Two FLEET Programs

Greg Gibeling
Tuesday October 24th, 2006
GDG04 – Two FLEET Programs

1.0 Introduction

In attempting to work through the problems from this homework, I found myself reminded of a series of the SHIPs we have discussed in the past: namely the "stride" SHIP which was responsible for generating stided (or simply incremented) streams of numbers. I found the lack of this abstraction among our current repertoire of SHIPs particularly painful while working through the program in section 3.2 Matrix Transpose.

While the addition of stride generation to the memory SHIP has been a powerful tool, it's disappearance as an abstract concept ensures that nested for-loops of any kind become a painful ordeal to program, let along diagram in FLEET.

As such, I advocate here for the separation of stride from memory access, in the manner described in section 2.2 Stride SHIP. I will also touch on the fetch SHIP, since we have had a number of conflicting proposals, and finally in section 3.0 Programs, I will show two example programs built using these SHIPs.

2.0 SHIPs

2.1 Fetch SHIP

This section exists not to propose an alternative fetch SHIP, as we have a wide variety from which to choose, but to state that I believe the fetch SHIP described in IES37 as the "conditional fetch SHIP," or perhaps the composite "four part fetch SHIP" are ideal.

2.2 Stride SHIP

All four of the programs described in the homework from IES40 would rely on counted for-loops in a language like C. While our memory access SHIP is exceedingly good at these kinds of operations, thanks to it's stride interface, we are left in the cold with respect to compose operations, such as nested for-loops.

To remedy this situation I propose the use of the general stride/count SHIP shown in Figure 1 below.

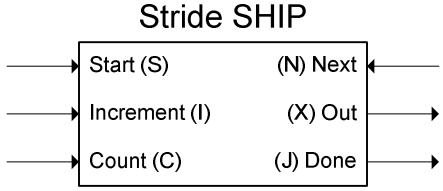

Figure 1: Stride SHIP

Upon receipt of a start, increment and count this SHIP will produce `C` numbers from `S` to `S + (C - 1)I`. No output will be generated until an `N` token is received, however. Upon receipt of an `N` token, after count numbers have been produced, the SHIP will generate a `J`. This means the environment must provide `C + 1` tokens at the `N` input, making it far easier to pipeline.

2.3 Read SHIP

In order to both simplify the memory read SHIP and show the use of the stride SHIP, figure 2, below, shows the composite use of the simple memory read interface previously documented and the stride SHIP to form the more complex, strided memory interface, previously documented.

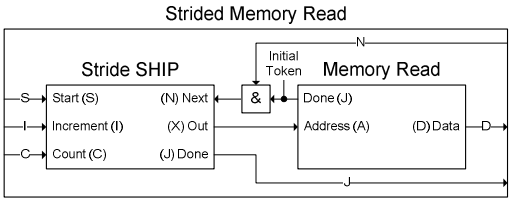

Figure 2: Strided Memory Read SHIP

2.4 Write SHIP

Similar to figure 2, figure 3 shows the composite write SHIP.

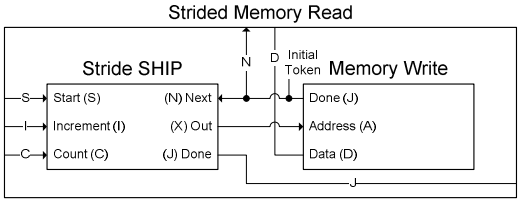

Figure 3: Strided Memory Write SHIP

3.0 Programs

3.1 Array Reversal

Reverse an array of C elements, starting at a specified base address (B), using two memory read SHIPs, two memory write SHIPs, one stride SHIP, and one simple (unconditional) fetch SHIP. The program will also require two copy SHIPs, a divide by two (shift right by one) SHIP, a subtractor SHIP, three barrier SHIPs, a token and and register SHIP.

```
Codebag Reverse {
move B -> stride.s
move 1 -> stride.i
move c -> divby2.i
move divby2.o -> stride.c

move stride.j -> barrier1.i0
move (Cleanup) -> barrier1.i1
move barrier1.o0 -> bitbucket
move barrier1.o1 -> fetch.cbd
move fetch.done -> J

move stride.x -*> copy2.i
move copy2.o0 -*> read1.a
move copy2.o1 -*> barrier2.i0
move copy2.o2 -*> sub2.i1
move read1.j -*> barrier2.i1
move barrier2.o0 -*> write2.a
move barrier2.o1 -*> bitbucket

move C -> sub1.i0
move 1 -> sub1.i1
move sub1.o -> register.i
move register.o -*> sub2.i0

move sub2.o -*> copy1.i
move copy1.o0 -*> read2.a
move copy1.o1 -*> barrier1.i0
move read2.j -*> barrier1.i1
move barrier1.o0 -*> write1.a
move barrier1.o1 -*> bitbucket

move write1.j -*> and.i0
move write2.j -*> and.i1
move and.o -*> stride.n
}

Codebag Cleanup {
move stride.x -0> bitbucket
move copy2.o0 -0> bitbucket
move copy2.o1 -0> bitbucket
move copy2.o2 -0> bitbucket
move read1.j -0> bitbucket
move barrier2.o0 -0> bitbucket
move barrier2.o1 -0> bitbucket

move register.o -0> bitbucket

move sub2.o -0> bitbucket
move copy1.o0 -0> bitbucket
move copy1.o1 -0> bitbucket
move read2.j -0> bitbucket
move barrier1.o0 -0> bitbucket
move barrier1.o1 -0> bitbucket

move write1.j -0> bitbucket
move write2.j -0> bitbucket
move and.o -0> bitbucket
}
```

