Computer Science 237
Computer Organization
Williams College
Fall 2006

The *Mice* Micro-architecture and the *Male* Assembler

Your final project for this semester is to construct a microcoded interpreter for the language of a hypothetical computer we will call the WC34000. The 34000 machine language is derived from 68000 machine language but simplified in ways intended to make its implementation much less complex. The microcode you write will run on a hypothetical micro-architecture based on the Mic-1 described in Chapter 4 of the 1990 edition of Tanenbaum. You will test and debug your implementation using an interpreter for this micro-architecture, available on the FreeBSD workstations and Macs in TCL 312, TCL 216 and TCL 217a.

The differences between the Mic-1 and the 34000 machine interpreter are all extensions intended to make the project easier. To distinguish the extended versions of the micro-architecture and the micro-assembler from the originals we add E's (for Extended) to their names yielding *Mice* and *Male*. The extensions and modifications are described below.

Expanding the Mic-1's memory component

Like the 68000, the 34000 provides 8 data and 8 address registers. These register sets must somehow be implemented using the registers in the micro-machine's scratchpad. Since the Mic-1 has only 16 scratchpad registers, there would be no room for the micro-program to store other values. Accordingly, the *Mice* machine has 31 registers in its scratchpad. This implies that the A, B and C fields of the micro-instruction format for the *Mice* machine must each be 5 bits long.[1]

Having many registers also affects the assembler. Rather than attempt to pre-assign a symbolic name to each register (as Mal does), the *Male* assembler will accept a notation of the form `R[n]` to identify register `n` where `n` is an integer constant between 0 and 30. In addition, the assembler will allow you to declare and use symbolic names for registers. Register name declarations will take the form:

```
<symbolic-name> = R[n]
```

All such declarations must precede the first line of micro-program code. The declarations are separated from the code by a line containing only the keyword `begin`. Each declaration must appear on a separate line. For example, if the MAL assembler worked this way, Tanenbaum's program would have begun with the declarations:

```
PC = R[0]
AC = R[1]
SP = R[2]
IR = R[3]
     .
```

[1]Actually, the details of the micro-instruction are somewhat irrelevant to the project. You will be writing in the micro-assembly language and will never have to look at the binary micro-code produced. These details are provided, however, to make the *Mice* machine feel more “real”.

```
        .
        .
    begin
        .
        .
        .
    end
```

Unfortunately, symbolic register names must begin with a letter or number, so we cannot give a register the name `-1`, but you could name it `NEG1`.

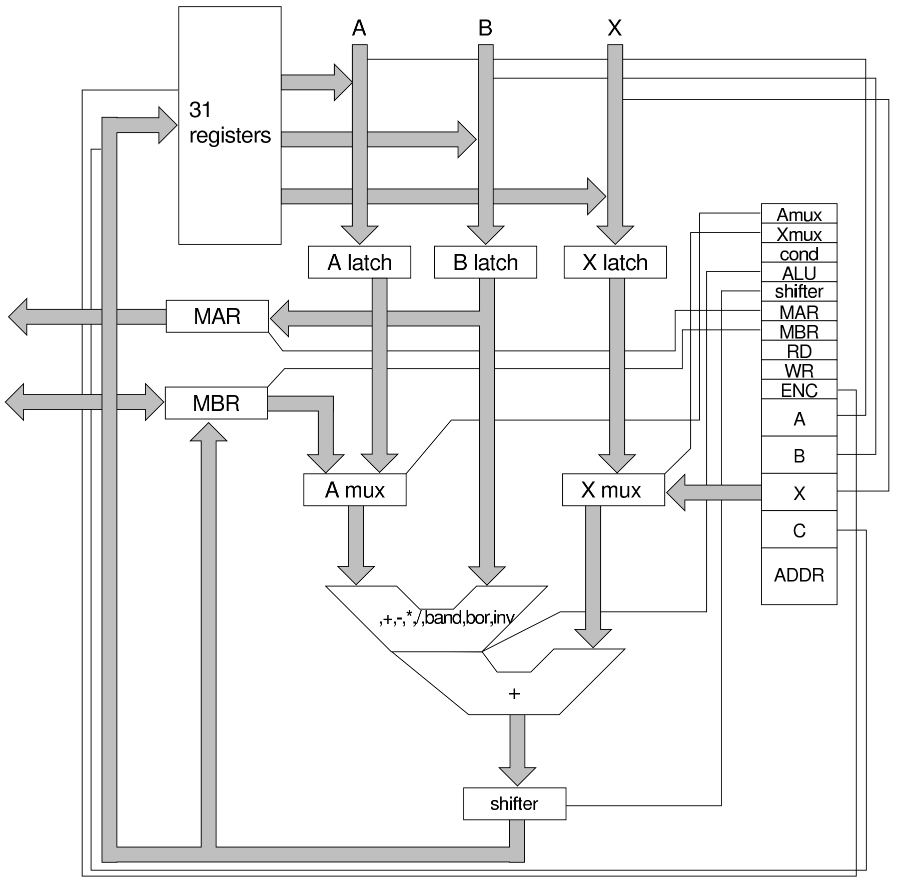

Figure 1: The extended micro-architecture used for the WC34000.

We have an “extra” bus that provides a third operand to a modified ALU. This operand can be any valid register address or a small constant between 0 and 31. Constants are specified with a leading hash, for example, `#15`. If no X value is specified, it defaults to `#0`.

