

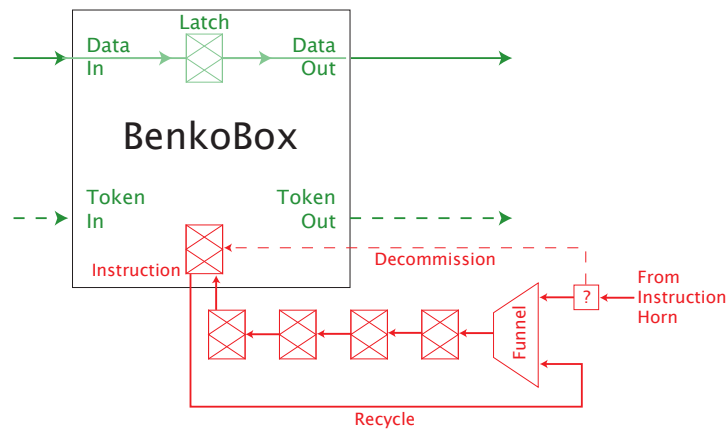
# AM15: Killing and Recycling Instructions

Adam Megacz

February 24, 2007

## Abstract:

This memo is a proposal for letting BenkoBoxes execute short *sequences* of instructions repetitively, without the involvement of the fetch unit. This is accomplished with the addition of very little hardware, and has very little impact on the design of the BenkoBox. Additionally, it simplifies the teardown of standing moves.



# 1 Motivation

Since quite early on, Fleet programmers have often expressed a desire for ship inputs and outputs (BenkoBoxes) to repeat short sequences of instructions. Currently, a *single* instruction may repeat by using the `Count` field. A sequence of instructions can repeat only via complex, high-latency interaction with a fetch unit – and this may lead to the fetch unit becoming a bottleneck.

The desire for multi-instruction sequences is especially acute in the presence of standing moves. In his Fall'06 semester project, Dominic Antonelli[1] brought this issue into sharp focus by showing how support for multi-instruction standing moves could eliminate a huge number of ships. These savings were often in the form of eliminating extraneous “duplicator” ships (which take an input and emit two copies of it) and “scatter” ships (which send a token to each of a series of ships, in a particular order).

# 2 Instruction FIFO

While writing the `balsa` code for the second FPGA version of Fleet, I noticed the need for fairly large FIFOs at the instruction port of each BenkoBox. If these FIFOs are not included, a ship which is waiting for the firing conditions of an instruction will cause any other instructions sent to it to “back up” into the instruction horn. This can potentially block off access to other ships. If a ship blocks access to some second ship, and action by the second ship is required in order to make progress at the first ship, deadlock results.

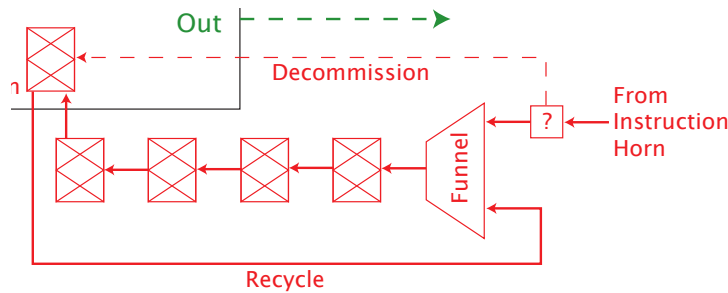
Therefore, it is reasonable to assume that Fleet will have a FIFO of instructions at the input to each BenkoBox. It is also reasonable to assume that this FIFO is at least four elements long, since the FIFO must be at least as long as the longest permissible fiber.

Noticing that Fleet must include this FIFO, let us seek ways to make additional use of it to support repetition. This can be done by adding just the following:

- One new instruction bit, `Recycle`

- One new *instruction*, Kill<sup>1</sup>.
- A single funnel element at the tail of the instruction FIFO.
- A single long state wire from the entry of the FIFO to the BenkoBox.
- A instruction-width set of wires from the BenkoBox back to the merge element.

These new elements are shown in the diagram below.



### 3 Behavior of New Bits

Normally, after an instruction finishes executing, it is discarded. We modify this slightly and instead declare that if an instruction has the `Recycle` bit set, it is instead *re-queued* at the tail of the instruction FIFO<sup>2</sup>. If the instruction has a count, it is executed *only once*, its count is decremented, and the decremented instruction is re-queued (assuming the count has not reached zero). If the instruction is a standing instruction, it is executed only once per trip through the fifo, but makes an unlimited number of trips through the fifo.

