UNIVERSITY OF MARYLAND

GLENN L. MARTIN INSTITUTE OF TECHNOLOGY ⋄ A. JAMES CLARK SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Introduction to Software Tools.

Several software tools are available to go with the hypothetical computer (Mic-1/Mac-1) of Chapter 4 in A. S. Tanenbaum's, *Structured Computer Organization*, 3rd Edition textbook, which has been summarized and augmented in the *Microarch.ps* or *Microarch.pdf* notes by C. Silio. These software tools include a microprogram assembler, a "macroprogram" assembler for the conventional machine level instructions of Mac-1, and a simulator that will allow running the machine using micro- and macro-programs that have been generated using these other tools. These programs, along with various other files such as sample programs, etc., reside in a directory that may be changed at any time by the computer staff. Thus the mechanism for access to these files is handled in a way that may not be familiar to you.
Before you can use any of files in a particular computer run, you must type the command

```
tap  ee350
```

This command will make sure that your run has access to the programs no matter where they may have been moved. [*]

Once this `tap ee350` command has been executed, for the rest of that computer run (until you logout) the simulator, assembler, loader, and microprogram assembler can be run using the commands given in the handouts for these programs. Other files, such as sample programs, can be accessed using

```
$EE350
```

as though it was the directory name. As an example, the command

```
ls  $EE350
```

will tell you what is in the directory.

Getting Started; Windows.

Once you have gotten your Glue (Open Lab) account and password, and have figured out how to login and logout, you need to know about some of the features of UNIX^{TM} [**] in order to effectively use the software tools for ENEE 350. This is not the place to give a full introduction to UNIX but a word of warning is in order for those who are accustomed to DOS. In UNIX file name paths use /, as opposed to the \ in DOS.
The simulator program, called **sim**, runs on Sun 3's, Sun 4's, or DECstations under X-windows. You do not need to know which type of machine you are actually logged into; the software is set

[*] In order for the `tap` command to work, the file named .cshrc in your home directory should include the following (as is the case automatically in the default setup for new accounts):
```
if (-e /local/software/bin/.twigrc) then source /local/software/bin/.twigrc
```
[**] UNIX is a trade mark of UNIX Systems, Inc.

up to automatically query your machine and pick the correct version. Most of the workstations you will be able to use are set up as X-terminals, and you will automatically be in the X-windows environment once you are logged in. If by any chance the screen does not show a window (an area with a border taking up less than the entire screen), then the command

```
startx
```

should initialize the X-windows environment. With the cursor within the window, type your

```
tap  ee350
```

to get set up so that you have access to the ENEE 350 software.
The mouse can be used to move, resize, and otherwise manipulate this window, and can also be used to create additional windows, destroy windows, etc. (You will have to type the `tap` command in any additional window from which you wish to access the ENEE 350 software, since each window is in effect treated as a separate login.) Pressing the different mouse buttons with the cursor in the various border regions of a window causes different useful window-manipulation actions.

Getting Started; Editing Files.

You need to learn at least the rudiments of one of the text editors which run under Unix. The most popular choices are *vi* and *emacs*. Information on both (and the entire *vi* manual) is available through the "help" function in the dash bar at the top of the normal X-windows screen. The choice of which editor to use is a personal one, and these notes will make no attempt to guide you through editor usage.

Overview of Software Tools.

The structure of the machine we have been using as our example is a set of hardware provided with a microprogram that interprets conventional machine level instructions. In order to do anything useful this hardware would have to be provided with a (binary) conventional machine level program in its main memory. In addition it would need a (binary) microprogram, to interpret the conventional machine level instructions, in its control memory. The set of software tools for the Tanenbaum example machine is centered around a simulator for this hardware. Just as with a real hardware version this simulator must be provided with a "macroprogram" (the conventional machine level program) and with a microprogram in order to do anything useful. Because the simulator also provides an informative graphic display of what is going on in the hardware, you need to also provide the filename of a file containing the symbolic microcode. The simulator program expects a *hexadecimal* version of the macroprogram, rather than a binary one, so the macroassembler provided as part of the set of tools automatically generates such a hexadecimal file as its output.
The executable programs in the tool set include

micgen	microprogram assembler
assem	macroprogram assembler
load	linking loader
sim	simulator and display program

In addition, there are some "disassemblers" for both the micro and macro levels which take the binary or hexadecimal code and try to translate it back into something that is at least semi-readable by a human.

dismic	microprogram disassembler
dismac	macroprogram disassembler

To allow working with essentially the machine of the Mic-1 example the microcode of Figure 4-16 in the text, with one change, is provided in the file

```
$EE350/halt.pascal
```

The one change is that a "`halt`" microinstruction has been included so that it is possible to stop the simulated machine. Similarly, the macroassembler `assem` recognizes the mnemonics of the Mac-1 instruction set (Figure 4-14) plus the additional instruction `HALT` with opcode FFFF_{16}.