3.2 Matrix Transpose

Transpose for an m*n matrix. Inputs are source address (src), destination address (dest), m (M) and n (N). Only output is done (J). For this code to run there will need to be 4 stride SHIPs, a memory read, a memory write, and a simple fetch SHIP (no conditional) all behaving as outlined in section 2.0 SHIPs. It also requires a barrier SHIP, a copy SHIP and a token-and SHIP.

The program will read the source matrix in column order, and write the destinatination matrix in row order. It will only produce a J token when the final memory write has taken place.

```
Codebag Transpose {
move src -> stride1.s
move 1 -> stride1.i
move N -> stride1.c

move dest -> stride3.s
move M -> stride 3.i
move N -> stride 3.c

move stride1.x -*> stride2.s
move stride2.j -*> stride1.n
move stride1.j -> bitbucket

move stride3.x -*> stride4.s
move stride4.j -*> stride3.n
move stride3.j -> barrier.i1

move barrier.o1 -> bitbucket
move (Cleanup) -> barrier.i0
move barrier.i0 -> fetch.cbd
move fetch.done -> J

// Notice that it is not clear
// how copies of literals are
// made, or standing moves from
// literals are cleaned up.  I
// will therefore omit their
// cleanup
move N -*> stride2.i
move M -*> stride2.c

move 1 -*> stride4.i
move M -*> stride4.c

move stride2.x -*> memread.a
move stride4.x -*> memwrite.a
move memread.d -*> memwrite.d

move memwrite.j -*> copy.i
move copy.o0 -*> stride4.n
move copy.o1 -*> and.i0
move memread.j -*> and.i1
move and.o -*> stride2.n
}

Codebag Cleanup {
move stride1.x -0> bitbucket
move stride2.j -0> bitbucket

move stride3.x -0> bitbucket
move stride4.j -0> bitbucket

move stride2.x -0> bitbucket
move stride4.x -0> bitbucket
move memread.d -0> bitbucket

move memwrite.j -0> bitbucket
move copy.o0 -0> bitbucket
move copy.o1 -0> bitbucket
move memread.j -0> bitbucket
move and.o -0> bitbucket
}
```

4.0 Conclusion

First and foremost, the separation of stride from the memory SHIPs clearly makes these kinds of strided and counted operations significantly easier to program. The reason for this is clear: the stride SHIP provides at the FLEET architecture level the same facility that simple, counted for-loops provide in higher level languages, or that repeat instructions provide in some other architectures.

The second realization is that the need to define pipeline interfaces in a uniform manner, and I advocate the design described in section 2.2 Stride SHIP, is far more clear once routines, or functions such as those in section 3.0 Programs are introduced. The key point here is that these two programs do not actually generate done tokens when they

are done, but slightly before. This may actually be slightly dangerous.

Third, many routines such as this will require a cleanup of all standing moves upon the generation of a token, and, ideally, the generation of a second token upon completion of the standing move cleanup. While this somewhat serializes the execution, the fact is that some form of serialization is necessary for virtualization of the FLEET.

Such a cleanup facility can be written at the code level, as it is here, or at the compiler level. The compiler could create the cleanup for a specific code bag by copying it, removing all non-standing moves and turning the remaining moves into `move ?? -0> bitbucket`. Or even better `move ?? -0> and.i?` thereby using a token and to generate a "cleanup done" token.

While these two options are clearly equally powerful, though using the compiler means a lot less coding, there is a third option: integrate this feature into the hardware. I do not yet endorse this as a solution, as it remains unclear how useful it will be, but I will outline the basic premise here. Either through the use of a special fetch SHIP which outputs fetched instructions as data, or through the standard memory read SHIP (in the case of first class instructions) the contents of a code bag could be sent to a special "create cleanup" SHIP. This SHIP would take a stream of move instructions as data, and for each one either simply drop it, or produce a corresponding "tear down" or "token move". The resulting token moves could be loaded into a FIFO whose output is connected to some form of barrier. The barrier's other input could be the "please clean me up" token from the original code bag. Upon receipt of this token, the barrier would then issues all of instructions in the FIFO, cleaning up the previous code bag, and possibly generating a "cleanup done" token.

Note that this scheme requires that certain SHIPs, the transformation SHIP, FIFO and fetch, be able to accept actual instructions as input, and is thus another compelling argument for first class instructions. However further discussion will be omitted from this paper.