In the Fleet syntax, we represent this recycle bit by surrounding the count field with parenthesis instead of square brackets – for example, `(3)` rather than `[3]` to indicate three trips through the fifo instead of three consecutive executions at the

<sup>1</sup>Essentially no cost – one can map `Kill` onto some otherwise-invalid combination of bits, such as `Latch` without `DataIn`

<sup>2</sup>consequence: the instruction following the `Recycle` instruction may not be latched until the recycled instruction is accepted by the funnel

head of the fifo. In the case where the count field is omitted (`count=1`) the recycle bit is irrelevant, so no ambiguity results from omitting the parenthesis as well.

Example: the following instruction sequence will repeatedly take a datum from the ship and send copies of it to each of three destinations:

```
ship.outbox:
  (*) take;
  (*) sendto dest1;
  (*) sendto dest2;
  (*) sendto dest3;
```

Only two problems remain. First, we must make sure that the first instruction is not recycled until the *last* instruction in the sequence is safely inside the instruction FIFO. Secondly, we need a way to tear down these recycled loops.

## 4 Kill Instruction

The kill instruction solves both of these problems.

At the entry to the merge element lies a tiny amount of combinational circuitry (shown in the diagram as a “?”) which checks for this bit pattern on instructions which emerge from the instruction horn. If an instruction emerging from the horn is a kill instruction, then it is discarded and the Kill wire is pulled high. No new instruction is admitted from the instruction horn until the Kill wire is once again pulled low.

At the BenkoBox, *whenever an instruction is latched*, arbitration takes place between two possible events:

- The Kill wire goes high
- The firing conditions for the currently-latched instruction are met

If the first case (Kill) wins the arbitration, the instruction is not executed, and is discarded – regardless of its Recycle bit. The Kill wire is also pulled low.

If the second case (firing conditions met) wins the arbitration, the instruction is executed. If its Recycle bit is set it is re-enqueued at the tail of the FIFO; otherwise it is discarded after execution.

Note that the Kill wire is pulled low only when it actually succeeds in killing an instruction. If no instruction is presently latched, the BenkoBox continues to wait, leaving the Kill wire high.

In addition to the behaviors mentioned above, the kill instruction also carries a count field. Each instruction killed decrements the kill's count; when it reaches zero it stops killing. Fleet guarantees that no instructions will execute between the first and last instruction killed by a given kill; that is, a kill with `count=n` kills *n* consecutive instructions.

With the kill instruction in hand, we can now retract our earlier requirement that standing moves terminate as soon as an instruction arrives behind them. In place of that feature we now explicitly kill standing moves.

## 4.1 Safe Startup

Now we can safely start up the loop of instructions by placing a “dummy” instruction at the head of the FIFO which will never execute properly. Once all the instructions are safely in the fifo, we Kill the “dummy” instruction:

```
ship.outbox:
  [*] nop;           // "blocker" instruction -- "standing nop"
  (*) take;
  (*) sendto dest1;
  (*) sendto dest2;
  (*) sendto dest3;
  kill;
```

Note: now that standing moves are not automatically torn down, a “standing nop” is essentially an infinite loop which blocks the BenkoBox from further progress.

## 4.2 Teardown

Once we are done feeding data through the looping BenkoBox, we simply issue a kill instructions with count equal to the number of instructions which were in the loop. We can include an ack at the end of the sequence in order to signal that the BenkoBox has been completely reset.

```

ship.outbox:
  [*] nop;           // "blocker" instruction
  (*) take;
  (*) sendto dest1;
  (*) sendto dest2;
  (*) sendto dest3;
  kill;

TEARDOWN: {
  ship.outbox:
    [4] kill;
    ack othership.port; }

```

### 4.3 The kill\* variant

The kill instruction has one other variant, which is written as kill\*. This instruction works just as kill does, but will only kill *standing* instructions – it has no effect on instructions with a finite count.

This modified form of kill is useful when dispatching a codebag in a situation where active instructions from a previous codebag might still be in the instruction fifo. In such a situation, these instructions would necessarily be non-standing instructions, since properly cleaning up after standing instructions requires a teardown operation, and the teardown operation would certainly come before any successor codebag.