The Microprogram Assembler.

The tool to provide microprograms from an input format that is at least somewhat palatable to a human is `micgen`. You will need to use this program only when you write microcode to implement new conventional machine level instructions. The convention used in this `micgen` command, and in the remainder of this set of notes, is that the the things you are to type literally appear in `''typewriter'' font` and *italic* is used to indicate filenames you need to supply, such as *inputfile* and *outputfile*.
The microcode listing in Figure 4-16 of Tanenbaum should be sufficient to indicate the basic capabilities of the microassembler.

A word of warning; the command

`micgen` *outfile*

will give an error message and not execute if *outfile* already exists. This keeps you from overwriting a previous file, but gets annoying at times. Use the command

`rm` *outputfile*

to delete the old *outputfile* before rerunning the microassembler.

The Macroprogram Assembler.

The macroprogram assembler is called with the command

`assem` *programfile*

and is quite adequately described in the handout. No options are needed or permitted. The *programfile* consists of your program written using the mnemonics of Mac-1 (Figure 4-14) and allowing the use of symbolic addresses. See the "Assembler and Linking Loader; User's Manual" for input formats, etc.

The Linking Loader.

Subroutines may be assembled separately from their corresponding main program, using separate `assem` commands. The `load` command links subroutines and main programs and generates an absolute load module for use by the simulator. See the "Assembler and Linking Loader; User's Manual" for details on the formats for the command.

The Simulator.

There is a separate handout, "Instructions for Use of the ENEE 350 Simulator", which outlines the various features of the simulator and how to use them.

The Disassemblers

These programs attempt to translate binary micropgrograms and machine language programs, respectively, into something approaching a human-readable source from which the binary might have been generated using `micgen` and `assem`.

The microprogramm disassembler call is

`dismic` *filename*

and the program expects that *filename* contains a legal binary microprogram. If used on a file which is not a binary micropgrogramm it does its best to translate what it finds, giving somewhat strange looking results.

To translate a binary machine language program *progfile* back into assembly language use the command

`dismac` *progfile*

Two comments are needed with regard to the `dismac` program. First, it translates every line as though the line contains a machine language instruction. This means that even non-executing parts of the program, such as constants, show up as though they were assembly language instructions. Second, this program is set up to allow you to enter a binary machine language instruction from your terminal and see the corresponding assembly language version. Since it normally waits for more input, even after finishing with an input file, yoy need to type "q" (quit) to exit the `dismac` program.

UNIVERSITY OF MARYLAND

GLENN L. MARTIN INSTITUTE OF TECHNOLOGY ⋄ A. JAMES CLARK SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Use of the Microcode Translator (micgen)

This program, which is part of the set of ENEE 350 tools which also includes an assembler, a linking loader, and an X-windows based simulator, is provided courtesy of Prof. Paul Amer of the University of Delaware (He is also responsible for the disassemblers; the other programs in the set were generated at UMCP).

The purpose of the program is to translate microinstructions which are given in a pascal-like format (see Chapter 4 of Tanenbaum's text *Structured Computer Organization, 3rd ed.)* into binary strings suitable for use in the simulator program.

To translate the symbolic microcode in your file "foo" and put the resulting binary microcode in a file named "bar", use the command line

```
micgen foo bar
```

One of the features of the program set is the ability to write new microcode for machine-language instructions which are not in the original instruction set of the hypothetical computer in Tanenbaum. The translation of such altered symbolic microcode files is the principal use for micgen.

There are only a couple of cautions concerning its use:

(1) The last line of any symbolic microcode file used as input to micgen **must** be terminated with a carriage return. Otherwise the micgen program will hang.

(2) The error messages generated by micgen can be very misleading; do not give much credence to their details, except for the line number of the first error reported. If you get an error message there **is** an error in your symbolic microcode, and there is almost certainly an error on the first line reported, but the message is often of little use in figuring out what the error is. In general, the first line number reported as in error is correct, in that there is **some** error associated with that line. Figure out what it is, correct it and then try again before paying any attention to other error messages. *Note item (1) above, which is one of the most common causes of error messages.*

(3) The `micgen` program is quite stupid in one respect. When presented with a line of microcode such as

```
34:ac:=pc+mbr;
```

it chokes, generating an error message. This is because it assigns the first operand it sees to the A-bus and the second operand to the B-bus, and the MBR is connected only to the A-bus. The error message is something about a bus conflict. If this is changed to

```
34:ac:=mbr+pc
```

it works fine, and is translated correctly.

