IST 1.0
The Instruction Set Testbed

Rosalee Nerheim-Wolfe

No. 92-001

Department of Computer Science
and Information Systems
DePaul University
Chicago, IL 60604

Abstract

IST (Instruction Set Testbed) provides a method of comparing instruction set architectures without the need to write any of the support software. IST's table-driven assembler uses a student-supplied architecture definition to assemble programs. IST's interactive simulator includes facilities to aid debugging and to gather performance statistics. The simulator has access to the architecture definition, so the simulator performs symbolic debugging. The simulator is based on Tanenbaum's microarchitecture, but allows the student to run at either the microcode level ("level-1") or at the conventional machine level ("level-2"). The conventional machine architecture is specified by the user-supplied architecture definition.

IST is written in C, and runs on a variety of Unix platforms.

Contents

1 Introduction

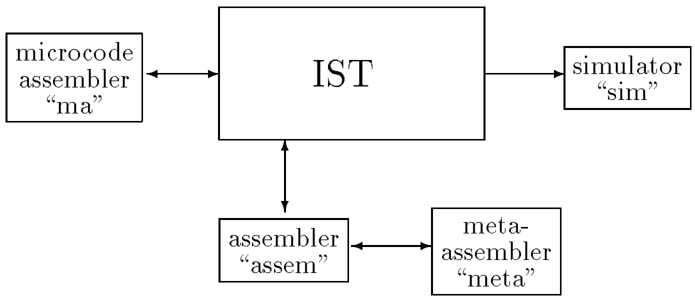
IST facilitates the development and performance evaluation of instruction sets. The user provides IST with an instruction set definition, a microcode implementation of that instruction set, and a program written in the newly-defined assembly language. IST provides the user with a powerful symbolic debugger, and a set of performance measures.

An advantage of IST is the ease of defining symbolic assembly languages without having to write the necessary assembler. Another advantage of IST is its symbolic orientation. All code, be it a program in the newly-defined assembly language or a microcode program, is written and debugged symbolically.

IST fosters innovation in the development of new instruction-set architectures and encourages the comparision of architectural designs.

1.1 The Components of IST

IST is the front end for a microcode simulator. It assembles both the user's assembly language program and microprogram. If both programs are assembled successfully, it then runs the microcode simulator. IST uses a microcode assembler called "ma" to convert the symbolic microcode into the binary code for the microarchitecture control store. IST provides an assembler to assemble the user's assembly-language programs.



1.2 Using Components separately

Each of the components – the microcode assember, the regular assembler, and the simulator – can be run separately from IST. See the sections of this document relating to the microcode assembler, the regular assembler, and the simulator.

2 Running IST

At the prompt, the user types

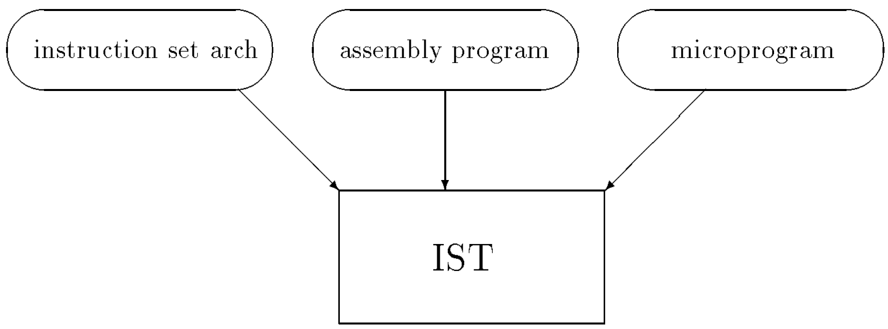
```
ist
```

IST prompts the user for

1. a file containing a definition of an instruction-set architecture. An instruction-set architecture refers to the instructions, the instruction format, and that part of the architecture that is visible to an assembly-language programmer.

2. a source file of a program written in the symbolic assembly language as defined in the instruction set architecture

3. a source file for the microcode implementation.



IST checks the time stamps of the programs and assembles the assembly program and compiles the microcode, if necessary. If the programs are syntactically correct, IST loads the assembled program into the simulator's main memory, loads the microcode into the control store, and hands off execution to the system's simulator. At this point, the user is in the simulator and is free to run the program, collect performance statistics, dump or set memory locations, etc.

