From: Eric Bush

To: MTB Distribution

Date: November 4, 1980

Subject: Multiple instruction sets (decors) for the ALM assembler

This MTB describes a new feature of the ALM assembler that will enable it to generate code for any of several incompatible machines, and to verify that a user's program is compatible with the machine and mode(eg., non-privileged, privileged, hyper)for which it is intended. In particular, it describes the operation and assumptions of alm_table_tool, a program that allows the ALM maintainer to automatically implement additions, deletions, or changes to the universe of instruction sets currently recognizd by ALM.

DECORS:

The problem of multiple instruction sets is not simply one of multiple machines. Among the instructions valid on a given machine are certain subsets that can only be executed when special conditions obtain. For example, on the Orion processor, programs running in rings > 0 cannot execute privileged instructions, and programs not in hyper mode cannot execute hyper instructions. The smallest unit of program/machine compatibility, however, is not "machine X in mode Y" either, because programs that choose their instructions carefully can run on any machine in any mode.

Thus we choose as the basic unit of program/machine compatibility the "decor", which we define as a subset of the set of all instructions available on any machine that we support. A single decor may thus contain instructions spanning several machines. Decors may be defined simply by enumerating instructions from this superset. Presumably, though, the useful decors will be those specifying the valid instructions for a given machine and mode. A program is compatibile with a decor if all of the instructions that it uses are members of that decor.

At this writing the assembler has been coded to recognize six decors:

1) The set of non-privileged instruction available on L68, 6180, and DPS8 machines.
2) The privileged and non-privileged instructions on L68 and 6180.
3) The privileged and non-privileged instructions on DPS8.
4) The non-privileged instructions on ORION.
5) The privileged and non-privileged ORIONs.
6) The instructions available in ORION hyper-mode.

The assembler can be automatically recoded to accept any desired partitioning of the total instruction set into decors by the use of alm_table_tool, a special utility program to be described below. Future generations can cut up the universe differently, as new processors and modes come along, or as they realize this not to be the best partitioning, by using this tool.

USER VISIBLE FEATURES:

Each ALM program will be considered to be written for a particular decor. The user will communicate his/her intended decor to the assembler by the use of the "decor" pseudo_op. It will take one operand: the name of the decor. The current operand names are:

1) L68,6180,DPS8
2) L68p,6180p
3) DPS8pn
4) ORION
5) ORIONp
6) ORIONh

The assembler can be automatically recoded to accept different names by using alm_table_tool (see below). This is particularly important for the "ORION problem" (see below).

Since, unlike a compiler, the assembler must generate exactly the machine instructions specified by the user, its management of multiple decors is limited to determining that the instructions written by the user are compatible with the specified decor. Any incompatibility sustains a "B" error. Since the user can force translation simply by changing the operand, the assembler guaratees only that the decor of the program is properly documented (via the operand), not that it will run.

If no decor pseudo_op is specified, then the assembler will assume the first decor: L68/6180/DPS8 non_privileged.

IMPLEMENTATION STRATEGY:

The predecessor of decors, the distinction between GE 645 and 6180 instruction sets, was coded in the last 4 bits of instruction description words in a table in oplook_.alm. The present implementation uses these 4 bits to represent the unique intersections of available decors. We call these decor classes. Decor class 2, for instance, currently designates the intersection of ORION, ORIONp, and ORIONh decors. Thus all the instructions that are unique to an Orion processor have decor class 2. (In general, one should expect that N decors will yield at least N unique intersections. Thus if future maintainers find themselves approaching 16 decors they should consider expanding this field).

When the assembler is first invoked, it initializes a decor variable to zero. If the decor pseudo_op is specified by the user, it is picked up in pass 1 and the decor variable is set to a value greater than zero corresponding to the operand given. In pass 2, as each instruction is processed, its decor class is used as a row index, and the current value of the decor variable as a column index, into a bit table which tells which classes are compatible with which decors.

ALM_TABLE_TOOL:

This section describes the operation and current assumptions of alm_table_tool, a utility program bound into bound_alm_.

Alm_table_tool takes as input two include files, a table of instructions X decors (DECOR_TABLE.incl.pl1) and a list of opcode defining ALM macros (defops.incl.alm) from oplook_.alm, and produces as output a new version of defops.incl.alm and two external static data structures: one of the bit table referred to above, and the other of an operand name table used to assign numeric codes to the operands. The bit table is referenced by pass2_.pl1 and the name table by pass1_.pl1. Alm_table_tool determines the unique intersections of decors from the table of instructions X decors, assigns them numbers, codes these numbers into the ALM macros for each instruction, determines the numeric code for each operand to the decor pseudo_op, and determines the dimensions and bit pattern of the resultant decor class X decor table.