UNIVERSITY OF MARYLAND

GLENN L. MARTIN INSTITUTE OF TECHNOLOGY ⋄ A. JAMES CLARK SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Assembler and Linking Loader; User's Manual

This handout describes an assembler and linking loader for the hypothetical computer of Chapter 4 in Tanenbaum's 3rd Edition textbook, Structured Computer Organization, *which is also described in the Example Microarchitecture Notes ("Microarch.ps" or "Microarch.pdf") put together by C. Silio. The standard assembly language instruction set is as given in Figure 4-14, page 185, of Tanenbaum, with the corresponding microcode being that of Figure 4-16. In the Microarch Notes the instruction set is specified in Figure 11 and the corresponding microcode is specified in Figure 13. One addition has been made to both the machine language instruction set and the microinstruction set over what is given in Tanenbaum's 3rd edition textbook; namely, a "HALT" instruction with opcode* `ffff` *(hex) has been added. These programs are part of a suite of programs supporting this hypothetical machine, the other routines being a "microcode assembler", a simulator with graphics capabilities, and a pair of "disassemblers" which translate binary machine language and binary microcode, respectively, into something at least semi-readable by a human.* [†] *There are provisions for adding new machine language instructions by rewriting the microcode, and corresponding provisions for telling the assembler about these new instructions. The simulator takes a file containing the machine language program and puts the program into the simulated main memory, and also takes a file containing the microcode and puts that into the simulated control store. This allows much experimentation with both microcode and conventional machine level instructions. The present versions of the local software are still somewhat experimental, so no guarantees can be made about being bug-free. In fact, any program this long is almost sure to have bugs; as of this writing the C language source file for the assembler has 2027 lines and that for the linking loader 680 lines. Things should actually work fairly well, as students have now hunted for bugs for more than ten semesters; there are no* **known** *bugs.*

Please report all real or suspected bugs to me via email. My email address is:

pugsley@eng.umd.edu

I. Introduction.

An assembler is a program which translates other strings of text into machine language. The present assembler allows the separate translation of main programs and subroutines; i.e., the various

[†] *The original support software was due to Prof. Paul Amer at the University of Delaware. The micro-assembler and the disassemblers are still his programs, but the others have been written locally by Prof. James Pugsley without using the U. Del. code. All of the programs are written in C (hopefully in ANSI C; at least they were all compiled successfully using the "-ansi" option on the C-compiler). It should in theory be possible to compile any of them except the simulator on any machine with an ANSI C compiler. The simulator makes extensive use of the Xlib commands from the X-Windows system, and will therefore run only on a machine which also supports X-Windows (such as Xfree86 under Linux). For some details concerning the X-libraries needed see the README file in the directory $EE350, which is available after executing a "tap ee350" command.*

routines do not all need to be in the same file. Because these routines eventually need to be made part of a single absolute load module that gets put into the computer's main memory, there is another program (called a *linking loader*) which takes care of this part of the job. The assembler translates each file assuming that memory location zero is the first location used by the routines in that file. The output of the assembler consists of a *relocatable object program* which is the appropriate code along with flags that indicate which addresses need to be fixed if the program is not actually loaded starting at main memory location zero. The linking loader then takes the relocatable object versions of all of the routines and puts them together in a single *absolute* object program which is often referred to as the absolute load module.
The assembler is invoked by the command

```
assem  <filename>
```

where "<filename>" represents the name of a file containing source (assembly language) programs. If the sample program shown below in Figure V.1 is in a file named "prog" in your current directory, then the command

```
assem  prog
```

invokes the assembler to translate this program. The assembler generates an immediate set of messages on your screen, and also creates three new files named "prog.rel" (the relocatable object code), "prog.list" (a more complete listing of what the assembler did to your source program), and "prog.esd" (the *external symbol dictionary* needed by the linking loader).
If you have several different files containing a main program and subroutines which all need to be used together, you must call the assembler once for each file. If there are three files are denoted by "<name1>", "<name2>" and "<name3>" there will be a total of nine files generated by the calls to the assembler. To link and load these files use the command

```
load <name1> <name2> <name3>
```

The format here is the command "load" followed by the names of files, with one or more blanks (or tabs) separating the file names. The linking loader looks for the " .rel" and " .esd" versions of each file name, and should generate an error message if it does not find some of what it needs.
The output of the loader is the file <name1>.abs which contains the absolute load module. In addition, the loader will generate a listing on your screen of the final combined external symbol dictionary, which gives information on all of the global symbols and their values.

II. Format of Assembly Language Instructions.

The format of each assembly language statement is

[**label**]   <**operation**>   [**operand**]   [/**comment**]

where square brackets indicate features which are optional or do not occur in every instruction. Every instruction must have a nonblank operation field, all of the other fields are optional in at least some instructions (but may be required for other instructions). Note that the names of the fields are only generally descriptive of their purposes. The operation field, for example, can contain the mnemonic for a machine operation, an assembler directive, a symbol, or a constant. In some of these cases it is stretching things to call what is in the field an operation.
A label, if present, must start in column 1 and must have an alphabetic character as its first character. All characters in the symbol must be alphabetic, numeric, or '_' (underline).
The operation-field, as indicated above, may contain a variety of things depending on the particular usage.