The following is the beginning of a session. What the user types is written in **boldface**.

```
% ist
instruction set?one.addr
assembly program?testnew.asm
microcode program?control.mal
testnew.mac is up to date
control.mic is up to date
*** Welcome to the Instruction Set Testbed (IST) ****
control.mic read successfully.  program length:  79
testnew.mac read successfully.  program length:  6
asm:
```

By typing the architecture-definition, assembly and microcode file names on the command line, the user can skip the initial prompts.

3 The architecture simulator: sim

3.1 Purpose

The program "sim" allows users to simulate architectures for any assembly language with the single restriction that the architecture be implemented with Tanenbaum's microarchitecture.

Users can choose to work at the microprogramming level (Tanenbaum's "level-1") or at the assembly language level (T's "level-2"). At microprogramming level, the user's commands refer to the microarchitecture's control store and registers. When the user decides to simulate the assembly language machine, commands refer to an assembly-language's memory and registers. When running at the assembly-language level, all microarchitecture details are concealed from the user.

3.2 Running the simulator

Normally, a user runs the simulator via the Instruction Set Testbed (IST). However, if the user has an assembled program stored on a file named <filename1>.mac and has a compiled microcode program stored on a file named <filename2>.mic, then the user can run sim directly. At the unix prompt, the user types:

```
sim  <filename1>.mac  <filename2>.mic
```

```
The next thing the user sees on the screen is
  **** Welcome to the Instruction Set Testbed (IST) ****
<filename2>.mic read successfully.  program length: xx
<filename1>.mac read successfully.  program length: yy
asm:
```

If the user forgets to name the ".mic" and ".mac" files at the command line, the simulator will search for the files `tanen.mic` and `loadst.mac` in the current directory. If simulator cannot find these files, it terminates.

3.3 Command set summary

Detailed explanations for each command follow the summary.

I. Switching between microarchitecture level and assembly level
   (a) micro
   (b) asm

II. Loading memory, examining and setting registers
   (a) load <pathname>
   (b) examine <register name>
   (c) examine <address>
   (d) examine <address1> - <address2>
   (e) set <register name> <value>
   (f) set <address> <value> (level 2 only)

III. Setting breakpoints and tracing registers
   (a) breakpoint <address>
   (b) trace <register name>
   (c) untrace <register name>
   (d) detail <detail amount>

IV. Executing a program
   (a) step
   (b) go

V. Other commands
   (a) stats
   (b) help

(c) shell escape

(d) quit

All commands can be shortened to their initial three letters. “sim” is case-insensitive.

3.3.1 micro

The command **micro** switches the simulator to the microarchitecture (level-1) level. As a reminder to the user that sim is running at level-1, the prompt changes to

```
micro:
```

3.3.2 asm

The command **asm** switches to the assembly (level-2) level. The prompt when running at level-2 is

```
asm:
```

Switching from level-1 to level-2 should only be done when the mpc is zero. Switching at any other time will leave a level-2 instruction partially executed.

3.3.3 load <pathname>

The **load** loads a compiled program into memory. <pathname> is a unix pathname.

If running at level-1, <pathname> contains compiled microassembly language (“mal”) code, and is loaded into the control store. <pathname> must end in “`.mic`”.

If running at level-2, <pathname> contains compiled assembly language, and is loaded into the level-2 memory. <pathname> must end in “`.mac`”.

The loaded program must be compiled prior to running sim. To compile mal programs, escape to the shell (or exit from sim) and type:

```
ma <pathname>
```

where <pathname> contains a microassembly language program. The program searches for a file whose name is <pathname> followed by “.mal” It compiles program and places the object on <pathname>.mic.

To assemble assembly language programs, use a shell escape (or exit from sim) and type:

```
assem <arch-defn> <pathname>
```

where <arch-defn> is the pathname of the file that contains the definition of the instruction-set architecture, and <pathname>.asm contains an assembly language program. The object is stored on <pathname>.mac

3.3.4 examine

The **examine** command takes three forms.

```
examine <register name>
examine <address>
examine <address1> - <address2>
```

The **examine** command prints the contents of the specified register or addresses.

