GREG GIBELING
GDGIB@EECS.BERKELEY.EDU
UC BERKELEY
11/1/2005

CS294-4 Homework #7

1.0 SHIPs & Mechanisms

During the course of CS294-4, there have been many proposals for FLEET in general and SHIPs in particular, often as a direct result of a toy problem under consideration. However, as time progresses, and with each passing discussion of implementation details I find it more pressing to enumerate the list of not only SHIPs, but mechanisms and data types in FLEET. This section attempts to do so without explanation, which it is my intent to provide in later sections.

1.1 Mechanisms

In this section I list the basic features of FLEET which make it unique. This should be a relatively straightforward re-hashing of that which we already know, or have been assuming, in a more concise summary form.

- The `move` instruction
- Concurrent code bags & descriptors
- Literal instructions
- Out of band data values for all data types
- Single producer, single consumer data values
- Lossless, out of order data delivery

1.2 Data Types

One of the original pieces of the FLEET proposal which set it apart from existing architectures was the introduction of data types at the ISA level. This section lists those types which we have found essential to date.

- Integers
- Tokens
- Booleans
- Floating Point

On and off there has been discussion of a string or character data type as well. I find this unnecessary at the ISA level, unless we intend to add SHIPs which directly process character data. In light of that ASCII to Unicode switch, I question the wisdom of this approach and have therefore left mention of these data types out of this document. I am not answering questions about the wisdom or value of their inclusion, merely ignoring them at the present time.

I would also like to take this opportunity to steal a notation from type theory publications that a variable or port name should be annotated with a subscript indicating its type, namely the first letter of the type, and will appear in italic, when written in the text of this document. Thus a token might be X_T.

1.3 SHIPs

On of the most fun parts of this class has been proposing, specifying and suggesting uses for SHIPs. However, in our general discussions we have often digressed into constructing rather strange and specialized SHIPs, which I do not imagine will be of overwhelming value (eg, a Warnock sorter). In light of this, I am here attempting to list those basic SHIPs, whose inclusion in FLEET I find essential at this stage. Further discussion and additional SHIPs are clearly warranted.

Because the SHIPs vary widely in their intended uses and reasons for inclusion the list below is broken into sections of descending commonality and increasing specialization. The first section describes operations which are natural for today's CPUs, and the second optimizations and additions which are conceptually obvious if not commonplace in implementation. However the third section describes operations entirely unique to FLEET and it's design.

- Standard Operations
  - 2-input ALU (Add, Subtract, Bitwise Operations)
  - Logical Test
  - Bitwise Inversion
  - Memory Read
  - Memory Write
- Semi-Standard Operations
  - 2-input Boolean Operations
  - Boolean Not
  - 3-input fused multiply and add
  - 3-input Add
  - Register
  - Filter
  - Real Time Clock
- Unique Operations
  - Counted Input And
  - Triggered Token Source
  - Replicate
  - Gateway
  - Bit-Bucket
  - FIFO
  - Load Code Bag

2.0 SHIPs: Specifications and Justifications

To date most of our discussions and results have been in English, and this is fine. However in light attempting to write this section, I find that perhaps the time is right to begin thinking about formal specifications of the behavior of a FLEET system. I hope that this section will be well written and clear, but without a formal notation I fear we will still find room for debate and confusion.

2.1 Standard SHIPs

2.1.1 ALU (2-Input, Configurable Operation)

I find this to be the most basic and obvious SHIP required, capable of performing any simple two input operation on each pair of (integer) values which arrive at its inputs. It's operation would be controlled by an encoded integer input, allowing a certain amount of interesting flexibility.

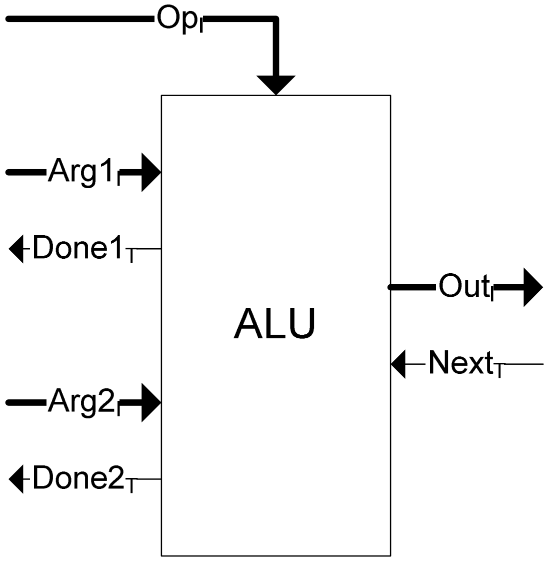

Figure 1: ALU, 2-Input

First of all, by connecting the output, Out_I, to the input $Arg2_I$, and we can produce an interesting variant on an accumulator based machine, with the ability to feed a series of inputs and operations in using $Arg1_I$ and Op_I.