The operand-field may contain either a constant or a symbol. In addition it is possible to have a *literal* in the operation field. Literals are discussed below in section III.
Fields are separated by one or more blanks or tabs. There is no provision for continuing a statement on a second line because there is zero likelihood of ever wanting to do so. One of the most common errors is to get your "comment" starting not in the leftmost column but one column to the right. This means it gets treated by the assembler as being in the operation field rather than in the comment field. While the assembler doesn't care whether you line things up neatly, as is done in the examples on page 6. both the TA's and I do when we come to grade your work; you should use tabs to keep things aligned.
The comment-field extends from an initial slash , '/', to the end of the line. The judicious use of comments makes programs much more understandable, but too many comments is almost as bad as none at all.

III. Constants, Symbols and Literals.

The descriptions above of the various fields refer to constants and symbols, and these must have certain forms to be recognized by the assembler.

Constants Constants must be integers and may be specified in either decimal or hexadecimal. Decimal constants are written in the usual form. Hexadecimal constants must have the characters "0x" as a prefix. Thus the constant fifteen in hex would be typed as "0xf" for the assembler. Either type of constant may be preceded by a "+" or a "-". Leading 0's will be ignored; but note that leading 0's in the hex case **follow** the prefix "0x". Thus the constant fifteen may be specified in hex as "15", or "015", or "0xf" or "0x0f" or "0x00f", etc. Trying to use something like "00xf" is incorrect, and will generate an error message.

Symbols Symbols are one of the main features of assembly language that distinguish this level from the machine language level. A symbol is a string of characters each of which must be one of alphabetic, numeric, or '_' (underline). The first character in the string must be alphabetic, and the limit to the number of characters in a symbol is currently 10. As examples, "asub_xxx", "x2" and "abc2_43zp" are legal symbols, but neither "2xy" nor "a4.sub" is.

Literals This assembler, in common with many, allows the programmer to specify constant data without having to consciously reserve a storage location and put the contstant into that location. This is done by enclosing the constant in parentheses and treating the resulting character string as though it were a symbol. The process is probably most readily described through a simple example. The assembly language statement

```
lodd   3
```

causes the contents of memory location 003 to be copied into the accumulator. By contrast, the statement

```
lodd    (3)
```

causes the assembler to reserve one memory location whose symbolic address is (3) **and** into that location to put the code for the integer 3. Any legal constant can be enclosed in parenthesis to form a literal. Try some examples of literals and look at the resulting listing file; in particular see what shows up in the symbol table.

The assembler makes two passes over the source language program. In the first pass the various symbols are identified and their values determined. Then in the second pass the assembler generates the relocatable code, inserting the values of the symbols. One common use for symbols is as symbolic addresses. You should be careful to distinguish between the *value* of a symbol used as an address and the *contents* of the addressed location.

IV. Assembler Directives.

Assembler directives (sometimes called "pseudo-instructions") provide information to the assembler relating to how it is to translate the given source program. Assembler directives appear in the operation field of an assembly language statement but do not result in the generation of any relocatable code. The paragraphs in this section describe each of the available assembler directives, in alphabetic order; examples of their use are given in section VI.

`END` Used (with a symbol or constant in the operand field) to indicate to the linking loader, and eventually to the simulator, that the initial value of the program counter should be the value of the operand. If there is no END statement the starting address is taken to be 000. Only a main program should contain an END directive, and should contain no more than one such.

`ENTRY` Used with "<symbol> in the operand field to indicate that <symbol> is the entry-point to a subroutine. Specifies that <symbol> is global (i.e., usable by routines outside the file containing the current routine). You can think of this as telling the world that the value of <symbol> is defined here, should the world need to know.

`EQU` (for "equate") Used to define the value of a symbol. The format is

"<label>   EQU   <value>"

where value may be either a constant or another symbol whose value is already known. This latter restriction prevents "forward references"; thus the sequence

```
x  EQU  2
y  EQU  x
```

is legal, defining both `x` and `y` to have the value 2, while the sequence

```
y  EQU  x
x  EQU  2
```

is not legal, since it attempts to equate `y` to something which has not yet been given a value. A smarter assembler might allow such forward references, as long as they did not become circular and thus ultimately ambiguous.

`EXTRN` Used with <symbol> in the operand-field to indicate that <symbol> is referenced in routines in the current file, but is defined (given a value) in some other file. In that other file <symbol> should appear as the operand in an ENTRY or GLOBAL directive. This keeps the assembler from objecting that the value of <symbol> is not known, and leaves it to the linking loader to correctly set the value at link time.

