M E M O

To: ALTO GROUP

Date: August 8, 1974

From: Chuck Thacker

Location: Palo Alto

Subject: Alto Microassembler

Organization: PARC/CSL

File: CTALTOMICROASSEMBLER
Archive category: Alto

This document describes the source language and operation of MU, the Alto microassembler. MU is downward compatible with DEBAL, the original Alto assembler/debugger, but has a number of additional features. MU is implemented in BCPL, and runs on the Alto.[1]

The source language

An Alto microprogram consists of a number of statements and comments. Statements are terminated by semicolons, and everything between the semicolon and the next CR is treated as a comment. Statements can thus span several text lines (the current limit is 256 characters maximum). All other control characters and blanks are ignored.

Statements are of three basic types: declarations, address predefinitions, and executable code. The syntax and semantics of these constructs is as follows:

Declarations

Declarations are of three types: symbol definitions, constant definitions, and R memory names.

Symbol Definitions

Symbol definitions have the form:

$name$Ln$_1$,n$_2$,n$_3$;

The symbol "name" is defined, with values n_1, n_2 and n_3. There is a standard package of symbols for the Alto which should appear at the beginning of every source program. For those who must add symbol definitions, the interpretation of the n's is given in the appendix.

Constant declarations

Normal constants are declared thusly:

$name$n;

---

[1] Because the symbol table is allocated above the operating system, MU will currently run only on 64K Altos.

This declares a 16 bit unsigned constant with value n. The assembler assigns the constant to the first free location in the constant memory, unless the value has appeared before under another name in which case the value of the name is the address of the previously declared constant. An alternative constant definition is used for mask constants which have a specified bus source field (recall that the constant memory address is the concatination of the rselect and bus source fields of the microinstruction). The syntax is:

$name$Mn:v; 4≤n≤7, 0≤v<2**16

N specifies the desired bus source value, v is the constant value.

R Memory declarations

R memory names are defined with:

$name$Rn; 0≤n<40B
(100B if you have an Alto augmented with 32 extra R registers)

An R location may have several names.

Address predefinitions

Address predefinitions allow groups of instructions to be placed in specified locations in the control memory, as is required by the OR branding scheme used in the Alto. Their syntax is:

!n,k,name0,name1,name2,···,namek-1;

This declaration causes a block of k consecutive locations to be allocated in the instruction memory, and the names assigned to them. n defines the location of the block, in that if L is the address of the last location of the block, L and n = n. Usually, n will be 2**p-1 for some small p. For example, if the predefinition

!3,4,foo0,foo1,foo2,foo3;

is encountered in the source text before any executable statements, the labels foo0-foo3 will be assigned to control memory locations 0-3. If there are too few names, they are assigned to the low addresses in the block. If there are too many, they are discarded, and an error is indicated. If there are missing labels, e.g. "foo0,,foo2,;;", the locations remain available for the normal instruction allocation process. A predefinition must be the first mention of the name in the source text (forward references or labels encountered before a predefinition of a given name cause an error when the predefinition is encountered.)

Executable statements

Executable code statements consist of an optional label followed by a number of clauses separated by commas, and terminated with a semi-colon

label: clause,clause,clause···;

If a label has been predefined, the instruction is placed at the control memory locaion reserved for it. Otherwise, it is assigned to the lowest unused location.

Clauses are of three types: gotos, nondata functions, and assignments.

GOTO

Goto clauses are of the form :label, and cause the value of the label to be assembled into the NEXT field of the instruction. If the label is undefined, a chain of forward references is constructed which will be fixed up when the symbol is encountered as a label.

Nondata Functions

Nondata functions must be defined (by a literal symbol definition) before being encountered in a code clause. This type of clause assembles into the F1,2, or 3 fields, and represents either a branch condition or a control function (e.g. BUS=0, TASK).

Register transfers (assignments)

All register transfers are specified by assignments of the form:

rdesta←rdestb←'''←source

This type of clause is assembled by looking up the rdests, checking their legality, and making the field assignments implied by the symbol types. Each destination imposes definitional requirements on the source (e.g. ALUOUTPUT must be defined, BUS must be defined).. These requirements must be satisfied by the source for he statement to be legal. When the source is encountered, it is looked up in the symbol table. If it is legal and satisfies the definitional requirements imposed by the destinations, the necessary field assignments are made, and processing continues. If the entire source defines the BUS, and the only remaining requirement is that the ALU output must be defined (e.g. L←MD), the ALUF field is set to 0 (ALU OUTPUT = BUS), and processing continues. If neither of the above conditions holds, the source can legally be only of the form: "thing which defines the bus ' alu function." The source taken is broken into two substrings, and each is looked up in the symbol table. If two substrings can be found which satisfy the requirements, the field assignments implied by both are made. If the boundary between the two substrings is advanced to the end of the token without definition, an error is generated. This method of evaluation is simple, but it has pitfalls. For instance, L← T+2 is illegal, but L←2+T is not, providing that the constant "2" has been defined.

The constant "0" is special, in that when one or more clauses in a statement reuire that the bus be 0, the constant is not output, but a flag is set. When processing of the statement is completed, if any clause has caused the R memory to be loaded, the constant is not used, since the hardware forces the bus to 0 in this case.