The <register name> must refer to a valid register name. At the microarchitecture level, valid register names include mar, mbr, mpc (micro pc), mir (micro instruction register), alatch, blatch, pc, ac, sp, ir, tir, zero, one, neg1, amask, smask, a, b, c, d, e, and f. The simulator prints the contents of the registers as a binary string except for mir. The contents of the mir are printed in symbolic mal for legibility.

When at the assembly-level, register names depend on the assembly language. For example, valid register names for Tanenbaum's one-address assembly language are pc, ac, and sp.

When examining memory, a user can specify a single <address> or a range from <address1> to <address2>. Sim prints the contents of the control store in symbolic microcode for legibility.

3.3.5 set

The **set** command has two forms:

```
set <register name> <value>
set <address> <value>
```

The **set** command changes the chosen register (or address) to the specified value. Specifying an address is valid only when simulating the assembly (level-2) level.

The <register name> must be a valid register name. The <value> can be specified in decimal or hex (0x—). The <address> should be decimal.

Notice: Users cannot change values in the control store via the **set** command.

3.3.6 breakpoint

The **breakpoint** command has the following form.

```
breakpoint <address>
```

The **breakpoint** command specifies a breakpoint. After specifying a breakpoint, the user can type the command "go", and execution will stop when the pc (or mpc) contains the memory address given by the breakpoint command.

3.3.7 trace, untrace, detail

The **trace** command has the following form.

```
trace <register name>
```

The **trace** command prints the specified register after each instruction is executed.

Suppose the user is at the assembly level. S/he types

```
trace pc
go
```

This will cause the contents of the pc to be printed after each instruction is executed.

```
untrace <register name>
```

The **untrace** command stops the tracing of the specified register.

```
detail <detail amount>
```

It's tedious to specify individual registers with the "trace" command. The "detail" command allows a user a quick way to turn on a lot of tracing.

At the microarchitecture level, the <detail amount> ranges from 0 to 4:

detail amount	action taken
0	trace no registers
1	trace mpc, ir, and tir
2	trace mar, mbr, ac, mir, mpc, ir, and tir
3	trace a,b,c, mar, mbr, ac, mir mpc, ir, and tir
4	trace a,b,c,d,e,f,mar,mbr,ac, mir,mpc,ir,tir, alatch and blatch

At the assembly level, the <detail amount> ranges from 0 to 1:

detail amount	action taken
0	trace no registers
1	trace pc, ac, and sp

If using "detail" and "trace" together, use the "detail" command first. Using "detail" after "trace" may contradict the "trace" command otherwise.

3.3.8 step, go

The **step** command executes one instruction.

The **go** commences program execution. Execution begins at the current value of the mpc (or pc) and continues until a breakpoint is encountered or the assembly instruction STOP is encountered.

3.3.9 stats

The **stats** command is used to collect performance statistics. The user types

```
stats on
```

the **stats on** command directs the simulator collects mpc cycles, pc cycles, assembly words fetched, and assembly words stored. The simulator prints and resets this data after the next step or go. The command is only applicable to the assembly level.

```
stats off
```

The **stats off** command shuts off collection and printing of performance statistics.

3.3.10 escape to shell

Sim treats any line preceeded by an exclamation point (!) as a shell command.

3.3.11 help

The **help** command lists a summary of the simulator commands.

3.3.12 quit

The command **quit** exits to the unix prompt.

4 The microcode assembler: ma

The microcode assembler (called ma) accepts a file containing microcode assembly language (mal) and produces an object file that can be loaded into the control store of the microarchitecture simulator, sim. The microcode assembly language is described in detail in Tanenbaum's book, *Structured Computer Organization*, and also in the the last section of this document.

In addition to checking for syntax errors, ma also checks the microcode source for statements that are impossible to implement on the microarchitecture. For example, the statement:

```
rd;wr;
```

is syntactically correct, but will cause a bus conflict. ma also checks for attempts to set the constant registers, simultaneous commands for either the alu or shifter, simultaneous setting of more than one scratchpad register and the simultaneous setting of the mbr and issuing a rd signal.