`GLOBAL` (Actually identical in its action to ENTRY.) Used to define a symbol as global and available for use by other files. The symbol need not denote an entry point to a subroutine.

`RES` (for "reserve") Used with a constant in the operand field to reserve memory locations. E.g., the line

```
answer   RES   3
```

would reserve three main memory locations, the first of which has symbolic address "answer". Note particularly that this directive simply reserves the locations, it does **not** in any way specify what the contents of these locations will be initially. In fact the contents will actually be random. (You might check this by assembling a program containing an RES directive and then loading it more than once. A look at the " .abs" files generated by the two different calls to the loader will show that the contents of the reserved locations are not the same in the two absolute load modules.)

V. Defining new opcodes for the assembler.

It is possible to define up to 7 new instruction-mnemonic/opcode combinations for any given program. If the file in which the assembly language program resides is named `xyz`, so that it will be translated using the command `assem  xyz`, then new instructions for this program may be defined by creating a file named `xyz.opcodes_new` which contains one line for each new instruction, with the format for each line being
`<instruction mnemonic>  ''tab''  <16-bit opcode for instruction>  ''tab''  <number of bits of operand for instruction>`
Here "tab" means just hit the Tab key. The assembler always looks for such a file when translating a program. The absence of such a file generates the message seen in the examples above; "No new opcodes specified for this program". There are a couple of restrictions on this feature of the assembler: 1) you cannot redefine any existing opcodes and must choose an unused one for the "opcodes_new" feature. This means, among other things, that no new memory-reference instructions can be implemented, since all possible opcodes with 12-bit operand specification have already been defined; 2) the only allowed values for the number of operand bits are 8, 4 and 0.

VI. Examples.

As a simple example of what the assembler and linking loader do, assume that the following main program, shown in Figure V.1, is contained in the file "mainprog" in the current working directory.

```
x    EQU    2
     EXTRN  addv
a    12     /integer twelve
b    0xf    /integer fifteen
ans  RES    1                 /reserved for answer
go   lodd   a
     push
     lodd   b
     push
     call   addv
     insp   x
     stod   ans
     halt
     END    go                /starting address
```

Figure V.1. Example main program.

The assembler generates messages as it works indicating what it has found and how it is translating this. Error messages are generated for, hopefully, all of the illegal and even some of the legal but non-useful things you might try to do in your source program. Any suggestions as to more meaningful or additional error messages will be appreciated. The command

```
assem   mainprog
```

will provide on your screen the information shown in Figure V.2.

```
No new opcodes specified for this program.
0    0    x     EQU     2
1    0          EXTRN   addv
2    0    a     12              /integer twelve
3    1    b     0xf             /integer fifteen
4    2    ans   RES     1       /reserved for answer
5    3    go    lodd    a
6    4          push
7    5          lodd    b
8    6          push
9    7          call    addv
10   8          insp    x
11   9          stod    ans
12   10         halt
13   11         END     go      /starting address
```

Figure V.2. Appearance of screen when main program is assembled.

At this point it is probably instructive to look at some of the other files generated by the assembler. The file “mainprog.list” can be viewed by use of the command

```
more    mainprog.list
```

and is as shown in Figure V.3.

The bottom portion of this list file consists of the *symbol dictionary* for the program. This gives the symbol, its value, whether this value is absolute or must be altered when the loader decides where the program really goes in memory (called relocation), whether the scope of the symbol is local (this file only) or global (defined and/or referenced in another file), and for global symbols whether the symbol is defined (D) in the current file, referenced in the current file but defined elsewhere (R) or is the program starting address (S).

The other files you might want to examine at this point are those which are passed along to the linking loader (mainprog.list is for human benefit only). These consist of “mainprog.rel” and “mainprog.esd” which are, respectively, the relocatable object file and the external symbol dictionary derived from “mainprog”. All of the information in these files is also contained in the “ .list” file, so they will not be reproduced here.

One of the features of this example main program is that it calls a subroutine whose source code is in some other file. This is signified by the `EXTRN  addv` statement on line 1. The code for the subroutine is very simple (this is just the example we had on the board in class) and is given in Figure V.4, where there are no comments because the program is self-explanatory. The line `ENTRY   addv` defines the symbol “addv” as global so that its value can be inserted where needed by the linking loader.

If the subroutine is in a file named “sub” then it is assembled using the command

```
assem   sub
```

You should try this and look at the information that appears as a result of this command. The assembler generates a file “sub.list” which is shown in Figure V.5.