The destination "SINK" allows a clause to specify a bus source without specification of a destination. It is useful, for example in constructs of the form: SINK←MASK CONSTANT, L←DISP XOR T,, which will cause the value of DISP to be anded on the bus with the mask constant

Operation

The assembler is invoked with:

MU/global switches sourcefile.optional extension listfile/L binfile/B

Legal global switches are:

/L produce a listing file
/D debug mode
/N do not produce a binary file (overridden by binfile/B)

If listfile/L is absent but the /L global switch is set, listing output will be sent to sourcefile.LS.

If binfile/B is absent, binary output is sent to sourcefile.MB.

Error messages will be sent to the listing file if one has been specified, unless debug mode has been set. In debug mode, errors are sent to the system display area, and a pause occurs at at every error (and at certain other times). Typing any character proceeds.

If no listing file has been requested, debug mode is set independent of the global switch.

Output file

The assembler produceds MICRO format binary output. The string names of the two memories specified in the file are CONSTANT and INSTRUCTION. Only defined locations in these memories are output. MICRO format is compatible with the PROM blowing program, and with RAMLOAD, a description of which is attached. Note that the instruction memory specified in the binary file does not include the 3 bit F3 field, which exists only in the debugging RAM.

Listing file

The listing file contains:

1.) All error messages (unless debug mode is set)

2.) A listing of all unused but predefined locations and unresolved forward references.

3.) A listing of the contents of the constant memory

4.) A listing of the names assigned to the R memory

5.) A listing of the object and source code (with comments and declarations removed. The 35 bit instruction is printed out in the following order:

LOCATION: RSEL ALUF BUS SOURCE F1 F2 LOADL LOADT F3

Appendix I: Literal symbol definitions

The value of a symbol is a 3 word quantity. The first word contains a type (6 bits) and a value (10 bits) which detemines the interpretation of the symbol in all cases except when it is encountered as the source in a register transfer clause (assignment). The second word contains the type and value used in this case. The third word contains bits specifying the definitional requirements and source attributes applied when the symbol is encountered in an assignment:

BIT0: L OUTPUT MUST BE DEFINED'	(USED BY LHS PROCESSING)
BIT1: BUS MUST BE DEFINED'	(LHS)
BIT2: ALM MUST BE DEFINED'	(LHS)
BITS 3-7: 0	
BIT8: L IS DEFINED	(USED BY RHS PROCESSING TO CHECK LEGALITY OF CLAUSE
BIT9: BUS IS DEFINED	(RHS)
BIT10: ALU IS DEFINED	(RHS)
BIT14: ALU OUTPUT IS DEFINED IF BUS IS DEFINED	(USED BY RHS PROCESSING WHEN SOURCE IS OF THE FORM "BUS DEFINER.ALU FUNCTION")

Assignment processing proceeds by successively ORing the attribute words for the destinations with -1. When the RHS is encountered, the attribute word contains 0'S in bits 0-2 for things which must be defined. Legality of the source (if it is a defined symbol) is tested by computing

DESTINATION ATTRIBUTES (0-2)' AND SOURCE ATTRIBUTES (8-10)

The 0 bits in bits 0-2 of the result represent unsatisfied conditions. If the only requirement is ALU definition, and if the BUS is defined, the ALU function is set to gate the bus through (defining the ALU), and the clause is complete. If this doesn't work, the source string is dismembered, looking for two substrings, the first of which defines the bus (bit9), and the second of which defines the ALU output if the bus is defined (bit14). If two substrings are found, the implied assignments are made, and the clause is complete. Otherwise, an error is indicated.

The symbol type(s) determine the fields to be set in the microinstruction: Some types are legal only as an isolated clause, some are legal only as the source or destination in an assignment. The currently defined types are:

TYPE:		LEGAL AS:	INSTRUCTION FIELD RECEIVING VALUE	SIDE EFFECTS
0	ILLEGAL	NEVER		
1	UNDEFINED ADDRESS	ADDRESS		
2	DEFINED ADDRESS	ADDRESS	NEXT	
3	R LOCATION LHS	LHS	RSEL	
4	R LOCATION RHS	RHS	RSEL	
5	←CONSTANT	RHS	RSEL,BS	
6	BUS SOURCE	RHS	BS	
7	NONDATA F1	CLAUSE	F1	
10	DATAF1←	LHS	F1	
11	←L DEFINING F1	RHS	F1	
12	NONDATA F2	CLAUSE	F2	
13	DATA F2←	LHS	F2	
14	←DATA F2	RHS	F2	BS←1,BWBZ←TRUE,RSEL←0 (←DNS,←ACDEST)
15	DATAF2←	LHS	F2	BS←0,RSEL←0

16	END	CLAUSE	-	(ACDEST←, [illegible]RCE←) KEEPGL [illegible] (STOREFLAG←FALSE)
17	←L	RHS	-	
20	L←	LHS	LOADL	
21	NONDATA F3	CLAUSE	F3	
22	DF3←	LHS	F3	
23	←DF3	RHS	F3	
24	←ALU FUNCTIONS	RHS	ALUF	
25	T←	LHS	LOADT	
26	←T	RHS	ALUF	SETS ALUF TO VALUE
27	NO LONGER USED			
30	PREDEFINED ADDRESS			
31	LMRSH,LMLSH	RHS		
32	←MASK CONSTANT	RHS		
33	←F2	RHS	F2	BS←2
34	←F1	RHS	F1	BS←2

The current symbol definitions are attached.