

AM03: Standing Move Cycles

Adam Megacz
megacz@cs.berkeley.edu

September 17, 2006

This memo is an intermission in my effort to come up with a coherent, usable scheme for standing moves. While trying to write the classic "uppercase()" function using standing moves, I ran into a problem that I believe is quite common when this sort of move is in use.

0.1 Simple and Complex Moves

I would like to use the term *simple move* to refer specifically to the single-word form of the move instruction.

In a FLEET whose switch fabric supports only this instruction, every data word which moves through the switch fabric will do so as the result of a distinct instruction leaving the fetch unit. Thus the total bandwidth out of the fetch unit will be at least as great as the total bandwidth across the switch fabric. This is likely to be a performance impediment.

For this reason, several enhancements were proposed to FLEET which I will collectively term *complex moves*. These include *records*¹, *standing moves*, and *counting moves*.

1 Case Study: Summing A List of Words

As a case study, I will write a simple program to sum a list of COUNT words, starting at address 0 (for convenience).

In the code below I will use reversed arrows and indentation in an attempt to make the "causal chain" clearer. Assume that COUNT is replaced with the

¹it should be noted that of all the complex moves, only records have the advantage of configuring the switch fabric *once* for the transmission of multiple data words. This deserves much thought.

number of words to be summed. I will use the symbol `*->` to denote a standing move.

I will also make use of a slightly modified `fifo` ship which has an additional port called `release`. The modified `fifo` ship will not produce an output until it both contains a data word *and* has received a token on its `release` port.

The following block of code places the number of words to sum in the `fifo` and releases the `fifo`.

```
COUNT -> fifo.in
token -> fifo.release
```

I will also make use of two rather contrived ships:

- `iszero`, which takes a value on its `in` input and generates a token on either its `true` or `false` output depending on whether or not the input is zero.
- `decrement` ship which decrements its `input` and places the result on its `output`.

Every time the `fifo` is released, we will remove its value (the number of words remaining to be summed) and send that value to three places:

```
fifo.out *-> memread.address, iszero.in, decrement.in
```

- the `address` port of the memory read ship, which will cause the next word-to-be-summed to appear at `memread.out`
- the `iszero` ship, to check if we are done summing words
- the `decrement` ship, to generate the new count of words remaining to be summed.

The following block does the actual computation:

```
memread.data *-> adder.in1
0 -> adder.in2
adder.out *-> adder.in2
```

We “prime” the `adder` with a `0` input on its `in2` input, and thereafter loop its output back into that input. Every value that emerges from the `adder`’s output will be fed back as one of the next values to be added.

Note how we have used standing moves to wrap up a set of ships (actually only one ship in this case, the `adder`) such that they present a different interface to the outside world (in this case, an accumulator). I think this is an important role for standing moves: they can encapsulate a properly-configured set of ships in such a way that they work like a single ship.

Finally, we need to deal with the loop repetition and termination conditions. These will be determined by the token that the `iszero` ship generates.

In the event that we have *not* run out of words to sum, we want to release the next value `fifo`. That next value will be taken from the output of the decrement ship:

```
iszero.false *-> fifo.release
                fifo.in  <-* decrement.out
```

In the event that we *have* run out of words to sum, we will release a secondary `fifo` which has been configured with a codebag waiting to be delivered to the fetch unit. This codebag deconfigures all of the standing moves created by this codebag:

```
iszero.true  -> fifo2.release
                fifo2.in <- { unmove }
                fifo2.out -> fetch
```

The complete example appears below:

```
token -> fifo.release
COUNT -> fifo.in

fifo.out    *-> memread.address, iszero.in, decrement.in

memread.data *-> adder.in1
0            -> adder.in2
adder.out    *-> adder.in2

iszero.false *-> fifo.release
                fifo.in  <-* decrement.out

iszero.true  -> fifo2.release
                fifo2.in <- { unmove }
                fifo2.out -> fetch
```