The next step is to link the main program and the subroutine, which is done with the command

```
load  mainprog  sub
```

```
Line    Loctr   Code    Rel     Label   Op      Operand Comment
0       0                       x       EQU     2
1       0                               EXTRN   addv
2       0       000c    abs     a       12              /integer twelve
3       1       000f    abs     b       0xf             /integer fifteen
4       2               rel     ans     RES     1       /reserved for answer
5       3       0000    rel     go      lodd    a
6       4       f400    abs             push
7       5       0001    rel             lodd    b
8       6       f400    abs             push
9       7       e000    ?               call    addv
10      8       fc02    abs             insp    x
11      9       1002    rel             stod    ans
12      a       ffff    abs             halt
13      b                               END     go      /starting address

Symbol  Value   REL     Scope   EXT

x       0002    A       L
addv    0000    ?       G       R
a       0000    R       L
b       0001    R       L
ans     0002    R       L
go      0003    R       G       S
```

Figure V.3. Contents of file "mainprog.list".

```
        ENTRY   addv
addv    lodl    1
        addl    2
        retn
```

Figure V.4. Example subroutine.

This creates a new file named "mainprog.abs" which contains the absolute load module. This file is shown in Figure V.6.

The load command will attempt to link and load whatever relocatable object files you specify on the command line. The loader looks for the " .rel" version of each file and proceeds to link them. If you have more than one main program there will be confusion as to the correct starting address, but the loader should give a message telling you what it has assumed. If the same global symbol is used in more than one file, the loader should also tell you about that error.

It is instructive to execute the command

```
load   sub   mainprog
```

and compare the resulting absolute load module with the one given above in Figure V.6. † The

† This second load module will be named "sub.abs". Since the loader has no real way to tell what

```
Line    Loctr   Code   Rel     Label   Op      Operand  Comment

0       0                              ENTRY   addv

1       0       8001   abs     addv    lodl    1

2       1       a002   abs             addl    2

3       2       f800   abs             retn

Symbol  Value   REL    Scope   EXT

addv    0000    R      G       D
```

Figure V.5. Subroutine "sub.list" file.

```
START = 0003
000c
000f
1c2f
0000
f400
0001
f400
e00b
fc02
1002
ffff
8001
a002
f800
```

Figure V.6. Load module; file "mainprog.abs"

load modules will not be identical. Memory gets allocated to the first-named file first, so the load module in Figure V.6 has the main program in memory starting at location 000, and the subroutine following the main program. The load module "sub.abs", by contrast, has the subroutine first starting at location 000, followed by the main program. Note the difference in the starting address reported to the simulator (the first line of the load module file).

---

is a main program and what is a subroutine it uses the first file name it gets as the name for the load module.

UNIVERSITY OF MARYLAND

GLENN L. MARTIN INSTITUTE OF TECHNOLOGY ⋄ A. JAMES CLARK SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Instructions for Use of the ENEE 350 Simulator

These instructions are for the X-windows based simulator for the hypothetical computer from Chapter 4 of Tanenbaum's text, *Structured Computer Organization, 3rd. ed*, as summarized and extended in C. Silio's *Microarch* notes. Students have tested the software for many semesters so it should be relatively error free.

As with all of the ENEE 350 software, before actually calling the simulator you will need to execute the command

```
  tap ee350
```

This gives your run access to the ENEE 350 programs. If you are using more than one window, each window from which you want to access the course programs must have had such a `tap` command executed in it.

Starting the simulator.

The simulator must be called from within a window in the X-windows system. Most machines in our open laboratories automatically start X-windows when you login.

The command to start the simulator is of the form

```
  sim  <main.abs>  <binary-micro>  <symbolic-micro>
```

where:

(a) `<main.abs>` is the name of the file containing the absolute machine language program (output from the `load` command). †

(b) `<binary-micro>` is the name of the file containing the binary microcode (use "$EE350/halt" if you have not changed the microcode).

(c) `<symbolic-micro>` is the name of the file containing the symbolic microcode (use "$EE350/halt.pascal" if you have not changed the microcode.

Thus, for example, if you have a main program in a file named `main` which calls a subroutine which is in a file named `sub` the sequence of commands to assemble both programs, load the resulting absolute load module, and start the simulator to check the execution of the program would be:

† "`.abs`" because this file is normally created as the output of the `load` program, which automatically appends the "`.abs`" when it creates the file.

```
assem main
assem sub
load main sub
sim main.abs $EE350/halt $EE350/halt.pascal
```

Move the cursor so that the simulator window fits on the screen and click the left-hand mouse button once to see the datapath drawing.

Controls.

The simulator is controlled by a combination of mouse buttons and keys on the keyboard.

