

UC Berkeley computer science

Subject: FLEET Assembly
Date: May 4, 2006
Authors: Igor Benko
Number: UCIB-2006-ib08

References:

- [1] Igor Benko: XML front end for ArchSim, UCIB# 2005-ib03, September 2005
- [2] Ivan Sutherland: FLEET – A One-Instruction Computer, UCIES #2055-is14, December 2005
- [3] Greg Gibeling: RAMP – Architecture, Language & Compiler, presentation at 2006 BWRC Winter Retreat; ramp.eecs.berkeley.edu
- [4] Arvind et al: RAMP – Research Accelerator for Multiple Processors, ramp.eecs.berkeley.edu
- [5] Igor Benko: ArchSim structure, UCIB #2005-ib01, September 2005

1 Introduction

This memo defines the FLEET “assembly” language interpreted by the FLEET simulator [1]. For an introduction to the FLEET architecture, please, refer to [2].

I restricted this memo to description of the FLEET assembly language. Specifically, this memo is not a guide to FLEET programming neither it provides instructions for using the FLEET simulator. I intend to address these topics in separate memos.

2 Language constructs

Below, I use examples to describe the FLEET assembly language. Section 7 provides a more formal BNF definition of the FLEET assembly language.

2.1 Program structure

A FLEET program addresses three areas:

- Assignment of names to port numbers simplifies readability of a program
- The program code, which consists of a set of code bags. Each code bag contains one or more move instructions.
- Definition of initial code that is issued to start program.

The program consists of three types of sections:

1. Declarations of imports

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley. The ideas contained herein are freely available for any academic purpose.

2. Definitions of aliases
3. Definitions of code bags

Each program file consists if instances of these three sections. Instances of sections of different types may interleave in an arbitrary order, and, in a program file, each section type is optional. Note that there may be multiple instances of each section type in a single program file. The only restriction is that an alias must be defined before it appears in code bag. This restriction allowed us to use a simple one-pass compiler and it also promotes a good programming practice.

The initial code is specified along with code bags.

2.2 Aliases

We use aliases to assign names (strings) to port numbers. Separate sets of aliases refer to sources and to destination. I chose to separate definitions of sources and destinations mostly to improve program readability and to be able to check whether a move instruction uses appropriate aliases for a source and for destinations.

An example of a definition of aliases is as follows:

```
aliases SimpleFleet {

    sources {
        alias    ConstOne.out    0
        alias    Fifo5.out        1
        alias    Fifo2.out        2
        alias    ConstTwo.out     3
    }

    destinations {
        alias    Fetch.in         0
        alias    Fifo5.in         1
        alias    Fifo2.in         2
        alias    BitBucket.in     3
    }
}
```

Alias names are case sensitive.

Note than an arbitrary name can be used as a port alias. Our convention, however, has been to name ports as "SHIP_NAME.PORT_NAME".

Inside an `aliases` construct we may have an arbitrary number of `sources` and `destination` sections, that may appear in any order. Furthermore, a program file may have multiple `aliases` sections that can interleave with code bags. An alias, however, must be defined before it can be used in a code bag.

2.3 Move instruction

FLEET has only one instruction, the move instruction. Each instruction specifies a source of data and a destination where the data is moved. During its execution, a move instruction removes a data item from the source and deposits the data item at the destination. At the moment, the FLEET assembly does not support non-destructive moves. However, multiple-destination moves described in Section 2.3.1 allow us to move copies of the same data from a source to multiple destinations.

The basic syntax of a move is straightforward:

```
move source -> destination
```

Optionally, one can terminate a move instruction with a semicolon.

```
move source -> destination;
```

I find the semicolon useful when several move instructions appear in the same line:

```
move src0 -> dest0; move src1 -> dest1
```

Note that several move instructions can appear in the same line without the semicolon:

```
move src0 -> dest0      move src1 -> dest1
```

Recall that alias strings refer source and destination addresses. A program must define the aliases before they can be used. Using an alias that has not been defined causes a compilation error.

Move instructions may also contain source and destination addresses:

```
move 10 -> 0;
```

The move instruction above will cause a data move from source 10 to destination 0. We recommend that this syntax is used for testing purposes only, because it leads to

programs that are almost impossible to understand and that cannot be mapped to another FLEET implementation by simply changing the aliases.

2.3.1 Multiple destinations

A single move instruction can move copies of data from the same destination to multiple sources. For example, to move data from one source to multiple destinations one writes

```
move source -> dest0, dest1, dest2
```

The same destination may appear multiple times in the destination list:

```
move source -> dest0, dest0
```

The move above moves two copies of the same data item to the same destination.

Currently there is no limit on the length of the destination list. Specific hardware implementations will impose their own limits.

