now have run out. The second column is for an instruction that will recirculate. These instructions execute once for each recirculation. The third column is for a counting or standing instruction that will repeat again and again by remaining in stage I.

Table 2: Actions of control module D

	one shot only	**recirculate**	**repeated**
state wire s	drain if data wait	drain if data wait	drain if data wait
state wire t	drain if token wait	drain if token wait	drain if token wait
state wire is	drain	drain	drain
state wire it	drain	drain	drain
state wire i	drain	drain	leave full
state wire j	never	never	fill
register J	never	never	load
state wire o	never	fill	never
register O	never	load	never
state wire d	if load data	if load data	if load data
register D	if load data	if load data	if load data

STAGE E

Stage E is included to act after stage D for repeating instructions. Instructions that recirculate fill state wire o. Instructions that repeat fill state wire j. Instructions that fill neither i or o are in their final execution. Thus stage D serves repeating instructions. State wire j, its predecessor state wire, indicates that there are some valid instruction parts in register J. Stage E fires whenever state wire j is full because Stage E has no successor stage. Moreover, as indicated in Table 2, when state wire j fills state wire i remains full. Thus stage E will fire only when there is an instruction present in stage I.

Stage E always drains state wire j. It also copies some bits from register J into register I to modify the instruction for its next cycle. For example, it must reload the counter bits for a counting instruction. Notice that stage E also has access to state wires it and is. It fills them if the bits in the instruction indicate that the instruction should ignore s or t.

There is some timing work to be done in the case of a kill instruction. Suppose that E fires just as stage R fires to kill the instruction in I. Is there a conflict between stage R draining state wire i and stage E loading some parts of register I?

UCIES# 2007-is45
25 January 2007

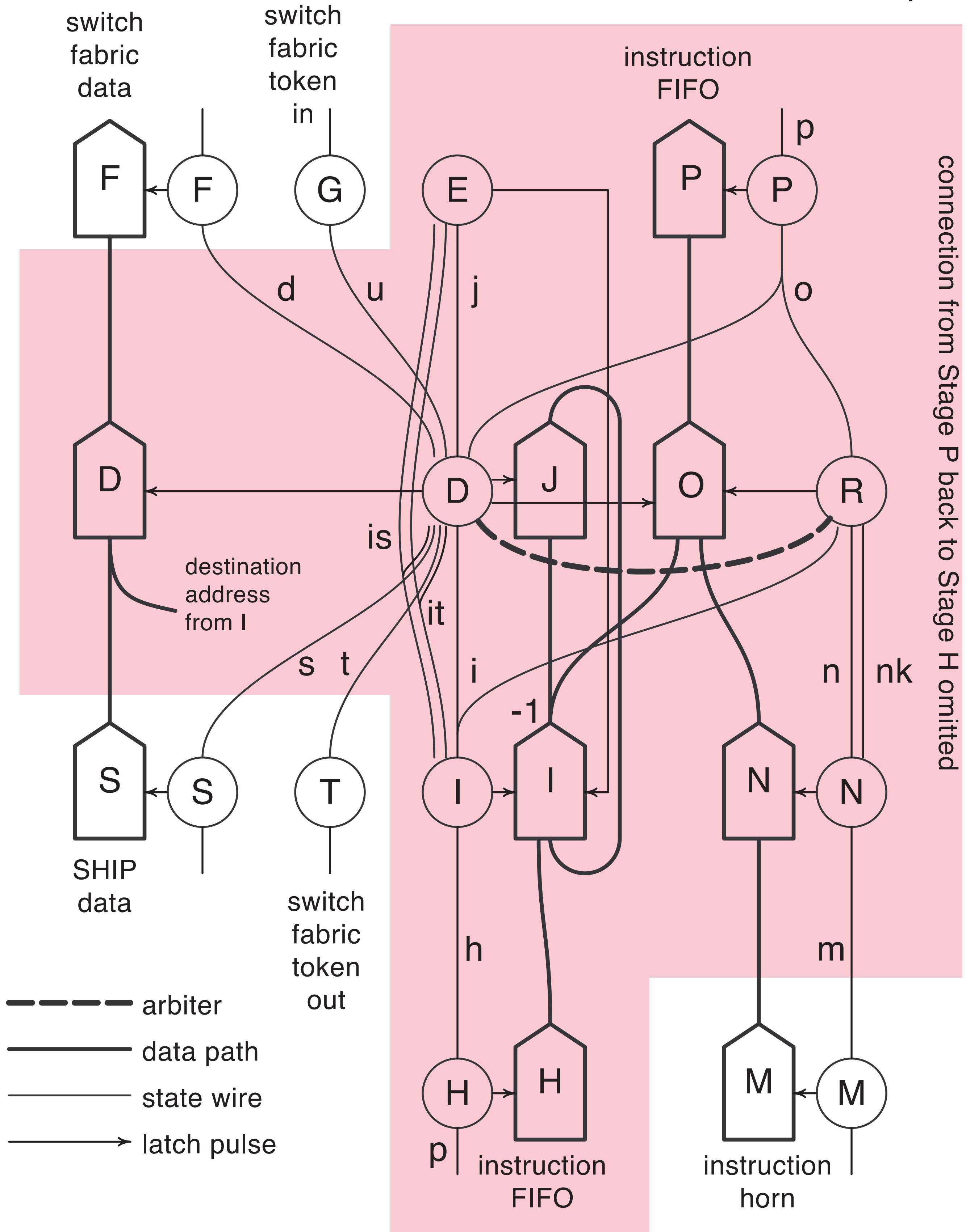

Figure 1: OutBox data and control paths