Mouse Buttons. Each of the three mouse buttons has a fixed function.

left button The left mouse button steps the simulator. What a "step" consists of is mode dependent; modes are controlled from the keyboard and are discussed below.

middle button The middle mouse button is used to pop up a window showing the contents of the lowest 512 memory locations (addresses `000-1ff`). This is the region of memory in which your program normally resides. Clicking the button again will remove the window.

right button The right mouse button is used to pop up a window showing the contents of the highest 512 memory locations (addresses `e00-fff`). This is the region of memory normally used for the stack. *Note: in its current configuration the simulator initializes the stack pointer to fffc. The highest four memory addresses refer to device registers rather than to actual memory locations.*

Keys. Certain keys or key sequences are used to control the operating mode of the simulator. The pertinent keys are listed in alphabetic order below.

c or **C** (clock) Puts the simulator into "clock-pulse" mode. In this mode each click of the left mouse button advances the simulation by one clock pulse (there are four clock pulses per microinstruction).

g or **G** (go) Puts the simulator into "run-until-halt" mode. One click of the left mouse button will cause the simulator to run without stopping until it has reached and executed a "halt" instruction. Note that if your program contains no halt instruction the simulator will never stop. See the **Warnings and Error Recovery** section below for what to do in this case.

i or **I** (instruction) Puts the simulator into "instruction" mode. In this mode each click of the left mouse button will advance the simulation by one machine instruction.

m or **M** (microinstruction) Puts the simulator into "microinstruction" mode. In this mode each click of the left mouse button advances the simulation by one microinstruction.

pa or **PA** (pause at) actually **pA** and **Pa** would also work) This key combination pops up a window into which you can type the mnemonic of a machine instruction, such as "lodd" or "CALL" (not case-sensitive) if the cursor is moved to the pop-up window. In this pop-up window, either the backspace or delete key may be used to correct your typing. Pressing the return key transmits your input to the simulator and removes the pop-up window. The simulator will now be in "pause-at" mode, and each click of the left mouse button will cause the simulator to advance until either it has completed execution of the next instruction you specified in the pop-up window or until it has completed a halt instruction. The mode display at the lower right of the simulator window will tell you what you specified. Again note that if you specify something that is not a legal mnemonic for the standard machine language instruction set there will never be a match and the simulator will not stop unless it hits a halt instruction.

pe or **PE** (pause every) Pops up a window into which you can enter an integer. Pressing the return key transmits the integer, and puts the simulator into "pause-every" mode. In this mode each click of the left mouse button will advance the simulation by this integer number of machine instructions. Note that pause every 0 instructions is not useful, since the simulation will not advance. Again, simulation will stop if a halt instruction is reached.

q or **Q** (Quit!) Exit the simulator.

r or **R** (restart) Restarts your program. I.e., puts the program counter back to the appropriate starting address as determined from the absolute load module, restores the contents of the portion of memory holding your program to the values they contained when the program was first loaded, and reinitializes the registers. The portion of memory used for the stack area is not initialized, since the contents of this portion of memory should have no bearing on how your program operates (you should assume there is garbage there in any case).

s or **S** (save) Saves the contents of the simulated memory in the file <`main`>`.mem_start` (if key pressed before simulator is started or immediately after a restart) or in the file <`main`>`.mem_end` (if key pressed when the machine has executed a halt instruction). You can only save the initial and final contents of the memory, but you can look at the memory whenever the simulator is stopped by clicking the middle mouse button.

Note that keys have an effect only when the simulator is stopped. If it gets hung up and just keeps running you will have to do something other than hit one of the keys described above.

Warnings and Error Recovery

Any unexpected key should ring the bell on your terminal but have no other effect on the simulator. If you hear the bell when you press a key it means that the key you have hit is not appropriate with the simulator in its current state.

The file containing the C-language source code for the current version of the simulator is more than 3800 lines long, and there **are** certainly bugs in the program. It is recommended that when you initially position the simulator on the screen with the mouse that you leave showing a corner of the window in which you typed the "sim" command. Putting the cursor in that window and typing Control-C (hold the Control key down while typing C) will abort the simulator job. This is the quickest way out if you are hung up in a simulated infinite loop for some reason, or the simulator has decided to run wild.

Control-Z can be used to suspend the simulator, as with most programs under Unix, but in this case this feature is not particularly useful. The X-windows environment queues up events such as keypress and buttonpress events, and when you restart the simulator using the "fg" command **all** of the queued up events get processed. There is currently no way to interrupt the simulator (when it is churning away) that will allow a clean restart from the point of interruption. It seems as though the variety of modes provided should make such an interrupt feature unnecessary.

Please report all real or suspected bugs via email to pugsley@eng.umd.edu

Any suggestions for modifications that would improve the simulator are also welcome.

UNIVERSITY OF MARYLAND

GLENN L. MARTIN INSTITUTE OF TECHNOLOGY ⋄ A. JAMES CLARK SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

TSIM

This simulator for the machine of Chap. 4 in Tanenbaum's 3^{rd} edition text is provided courtesy of Kevin Hildebrand, who was an ENEE 350 student at the time he wrote it. It does not require the use of X-Windows.