4.1 Running the microcode assembler

Normally, the microcode assembler is invoked from IST, however it is possible to run it as a stand-alone program. To run the microcode assembler, type

```
ma <filename>
```

where <filename>.mal holds the microcode source. The microcode assembler creates an object file that has the same name as the source except that the extension is changed to "mic".

4.2 Line renumbering

Ma automatically renumbers your source program. Tanenbaum's mal requires that a line number (label) preceed the body of every microinstruction:

```
0: mar := pc; rd;
1: pc := pc + 1; rd;
2: ir := mbr; if n then goto 28;
.
.
.
28: tir := lshift(ir+ir); if n then goto 50;
```

Further, the line numbers merely REFLECT microinstruction location; they do not DEFINE microinstruction location. This means that if you would like to insert some new instructions between lines 2 and 3, then all the old lines must be renumbered and all the gotos adjusted. This is tedious at best when done by hand. Ma now allows the programmer to use any alphanumeric string (except the string "goto") to label a microinstruction. For example, a programmer could modify the microcode given above in the following manner:

```
0: mar := pc; rd;
1: pc := pc + 1;
2: ir := mbr; if n then goto 28;
2a: ir := mbr; mar := pc; rd;      {double word instr}
2b: pc := pc + 1; rd;              {fetching next word}
2c: a := mbr; if n then goto new;
.
.
.
```

```
28: tir := lshift(ir+ir); if n then goto 50;
.
.
.
new: a := lshift(a); if n then goto new3;
```

Ma renumbers the lines so that they appear as follows:

```
0: mar := pc; rd;
1: pc := pc + 1;
2: ir := mbr; if n then goto 31;
3: ir := mbr; mar := pc; rd;      {double word instr}
4: pc := pc + 1; rd;              {fetching next word}
5: a := mbr; if n then goto 113;
.
.
.
31: tir := lshift(ir+ir); if n then goto 53;
.
.
.
113: a := lshift(a); if n then goto 117;
```

5 The assembler: assem

Assem accepts symbolic assembly language programs and produces object code. Much of the language is defined by the architecture-definition file, but there are some universal rules about the structure of the language.

1. Any alphanumeric string that begins in column 1 is considered a label. A label is given a value by either a) beginning a line of executable code with it or b) using any of the pseduo-ops EQU, ORG or CONST
2. Separate opcodes from other fields with spaces.
3. Other fields can be separated by spaces or commas. If the field itself is a punctuation, there need be no separation.
4. Arithmetic expressions are legal anywhere the use of a label is legal (excluding label definition.) The operators in such an expression are restricted to + and -. Assem is a two-pass assembler, so there is no restriction regarding forward references.
5. STOP is a built-in opcode that is available no matter the assembly language being used.

6. END is a built-in pseudo-op that must appear as the last line of code in an assembly language program.

7. Execution of the code begins as location zero.

8. ORG changes the memory location counter. It takes an arithmetic expression as an operand. Caution: it is possible to overwrite previous code by careless use of ORG

9. CONST sets a (16-bit) value at the current memory location. It takes an arithmetic expression as an operand. Consecutive CONST pseudo-ops will set values in consecutive memory locations.

5.1 Object code

Assem creates an object file that includes some symbolic opcode information as well as the object code. (The object file has the extension ".mac") The opcode information is used by the microarchitecture simulator "sim".

5.2 Running the assembler

Normally, the assembler is invoked by IST. However, the assembler can be run as a stand-alone program.

In order to correctly process the assembly code, assem requires

1. the source program (ends in "asm")

2. the architecture definition file. (This file contains all the registers and instruction formats of your "level-2" machine.)

To run the assembler, type

```
assem <architecture-definition file> <filename>
```

where <architecture-definition file> holds the definition of the instruction-set architecture, and where <filename> holds the symbolic assembly language program. Assem searches for a source file named <filename>.asm The assembler creates an object file named <filename>.mac

6 The architecture definition file

This is one of the two files required by the assembler. The assembler invokes the meta-assembler ("meta") to process the architecture definition file. The meta-assembler creates two files. The first, called `instr.set`, contains the information that the assembler needs in order to translate programs written in the newly-defined assembly language. The second, called `instr.log`, records the formats of the new language in a form suitable for humans, and also records any errors in the architecture definition file.