The modified form is also useful for executing operations before the first iteration of the loop, as shown in the following example:

```

ship.outbox:
  take, sendto dest5; // executes before first iteration
  [*] nop;
  (*) take;
  (*) sendto dest1;
  (*) sendto dest2;
  (*) sendto dest3;
  kill*;              // will not kill "take, sendto dest5"

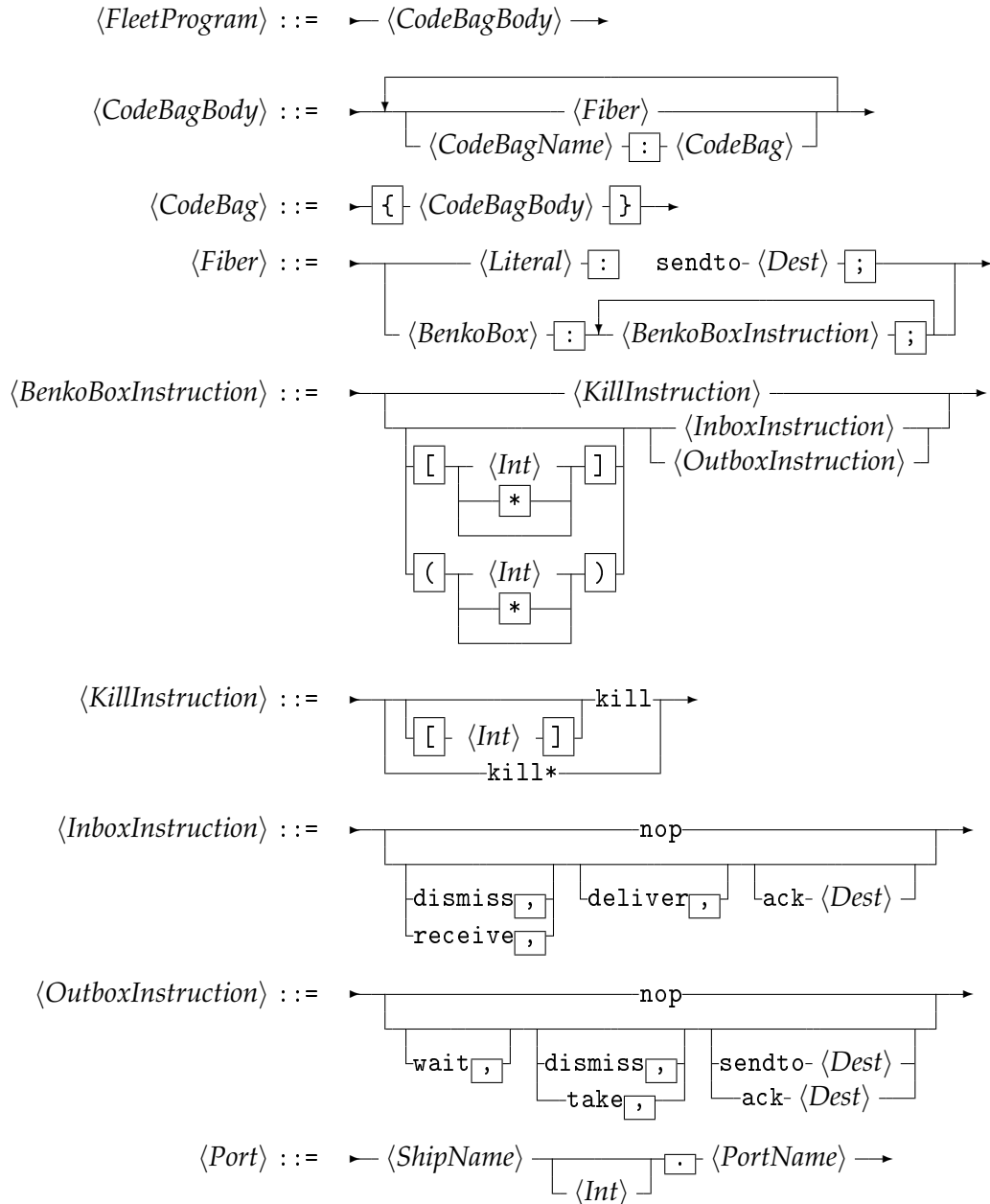
```

To avoid a very complicated semantic issue, the kill\* instruction may not have a

count field (its count is implicitly always 1). Multiple `kill*` instructions may be issued, but there is no guarantee that the instructions they kill will be contiguous.

## 5 Syntax

The updated syntax diagram is shown below.



## 6 Conclusion

By adding a fairly small amount of logic we have greatly increased the capabilities of the BenkoBox. This was accomplished largely by leveraging hardware (the FIFO) which is already required. As a fortunate side effect, the termination of standing moves has been simplified.

However, lest we forget the lessons of *The Wheel of Reincarnation*[2], we must keep focus on *why* we are increasing the power of the BenkoBoxes, and what the limit of that increase is. In particular, we chose to add instruction loops to serve two specific goals

- Goal #1: eliminate a large number of “glue” ships (mainly duplicate and scatter) and the latency required to move data to and from them
- Goal #2: for short instruction sequences, eliminate the complexity, latency, and potential bottleneck resulting from involvement of a fetch ship.

Fortunately, these two goals could be achieved at very minimal cost.

## References

- [1] Antonelli, Dominic, *CS 294-11 Final Project*, Fall 2006.
- [2] T.H. Myer, I. E. Sutherland *On the Design of Display Processors*. Communications of the ACM, Vol 11, No. 6, June 1968.