The instructions above cause data duplication to take place inside the switch fabric. This means, that the three-destination instruction above is *not* equivalent to three move instructions:

```
move source -> dest0
move source -> dest1
move source -> dest2
```

The three instructions above do not deliver copies of the same data item to three destinations. Instead, they fetch three consecutive data items from the source and deliver them to the three destinations. Because of concurrency of move instructions, one cannot make any assumptions about which of the three instructions above would be the first to fetch the data.

2.3.2 Counting move

The counting move allows us to specify how many consecutive data items will be moved from the same source to the same set of destinations. FLEET assembly allows two syntaxes for the counting move:

```
move,3 source -> dest0, dest1
```

```
move [3] source -> dest0, dest1
```

The two lines above specify the same counting move. Namely, three consecutive data items produced at `source` will be each move to both `dest0` and `dest1`. The reason for the dual syntaxes is to allow compatibility with FLEET extensions of the RAMP toolset [3].

Note that a counting move can apply to one-destination move as well.

The counting move terminates when the specified number of data items have been sent towards all of the destinations. The counting move also terminates when the data produced by the source is an OOB (out-of-bounds) data.

2.3.3 Standing move

A standing move sends data towards a list of destinations until the data source produces an OOB data item. The standing move is a useful tool when at the time of programming we do not know how many data items will need to be moved. For example, standing moves allow us to dynamically set up a pipeline inside FLEET, moving data through a sequence of ships. Usage of standing moves requires, however, that ships produce OOB data items to terminate the standing moves that form pipelines.

For compatibility with RAMP tools, FLEET assembly supports multiple syntaxes for the standing move. The following three lines are equivalent:

```
move,standing      source -> dest0, dest1, dest2
move [standing]    source -> dest0, dest1, dest2
move [ ]           source -> dest0, dest1, dest2
```

2.4 Literals

Literals (immediates) provide us with ability to specify data at the time of programming. Currently, FLEET assembly supports only decimal integer and string literals. The syntax is as follows:

```
move (10)          -> dest0
move ("FLEET")    -> dest1
```

Counting moves can be used along with literals. However, we cannot use a standing move with a literal. With the current simulator implementation of the switch fabric and of

the literal issue mechanism, usage of a standing move with the literal would flood the switch fabric.

To specify an OOB literal, we must prefix the left brace with `oob`:

```

move oob(10)          -> dest0
move oob("FLEET")    -> dest1

```

The move instructions above specifies that a literal with the value of 3, marked as OOB, will be moved to destination `dest0`, and a literal with the value of "FLEET", marked as OOB, will be moved to destination `dest1`.

2.5 Code bags

Code bags group instructions that the FLEET issue mechanism issues concurrently into the switch fabric. That is, we have no guarantees about the order in which instructions within a code bag will be launched. The syntax for a code bag is as follows:

```

codebag bagONE {
    move (1)          -> Fifo2.in, Fifo2.in

    move ("bagFIVE") -> Fetch.in
    move ("bagTWO")  -> Fetch.in
}

```

The name of the code bag is used as a code bag descriptor. That is, sending a code bag descriptor to the Fetch-and-Issue ship causes the instructions in the bag to be fetched and issued by adding them to the instruction pool. For example, the last two move instructions above cause fetch and issue of `bagTWO` and `bagFIVE`.

Code bag names inside a program must be unique. Code bag names are case sensitive.

If an undefined code bag name is used, an error is reported during the simulation when the name of an unknown code bag is sent to Fetch-and-Issue ship. The compiler could detect such an error if the compiler was aware of the special nature of Fetch-and-Issue ship. At the moment the compiler has no awareness of the special nature of certain ships. We expect that the code-bag loading mechanism would be revised and developed further, which may allow us to improve error reporting mechanisms.

Optionally, we can mark a code bag as `initial` to indicate that the code bag will be fetched and issued upon the start of the program. A program may contain multiple `initial` code bags. We cannot make any guarantee on the order in which `initial` code bags are fetched or the order in which instructions in `initial` code bags will be issued.

The syntax for initial code bags is as follows:

```
initial codebag startBag {
    move,2 ConstOne.out -> Fetch.in
}
```

Note that the sole purpose of an `initial` code bag might be to launch other code bags, or it may also contain instructions to do the actual data movement. If the initial code bag does not launch any additional code bags, only instructions within that code bag would be executed. Such a scenario may be appropriate for situations where an initial code bag uses standing moves to set up a number of pipelines that allow the FLEET to continuously process data arriving from “input” ships.