Instructions for Mac-1 Simulator (Text Version) Rev. 5/18/92

This program will simulate execution of programs written in Mac-1. It can be used to step through each microinstruction, or it can step by macroinstructions. The simulator actually uses assembled microcode to perform execution, so changes in the microcode can be observed. The simulator will show the currently executing micro and macro instruction in symbolic representation, although if a different microcode other than 'halt' is used, the symbolic representation of the macro-assembly may not be correct. Note, however, that the simulator will still correctly execute these instructions.

To start the simulator, type the following:

```
tsim   <object file>   [<microcode file>   <symbolic microcode file>]
```

For example, if your object file is *Prog1.abs* and you are using the 'halt' microcode, you would type:

```
tsim  Prog1.abs  halt  halt.pascal
```

If you omit the .abs extension on the object filename it will be automatically appended for you. If you leave off the microcode filenames, the simulator will use the files 'halt' and 'halt.pascal'. It will first look for these files in the directory pointed to by the environment variable EE350 (set automatically if you use 'tap ee350') and then it will look in your current working directory.

The simulator screen will appear and you will see the command prompt at the center left of the screen. At the top right are the memory and stack windows. Across the bottom half is the microinstruction execution area and at the top left is the macroinstruction area. Any micro or macro instructions shown are the instructions to be executed next. The values in the registers are from the execution of the previous command. The stack window will shift up or down depending on the location of the stack pointer. If you need to see more of the stack you can use the memory [D]ump command, described below.

Commands: (simply type the letter that is inside the brackets)

[S]: Prompts for the starting address shown in the Memory window. For example, typing address 0005 will display memory locations 0005 through 000C. Addresses entered are in hexadecimal.

[T]: Toggles between showing the execution of each microinstruction and showing the execution of each macroinstruction.

[D]: Prompts for a starting address used to display one screenful of memory. The current screen is saved and the program will switch to 'Dump' mode. There are several commands available in Dump mode, which is described below.

[G]: Executes your assembly language program until it encounters a halt instruction or a breakpoint. You will not see the steps as they execute, only the final result. Note: if your program does not have a halt instruction, or you encounter an infinite loop, pressing control-C will stop the simulator.

[B]: Shows the current breakpoint settings and allows you to change breakpoints. See below for a description of the breakpoint editor.

[Q]: Exits the simulator.

Pressing '?' will display a quick reminder of what each command letter stands for. Pressing any key other than those listed above will step one micro or macro instruction, depending on which mode you have selected. The program when first started will be in microinstruction execution mode.

Memory Dump mode commands:

After pressing [D] at the main simulator screen and entering a starting address, you will see a screenful of memory locations. You now have the following options:

[F]: Move forward to the next screenful.

[B]: Move backward to the previous screenful.

[D]: Enter a new starting address.

[O]: Outputs the currently displayed screen to a file. You will be prompted for a filename in which to save the data. The file can then be printed or viewed using standard Unix commands.

[Q]: Exits the simulator.

Pressing '?' will again provide a quick reminder of what each letter stands for. Pressing any key other than those listed above will return you to the main simulator screen.

Helpful hint: if you need to see more of the stack than the main simulator window shows, you can use Dump mode to view a much larger area. The memory viewer is confined to the 4K address space of the theoretical Mac-1 machine so if you specify an address larger than that, the display will wrap around to zero.

Breakpoint Editor:

The breakpoint editor allows you to define up to seven addresses where you wish the program to temporarily stop execution. By setting a breakpoint to an address in your code and then selecting the 'Go' option from the main menu, your program will execute up to the instruction containing the breakpoint and then return control to you. The program can then be single stepped, or another 'Go' command can be issued.

In addition to setting the address, a stop count must be specified. This allows you to stop your program after the selected address has been encountered a specified number of times. For example, you wish to stop your program after ten times through a loop so you set the address to somewhere in the loop and the stop count to ten. If you want the program to stop the first time the breakpoint is encountered, set the stop count to one.

The breakpoint editor displays the breakpoint number, the breakpoint address (in hexadecimal), the stop count, and the current number of times the address has been encountered.

After a breakpoint is encountered, the address in the macro display is displayed with an asterisk next to it to indicate which address contained the breakpoint.

Commands:

[S]: Prompts for a breakpoint number, the breakpoint address and the number of times the instruction must be encountered to stop execution. The address must be entered in hexadecimal and the three entries should be separated by spaces. Note: the stop count must be greater than zero for the breakpoint to take effect.

[C]: Prompts for a breakpoint number to be cleared. All entries for the selected breakpoint number are set to zero.

Known bugs:

Any location where more than single character input is required will not allow line editing. i.e., if you type something wrong you have to live with it. This is due to a flaw in the support software I am using and I am currently trying to find a way around it.

5/18/92 Kevin Hildebrand