Second, having a data controlled operation will allow us to more easily replicate the functionality of this unit (perhaps adding a hundred or them) without restricting the mix of operations our FLEET implementation requires to perform efficiently.

Third, at a hardware level all of these operations require very similar transistor structures, making separate implementations a much less attractive alternative. This is especially true in a chip where area is almost certainly going to be dominated by wiring, as in FLEET's switch fabric.

2.1.2 Logical Test

I aint finished yet. Don't got the time…

3.0 Example: A Composite Stride SHIP

In this section I give an example of how to construct expanded functionality from the primitive SHIPs listed above. This is clearly motivated by Igor's thinking from 10/31/05.

First of, the reader deserves some explanation as to why I chose to implement this SHIP out of others rather than as a primitive, as we have previously assumed would be the case. My primary motivation for this was the realization that any debate we were having as to the external functionality of a SHIP like this, especially with respect to loop initialization and termination conditions, cannot be solved easily with a single answer. No matter how we build a stride SHIP, it will be wrong for some value of wrong.

What is more so: if we operate under the assumption that we have on the order of 1000 SHIP ports available to us (assuming 20bits of a 32bit move instruction are for source and destination with half going to each) it is reasonable to assume that will gain performance not be adding specialty SHIPs such as Warnock or perhaps even stride, but rather by replicating the simplest SHIPs and allowing a programmer to make their own complex behaviors. Of course this behavior has a limit: the 3-input adder is still included as a SHIP because it is very useful, and easy to specialize back to a 2-input adder. Only performance numbers can really place the optimal boundary between direct SHIP implementation and composite functionality.

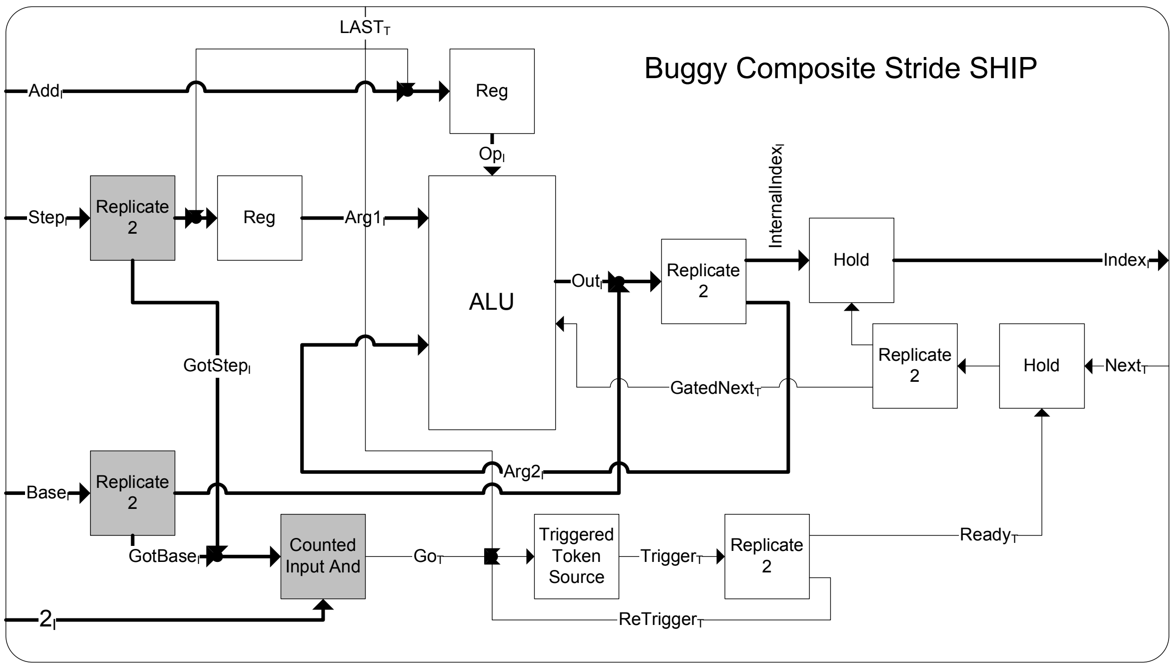

Figure 2: A Buggy Composite Stride SHIP

Shown in figure 2, above is a complete composite stride SHIP built from the primitive SHIPs as described in section 2.0 SHIPs: Specifications and Justifications. Some of the key features of this design include the fact that one could easily attach an external limit by using a 2 input ALU, a logical test and a LAST token generator, connected to the *LAST$_T$* input.

