

SUPERCEDED BY AM13

AM12: Literals

Adam Megacz
megacz@cs.berkeley.edu

January 17, 2007

1 Levels of Indirection

What is a Code Bag Descriptor? Tautologically, “a code bag descriptor” is what you send to the fetch unit.

So, what do you send to the fetch unit?

So far we have assumed that the thing you send to the fetch unit is a single pointer (ie memory address-width integer) which points to a data structure which in turn points to the actual instructions in question.

I propose to eliminate this level of indirection, sending the fetch unit the actual address of the instructions to be fetched, along with any metadata (ie number of instructions, etc). Each of these items of data should be a distinct input to the fetch unit. ¹

2 Literals

There seem to be three options for literals:

1. Embed them in instructions, as in MIPS. This implies that the largest literal must be at least one bit smaller than the instruction size.
2. Put literals in separate “literal bags”
3. Partition each codebag into a “literal portion” and a “instruction portion”

¹A previous idea about “trees of codebags” has been set aside since traversing a tree of unbounded depth requires a stack of unbounded depth, and manipulating such a data structure seems to be the domain of software rather than etched-in-silicon hardware. In addition, these “code bag trees” can be emulated with little overhead.

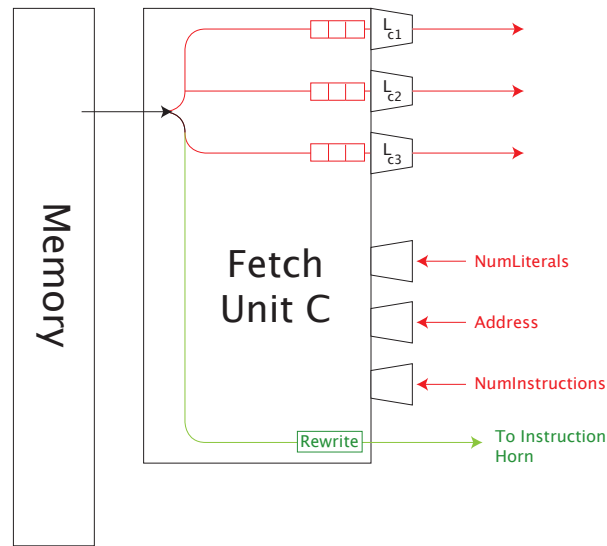
The first option has been rejected mainly because it is likely that most Fleets will deal with words (literals) which are the same size as instructions, meaning that two instructions would be required to load large literals².

The second two options have an additional advantage: they allow arbitrary regions of memory to be treated as literal bags. That is, any region of memory can be treated as a set of literals, even if that region of memory was not assembled with this role in mind.

Between the second two options, we have chosen option #3, since it subsumes option #2 in the case where the instruction portion's size is zero.

3 The Fetch Unit

The revised fetch unit diagram is shown below, assuming there are multiple fetch units, and the diagram shows the third fetch unit ("fetch unit C"):



- The Address input is a pointer to the instruction at one end of the literal portion.
- The NumLiterals field gives the width of the literal portion, which always extends from Address towards the largest address in memory.

²specifically, to load literals greater than NUMSOURCES, since the literal would occupy that portion of the instruction word

- The NumInstructions field gives the width of the instructions portion, which is always adjacent to the literals portion, and always extends towards the smallest address in of memory.

Note that in a codebag with NumLiterals=0, the Address pointer will actually point to a memory word which falls one word *outside* the codebag. This also implies that the word occupying the largest address of Fleet’s address space cannot hold a fetchable instruction.

Below is the memory layout and a sample code bag descriptor:

0xBEEF000B	garbage
0xBEEF000A	garbage
0xBEEF0009	Literal
0xBEEF0008	Literal
0xBEEF0007	Instruction4
0xBEEF0006	Instruction3
0xBEEF0005	Instruction2
0xBEEF0004	Instruction1
0xBEEF0003	Instruction0
0xBEEF0002	garbage
0xBEEF0001	garbage
0xBEEF0000	garbage

Two choices were involved in this layout:

- Which portion should the pointer point to?
- Which portion should extend towards the largest address and which towards the smallest?

In both cases we made choices that would allow a pointer to the first (lowest) word of some data block to be used as an input directly to the fetch unit, without having to decrement it or subtract NumLiterals from it. This choice was made since many pre-defined data structures adopt this convention for pointers to data blocks, and Fleet cannot alter these conventions.

On the other hand, no preexisting code or data formats are designed around Fleet instruction blocks, so having them start *one word away* from the pointer

which describes them, and having them extend towards the *smallest* address in memory is a convention which can be adopted with no cost.

Although instructions are loaded concurrently, there is a canonical ordering among the instructions in the bag which is used to determine the *source sequence guarantee* among instructions sharing a source. As shown in the diagram, if two instructions have the same source, the instruction occupying the *lesser-numbered* address will be executed *first*.

The fetch unit “fires” when data is present at all three of its inputs. When the fetch unit fires, it performs the following actions *concurrently*:

- The memory words from Address to Address+NumLiterals-1 are retrieved from memory and distributed, *round-robin* to the fifos leading to outboxes $L_{c1} \dots L_{cn}$.
- The instructions from Address-1 to Address-NumInstructions are retrieved from memory, rewritten (as explained below), and distributed to their appropriate sources.

The mapping from memory words to sources is undefined, except that:

- Memory words whose addresses are equal modulo n (where n is the number of outbox fifos on that particular fetch ship) will be placed in the same outbox fifo.
- Memory words in the same outbox fifo will be ordered such that the words from lower-numbered addresses *precede* words from higher-numbered addresses.

Although there may be multiple fetch units having principal literal outboxes $L_{a1} \dots L_{a4}$, $L_{b1} \dots L_{b3}$, $L_{c1} \dots L_{c5}$ etc, instructions are constructed as if there were only a single set of outboxes $L_1 \dots L_5$.

Instructions whose source or destination is a literal outbox L_i will be rewritten to reference L_{ci} , where c is the name of the fetch unit from which the instruction and literal were dispatched. The rewriting is performed in such a way that references to L_0 correspond to the first (lowest) memory word.

4 Serial Dispatch

Some code will want to use all of its literals for different purposes, preferring that those literals be dispatched concurrently. For these codebags, round-robin dispatch is quite appropriate.

However, there is a very common case where a literal portion contains a number of literals which are all to be processed the same way, *and must remain in this order*. Achieving this using round-robin dispatch is incredibly awkward

since the literals will be departing from different sources and may pass one another on their path through the switch fabric. The token sequencing necessary to work around this problem introduces an enormous overhead.

What we need is a way to put the fetch unit in a different mode – a mode in which all literals are placed *in the same outbox fifo*, in order.

An alternative argument would be that applications requiring a stream of data with a source-sequence guarantee ought to be using the memory read ship.

5 Other Notes

Note that it is possible to load a codebag in two separate halves – first with `NumInstructions=0, NumLiterals=X` and then with `NumInstructions=X, NumLiterals=0`. Indeed, the ability to load codebags this way raises the possibility of using literal bags as an argument-passing mechanism in procedure calls.

Fleet code which “unpacks” a network packet could easily be written by dispatching the network packet itself as a literal bag (ie `Address=(address of packet)`, `NumLiterals=(size of packet)`, `NumInstructions=0`), and then dispatching a second codebag with no literals in it which could conveniently reference fields of the packet as literals.