Alm_table_tool is currently implemented as a command.

Usage: alm_table_tool PATH1 PATH2

where PATH1 is DECOR_TABLE.incl.pl1
and
PATH2 is defops.incl.alm

The table of instructions X decors and the list of defop macros must conform to certain standards to be accepted by alm_table_tool. Since there are already current versions of both, one simple way to avoid running afoul of these standards is to make changes to the existing versions consistent with their current form. At this writing, the instructions X decors table is a huge PL1 comment in alm_table_tool itself. The defop macros are part of oplook_.alm's source code (via include file).

Assumptions about instructions/decor table:

A piece of the table appears in appendix A. Alm_table_tool assumes that the table consists of two parts, the first preceded by the keyword "NAMES:" and the second preceded by the keyword "TABLE:". It assumes that the names section consists of a series of definitions separated by whitespace. Each definition consists of a dummy name (any character string except "table") followed immediately by a colon, and a series of synonyms (separated by whitespace and terminated by a semicolon). In the table section, there must be one column for each dummy name in the names section, headed by that dummy name. Alm_table_tool uses the dummy names only to coordinate synonyms with table columns. Any name that is to be used as an operand to the decor pseudo-opshould be included as a synonym to some dummy name in the names section. Alm_table_tool assumes that there is a "¦" delimiter between each column header and one after the last header. It assumes the "----------" boundary follows. For each row, it assumes a "¦" delimiter after the instruction name, one between each row/column intersection, and one at the end of the row. (Just the kind of thing you'd expect). If a given instruction is not in a given decor, then whitespace should appear at the intersection of the instruction's row and decor's column on the table, otherwise an "X" should appear (alm_table_tool will also accept "x"). Alm_table_tool assumes that a row terminates with a new_line character.

Assumptions about defops:

Alm_table_tool assumes that "defop" is the name of the macro, that there are no spaces between members of the operand list, and that the last operand of every defop denotes decor

class. If one finds it desirable to alter any of these features of the macro, one should also alter alm_table_tool to handle the change. Alm_table_tool also assumes that the defop segment it receives as input has nothing else but defop macros in it.

Alm_table_tool's output:

Alm_table_tool writes new versions of defops.incl.alm, alm_data1, and alm_data2 in the working directory. Oplook_.alm must be reassembled to incorporate the new defop macros. Alm_data1 and alm_data2, which are referenced by pass1_.pl1 and pass2_.pl1 respectively, need merely be replaced in bound_alm by the new versions. Pass1_.pl1 and pass2_.pl1 need not be recompiled.

THE "ORION PROBLEM":

The "ORION problem" is the following. We are now in the process of writing software for a processor which we temporarily call ORION. At some point in the future, ORION hardware will acquire its "real" name ("Fred", let us say). "Fred" is perhaps the only name by which many of our users will know this hardware. A decor named "ORION" is thus meaningless to them. When "Fred" comes along, we will need to provide our users with FRED, FREDp, and FREDh operands to the decor pseudo_op. We, however, who must write code for FRED processors now, have to use the only name we have now viz, "ORION". How do we avoid the seeming (error prone) necessity of finding and rewriting all of our "ORION code" when we officially canonize the FRED operands?

The answer is to keep our ORION operands, but add three synonyms for them at FRED-time. The assembler will then accept either ORION or FRED. This redundancy can be accomplished merely by adding FRED, FREDp, and FREDh as respective synonyms in the names section.

```
/*  NAMES: A: L68 6180 DPS8;
           B: L68p 6180p;
           C: ORION;
           D: ORIONp;
           E: ORIONh;
           F: DPS8p;
```

TABLE:

	A	B	C	D	E	F
a4bd	X	X	X	X	X	X
a4bdx	X	X	X	X	X	X
a6bd	X	X	X	X	X	X
a6bdx	X	X	X	X	X	X
a9bd	X	X	X	X	X	X
a9bdx	X	X	X	X	X	X
aar	X	X	X	X	X	X
aar0	X	X	X	X	X	X
aar1	X	X	X	X	X	X
aar2	X	X	X	X	X	X
aar3	X	X	X	X	X	X
aar4	X	X	X	X	X	X
aar5	X	X	X	X	X	X