3 Comments

A program may contain comments that follow the common syntax:

```
codebag bagONE {
    /*
     * this is a multi line comment
     */
    move (1) -> Fifo2.in, Fifo2.in

    move ("bagFIVE") -> Fetch.in
    move ("bagTWO") -> Fetch.in
    // we have single line comments, too
    // move ("bagFIVE") -> Fetch.in
}
```

4 Imports

Imports allow us to split a program into several files:

```
import SimpleFleetAliases.fleet
```

An import simply provides the name of the file to be imported. The file name path is relative to the working directory of the simulator. An optional semicolon may follow the file name:

```
import SimpleFleetAliases.fleet; // optional ;
```

A program file may contain multiple imports that are read by the compiler in the same order as listed. Note that the import mechanism is implemented in a very simple manner: The compiler reads each file and processes it according to the same rules. That is, imports allow us to break down the code into several files, but do not support any pre-compiling or other library-like techniques. The collections of aliases and code bags are built in an additive mode. Namely, any aliases or code bags defined in an import are also visible and usable in **the** program file that uses an import.

Typically, we have used imports to separate definitions of aliases from the actual code.

Our convention has been to use file extension “.fleet” to indicate that a file contains FLEET assembly code.

5 Definitions of ships

So far I have not said anything about definitions and descriptions of ships. FLEET assembly does not define the structure of a FLEET, it does not define what ships exist in a FLEET, and it does not define functionality and behaviors of individual ships. The FLEET simulator has a separate XML file that defines the ships and the switch fabric used in s FLEET. A separate XML file, often called a library, records interfaces of individual ships, and we use Java to define the behavior of a ship or a switch fabric component. Please, refer to [1] for more details on configuration of the FLEET simulator.

6 Example

To provide further illustration of FLEET assembly syntax, we give a simple example of a FLEET program. The program consists of two files, one defining aliases, and another defining code bags. The operation of the program is simple: Literals are first moved to

two FIFOs, which are then drained. Concurrency effects may cause drainage to take place before the FIFOs are filled with data.

Listing of SimpleFleetAliases.fleet:

```
aliases SimpleFleet {
    sources {
        alias    ConstOne.out    0
        alias    Fifo5.out       1
        alias    Fifo2.out       2
        alias    ConstTwo.out    3
    }

    destinations {
        alias    Fetch.in       0
        alias    Fifo5.in       1
        alias    Fifo2.in       2
        alias    BitBucket.in   3
    }
}
```

Content of MoveTest.fleet

```
import SimpleFleetAliases.fleet

initial codebag startBag {
    // Fetch bagOne
    // ConstOne.out produce string "bagONE"
    move,2 ConstOne.out -> Fetch.in
}
```

```

codebag bagONE {
    // move two literals to Fifo2.in
    move,2 (1) -> Fifo2.in
    // fetch two code bags
    move ("bagFIVE") -> Fetch.in
    move ("bagTWO") -> Fetch.in
}

codebag bagTWO {
    // move two literals to Fifo5.in
    move,2 (2) -> Fifo5.in, Fifo5.in
    // fetch bagTHREE
    move ("bagTHREE") -> Fetch.in
}

codebag bagTHREE {
    // drain Fifo5
    move,standing Fifo5.out -> BitBucket.in
    // drain Fifo2
    move Fifo2.out -> BitBucket.in
    move Fifo2.out -> BitBucket.in
}

codebag bagFIVE {
    // kill the standing move
    move oob(3) -> Fifo5.in
}

```

7 Grammar

In the section we give a BNF grammar for the syntax of a FLEET program file. First, we review BNF notation and define extensions we use.

7.1 Review and extensions to BNF

<i>Symbol</i>	<i>Meaning</i>
::=	is defined as
	OR
<name>	Non-terminal
[item]	optional item
{ item }	Zero or more repetitions of the item
“character”, e.g. “[A single-character terminal

7.2 FLEET assembly grammar

The grammar below applies to a single file with FLEET assembly. For the sake of simplicity, the grammar below does not account for code comments.

<program> ::= { <import> | <aliases> | <codebag> }

<import> ::= **import** <name>

<aliases> ::= **aliases** [<name>] “{ { <sources> | <destinations> } }”

<sources> ::= **sources** “{ { <aliasdef> } }”

<destinations> ::= **destinations** “{ { <aliasdef> } }”

<aliasdef> ::= { **alias** <name> <number> }

<codebag> ::= [**initial**] **codebag** <name> “{ { <move> } }”

<move> ::= <move_c> | <move_b>

<move_c> ::= **move** [“,” (<number> | **standing**)] <source> -> <destinations>

<move_b> ::= **move** [“[“ (<number> | **standby** | “”) “]”] <source> -> <destinations>

<source> ::= <name> | <number> | <literal>
 <destinations> ::= <destination> { “,” <destination> }
 <destination> ::= <name> | <number>
 <literal> ::= “(“ <literal_string> | <number> “)”
 <literal_string> ::= “\”” <symbols> “\”” | <name>
 <number> ::= [“-“] <digit> { <digit> }
 <name> ::= { <symbol> | <digit> } <symbol> { <symbol> | <digit> }
 <digit> ::= “0” | ... | “9”
 <symbol> ::= “a” | ... | “z” | “A” | ... | “Z” | “-“ | “_” | “.” | “:” | “\” | “/”
 <symbols> ::= all characters including white space