The architecture definition contains all the register declarations as well as the instruction formats for an assembly language. The file consists of a register definition section followed by an instruction definition section.

6.1 Register definitions

The first line of the register definition section is

```
#registers
```

The # symbol is located in column 1. Following this are lines of the form

```
<symbol> <value> <scratchpad register index>
```

where <symbol> is the symbolic name of the register and <value> is the corresponding value used in the object code. If the register is not explicitly mentioned in the object code, the keyword 'null' may be used for <value>. The <scratchpad register index> specifies the microarchitecture's scratchpad register that will hold the (assembly-level) register. The following is an example of a register definition section for Tanenbaum's Mac-1 assembly language:

```
#registers
pc  null  0
ac  null  1
sp  null  2
```

6.2 Instruction definitions

Each instruction definition begins with the command

```
#instruction
```

The # symbol must be in column 1.

An instruction definition consists of field declarations. Each field declaration consists of naming the field, specifying the field's position within the instruction word, and listing legal values for that field. The field declarations must be listed in the order that they will appear in a line of assembly language.

To specify a field's name and position, use a line with the form

```
#<field name>  <beginning bit number>:<ending bit number>
```

Bits are numbered starting with 0 as the most significant bit. The user is not limited in the number of bits in an instruction. The instruction will be stored in contiguous 16-bit words. Instructions are aligned on word boundaries.

Legal values are specified immediately following the field name specification. Legal values are given as symbol-value pairs, one per line. The symbol specifies a legal assembly language symbol, and the value is its corresponding value in the object code.

There are two special types of legal values. They are labels and registers. If the user specifies labels as legal, then the assembler will accept any symbolic label as a legal value in the field. If the user specifies registers as legal, then the assembler will accept any register's symbolic name as a legal value in the field. To specify that labels are legal, use the word

```
labels
```

on a separate line. Registers are specified similarly.

The following declares the 8-bit opcodes of Tanenbaum's Mac-1 language:

```
#instruction
#opcode   0:7
 insp      0b11111100 ; binary (0b), octal (0)
 desp      0b11111110 ; or hex (0x) values are legal
#operand  8:15
 labels
```

6.3 Treatment of symbolic names

Symbolic names normally contain any number of letters or digits, and the initial character is a letter. However, it is sometimes convenient to use a punctuation character as a symbol to indicate an addressing mode. For example, if using the C addressing convention, &x would mean "the address of x" and *x would mean "the contents of the variable pointed to by x". For this reason it is possible to declare single punctuation characters as symbols:

```
#mode 4:5
 &      0b00
 null   0b01
 *      0b10
 @      0b11
```

There is a restriction on the use of punctuation. Punctuation and conventional symbols names cannot be mixed in a field declaration. The keyword 'null' indicates that a punctuation symbol need not be present in a legal line of assembly language.

Meta reserves the characters

```
   ;  #  (  +  )  ,
```

for its own use. The user may not use these characters as symbolic names.

6.4 Sample language declarations

The following is a complete declaration of Tanenbaum's Mac-1 assembly language:

```
#registers
pc null 0
ac null 1
sp null 2
#instruction
#opcode 0:3
 lodd 0b0000
 stod 0b0001
 addd 0b0010
 subd 0b0011
 jpos 0b0100
 jzer 0b0101
 jump 0b0110
 loco 0b0111
 lodl 0b1000
 stol 0b1001
 addl 0b1010
 subl 0b1011
 jneg 0b1100
 jnzr 0b1101
 call 0b1110
#operand 4:15
labels
#instruction
#opcode 0:7
insp 0b11111100
desp 0b11111110
#operand 8:15
labels
#instruction
#opcode 0:15
pushi 0b1111000000000000
popi  0b1111000100000000
push  0b1111010000000000
pop   0b1111011000000000
retn  0b1111100000000000
swap  0b1111101000000000
```