The grayed SHIPs in the figure also have an interesting feature that they are used only at startup, and thus could be reused as soon as the *Ready*$_T$ move is fired for the first time. A programmer could replicate that signal another time and use it load a new code bag which is able to re-use the gray SHIPs.

Equally interesting is the combination of the *Triggered Token Source* and *Replicate2* SHIPs, which are connected in a feedback loop. Upon the entry of the two standing moves *Trigger*$_T$ and *ReTrigger*$_T$, the token source is idle, and thus the two SHIPs are doing nothing at all. However once the *Counted Input And* fires a token through *Go*$_T$, we now have an infinite token generator. However, there is a flaw in this implementation, which might only now be obvious! The infinite token source is likely to jam the switch fabric with a constant stream of tokens, result in the LAST token never being seen, and possibly, deadlock. The correction is shown in figure 3 below.

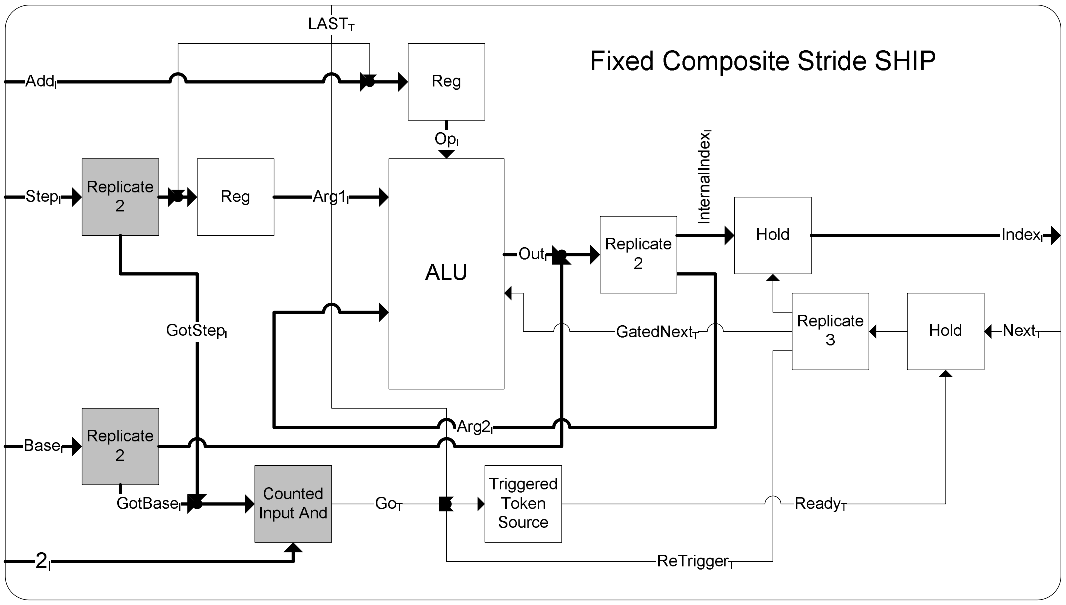

Figure 3: The Correct Composite Stride SHIP

The fairly simple correction to this problem was to widen the feedback loop by making it depend on an external token source *Next*$_I$ as well as the internal *Triggered Token Source*. While this does make the cycle time of this unit 3 traversals of the switch fabric instead of two, it also prevents deadlock.

4.0 Future Work

First of all, the documentation for the primitive SHIPs is incomplete.

Furthermore a number of composite SHIPs, such as the gateway or synchronizer SHIPs which we discussed on 10/31/05 need to be built out of primitive SHIPs (which should be possible), or perhaps specified as primitive SHIPs. Which leads directly to the problem of how to decide what functionality deserves a special SHIP. I would recommend some form of the MIPS 1% rule, or perhaps a 5% rule even: a new dedicated SHIP must show a 1% (5%) improvement in performance before it will be considered for inclusion.

And finally, in the last paragraph of the section regarding the stride SHIP, the idea of pipelined SHIPs having their performance measured in terms of switch fabric delays, or SHIP delays was mentioned, but glossed over. I think this may well be a powerful was of characterizing the performance of a design like this, it is relatively implementation independent, and it foster's a design style wherein feedback loops are kept as small as possible thereby increasing opportunities for virtualization (the interaction between virtualization and feedback loops in this style of framework is complicated, and I would recommend the reader to the literature from the SCORE project here at Berkeley, which I believe gives a good account of it).