2 The Problem

Unfortunately, this program has a problem. By clipping out select portions of the code, we can see that there is a *closed cycle of standing moves*

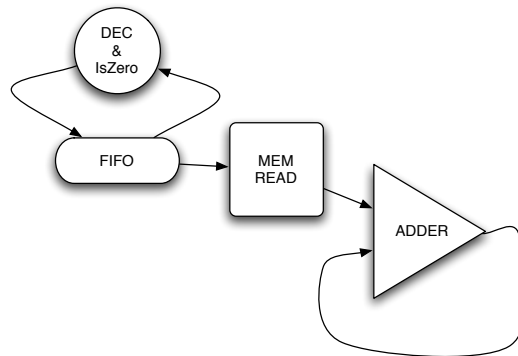
```
fifo.out    *-> ..., iszero.in, decrement.in
iszero.false *-> fifo.release
                fifo.in  <-* decrement.out
```

Notice that just these three instructions (four moves altogether) form a closed system that will rapidly generate every number from `COUNT` to `0`, *totally independent of the rate at which these values are consumed*.

This is a recipe for disaster. The switch fabric will quickly become flooded, and the `FLEET` will deadlock.

3 Solutions

A rough sketch of the data flow in this program would look something like this:



The most dangerous situation for clogging the switch fabric would be if the adder were the slowest ship in the sequence. In order to make sure that data items do not pile up without limit at the upper input to the adder, we must somehow *make the release of the fifo conditional upon the completion of the adder ship's work.*

In other words, we must break the cycle which contains only `fifo` and `decrement/iszero`, and expand it to include the entire computation.

Put briefly, our original program has flow control issues *because it contains a closed loop of standing moves, and that loop emits an unbounded number of data items to ships which are not part of the loop.* I claim that any FLEET configuration meeting these conditions is dangerous and should be avoided.

By bringing the adder's output into the smallest closed loop, we wind up with a configuration that still contains a closed loop of standing moves, but this closed loop does not emit words (or, perhaps, emits only a single word at the completion of its task). This is safe.

4 Corrected Code

The repaired program uses another fairly contrived ship called `tee`, which is similar to the UNIX `tee` command: it passes its input to its output unmodified, but generates a token on its token output every time a value passes from `tee.in` to `tee.out`.

```

token -> fifo.release
COUNT -> fifo.in

fifo.out      *-> memread.address, iszero.in, decrement.in

memread.data *-> adder.in1
0            -> adder.in2
adder.out    *-> tee.in
             tee.out      *-> adder.in2
             tee.token    *-> fifo2.release
             decrement.out *-> fifo2.in

iszero.false *-> fifo.release
             fifo.in  <-* fifo2.out

iszero.true  -> fifo2.release
             fifo2.in <- { unmove }
             fifo2.out -> fetch

```

Somehow this seems pretty clumsy to me, and I have a feeling that it only gets worse with more complex fleets.

I am still looking for ways to resolve this.

5 Cleaning Up

Much discussion has gone into the topic of cleaning up after standing moves. I think that the minimal mechanism for cleaning up after a standing move is a facility for sending an “unmove” instruction which tears down a standing move. This instruction is dispatched just like any other instruction (so it does not need to “skip ahead” of data which is queued up) and performs no buffer flushing – that is the responsibility of the programmer.

Since the “next code bag” must be ordered *temporally* after all of the unmoves have *completed* (rather than after they are simply *dispatched*), the dispatch of the “next code bag” *must be a consequence of the completion of the unmoves*. In order to facilitate this, an unmove instruction should generate a token when it completes.

Lastly, to simplify the programmer’s job, the assembler should expand an unmove command into one unmove for every standing move in the current code bag. The tokens from these unmoves should be directed to a token-counter which should wait for a number of tokens equal to the number of unmoves issued (this number is known at assembly time), at which point it should release the next codebag.