The following is an example assembly language that uses C-type addressing symbols:

```
#registers
pc null 0
ac 0  1
zero 1 5     ;scrpad register zero
sp 2 2
d 3 15       ;scrpad register f
m 4 14       ;scrpad register e
#instruction
#opcode 0:1
load 0b00
store 0b01
#mode 2:3
&     00     ;immediate addressing
null  0b01   ;direct addressing
*     0b10   ;indirect addressing
@     0b11   ;indexed
#operand 4:15
  labels
#instruction
#opcode 0:1
 jzer 0b10
#mode 2:3
 &    00      ;immediate addressing
 null 0b01    ;direct addressing
 *    0b10    ;indirect addressing
 @    0b11    ;indexed
#operand 4:15
 labels
#instruction
#opcode 0:7
  move 0b11000000
  add  0b11010000
  and  0b11100000
#operand1 8:11
  registers
#operand2 12:15
  registers
#instruction
#opcode 0:11
  compl 0b111000010000
  inc   0b111000100000
```

```
  dec   0b111000110000
  pop   0b111001000000
  push  0b111001010000
#operand 12:15
 registers
```

7 Register-transfer language

This register-transfer language is Tanenbaum's microcode assembly language ("mal"). The corresponding microarchitecture is described in detail in Tanenbaum's book, *Structured Computer Organization*, 3rd. ed.

7.1 The scratchpad registers

There are 16 scratchpad registers. Their names are `pc, ac, sp, ir, tir, 1, 0, -1, amask, smask, a, b, c, d, e,` and `f`. The registers `1, 0, -1, amask,` and `smask` hold constants. Their values are given in the following table:

register	value
`1`	`0000 0000 0000 0001`
`0`	`0000 0000 0000 0000`
`-1`	`1111 1111 1111 1111`
`amask`	`0000 1111 1111 1111`
`smask`	`0000 0000 1111 1111`

7.2 Other registers

The other registers in the microarchitecture are `mar`: the memory address register, `mbr`: the memory buffer register, `mpc`: the microarchitecture's pc, `mir`: the microarchitecture's instruction register, the `alatch` and the `blatch`

7.3 Storing into a register

Storing into a register is denoted by pascal-like assignment statements:

```
ac := a;
```

This is an instruction to store the contents of register `a` into register `ac`

7.4 ALU functions

add
```
tir := ac + a;
ac := ac + 1;
```

Remember that the register 1 stores the constant 1.

and `tir := band(ir,smask);`

The function band takes the bit-wise boolean "and" of two registers.) In this case, the contents of the ir and smask registers are being anded. The result is stored in the tir

invert `a := inv(ac);`

The invert function produces the complement of a registers. In the statement above, the complement of register ac is stored in register a.

pass The pass function does nothing.

```
a := ac;
```

The pass function just passes the value of the named register through the ALU without doing anything to it. The statement above illustrates the pass function. We are storing the contents of register ac into register a and we don't want the ALU to process the value in any way.

7.5 Shifter functions

The shifter can shift left or right one bit:

```
tir := lshift(tir);
ac := rshift(a);
```

Tricky stunt! shifting left two bits in one clock pulse:

```
tir := lshift(tir + tir);
```

7.6 Branching

7.6.1 Unconditional jump

```
goto <line-number>;
```

7.6.2 Conditional jump

The ALU has two "status" outputs called 'n' and 'z'. The output n is true when the leading data bit of the ALU is 1. The output z is true when all of the data bits from the ALU are zero. We can jump based on the value of n or z. For example, the following code fragment tests to see if the contents of register ac are zero. Remember that the value of n and z depend on the data outputs of the alu. For that reason, we must pass the contents of ac through the alu in order to test if its contents are equal to zero:

```
ac := ac;  if z then goto <line-number>;
```

Suppose we want to determine if the contents of the ac register are even. If so, we want to jump. We can AND together the contents of the ac register and the 1 register, and if the result is zero, then we can jump. But, suppose that we don't actually want to STORE the result of the AND function. If we don't want to store the output from the ALU (after it's gone through the shifter, of course), we use the pseudoregister alu:

```
alu := band(ac,1);  if z then goto <line-number>;
```

The following is an example program that shifts the contents of the ac to the right by the number specified in the a register. (If a holds a value of 5, then the ac will be shifted right 5 bits.)

```
0: alu := a; if z then goto 3 ;
1: ac := rshift(ac);
2: a := a + -1; goto 0;
   {the register -1 holds the value -1}
3: goto 3;
   {this language doesn't have a halt
      and doesn't need it (as we will
                           find out.)}
```