Even with 15 extra registers, the 34000’s registers would be hard to implement on the Mic-1. The problem is that there is no way to access a scratchpad register using an index value. The actual register number must be hardcoded in each Mic-1 instruction. To remedy this in the *Mice* machine, we will allow scratchpad register 0 to be used as an index register when referencing the scratchpad. At the level of binary micro-instructions, this will be done by specifying the value 31 for the `A`, `B`, X, or C field. Since there are only 31 registers and we start at 0, 31 cannot be used as a register number. Instead it will instruct the hardware to use the value in the low-order 5 bits of scratchpad register 0 as the number of the register to use. Since register 0 does have this special purpose, the assembler will recognize the symbolic name RP (Register Pointer) as equivalent to R[0]. In addition, the microcode assembler will allow the notation R[RP] to be used to indicate that indexing should specified in the A, B, X, or C field of a micro-instruction. For example, if register 0 contained 5, and register 3 contained a 1, the instruction

```
R[RP] := R[RP] + R[3]
```

would increment register 5 by one.

Finally, the 30th scratchpad register will also have a special purpose on the *Mice* machine. It will be used to hold the micro-program counter (MPC). This makes the MPC directly available to the micro-program, and will provide a convenient mechanism for doing multi-way branches and even a primitive form of procedure calling.

ALU and Shifter Improvements

The ALU in the *Mice* machine combines three values, A, B, and X. A and B are combined with infix operations '+', '-', '*' and '/' as well as the bitwise logical functions 'band' (bitwise and) and 'bor' (bitwise or) and a 1's complement operation ('inv') is available on the A bus. The value from the X bus is then added in.

In addition to the Mic-1 functionality, on op code binary '11', the shifter will *rotate* its input left 8 bits. This will have the effect of swapping the order of the bytes in a word. The micro assembler will generate this code when the function name 'rotate8' is used.

Input/output

To make input/output unrealistically simple, the *Mice* machine's MIR includes three bits named 'GET', 'OUT' and 'NUM'. When OUT is set, the processor will send the contents of the MBR to the system's standard output device for printing. If the NUM bit is one during a OUT, the MBR will be interpreted as a 16 bit signed integer. Otherwise, the processor will assume the the low order byte of the MBR contains a single ASCII character to be printed. When the GET bit is set, the processor will input a value from the system's standard input device. Again the NUM bit will determine whether numeric or character input is expected. All such I/O operations will complete in one micro cycle.

The micro assembler is extended to recognize the statement types 'GETCH', 'GETNUM', 'OUTCH' and 'OUTNUM' which set the appropriate bits in the micro instruction. These statements can be used like the 'RD' and 'WR' statements.

Finally, the MIR is extended to include a 'DUMP' bit and the Micro assembler is extended to include a 'DUMP' statement which causes the processor to print information to assist in debugging. A dump will always include all the scratchpad register values. In addition, if the MAR bit is 1, the contents

```
PC    = R[16]
PCW   = R[17]
B11   = R[18]
RSP   = R[20]
1     = R[21]
255   = R[22]
     .
     .
     .
begin
  1 := INV(B11);       ! initialize registers containing constants
  1 := BOR(1,B11);
  B11 := LSHIFT(1+1);
  B11 := INV(B11);
  1 := RSHIFT(B11);
     .
     .
     .
end
```

Figure 2: Sample Micro-code.

of memory from the address in the `MAR` up to the address `MAR`+`MBR` will be printed.

Halting

When your program wants the micro-architecture to halt, it should go into a tight loop of the form:

```
halt: go to halt;
```

The micro-interpreter will detect this obvious loop and halt.

Micro-code format

The format of *Male* code is identical to MAL code except that 1) statement labels containing alphabetic characters are allowed and encouraged, 2) the assembler treats all characters on a line after a ‘`!`’ as a comment, 3) each microinstruction must end with a semi-colon, and 4) microinstructions follow a line containing only the keyword `begin` and end with a line containing only the keyword `end`. Case is ignored in all labels and keywords. A sample of some typical micro-assembly language code is shown in Figure 2.

Running the micro-assembler

To use the micro-assembler, use the command

```
    male -L outfilename infilename
```

where *infilename* is the file containing your micro-assembly language program. The switch `-L` indicates that *outfilename* is the name that should be given to the file to which a listing of your program will be written. If you do not want a listing, leave off the `-L` switch, and the output file name designation. The micro-assembler will print any error messages on your screen and produce a file named `hmem` (meaning host memory) in which it will store an executable copy of your microcode.

Running the *Mice* Interpreter

To run your microprogram type the command

```
    mice
```

The mice program will read the microcode in `hmem`, a memory image for the target machine in `tmem` (target machine memory) if that file exists (the 34000 assembler will leave its output in `tmem`)[2] and the input for the target machine in `tinput` (if that file exists) and produce a `toutput` file corresponding to the target machine's screen output.

The *Mice* interpreter provides a set of debugging commands through which one can monitor the execution of a running micro-program. Debugging commands are read from the interpreter's standard input and debugger output is written to the standard output. All the commands accepted by the debugger are described below.

Before the interpreter begins executing a micro-program it enters the debugger and requests that the user enter a debug command. This gives one the opportunity to enable tracing or set breakpoints before your program begins running. The interpreter re-enters the debugging mode whenever it encounters a breakpoint, reaches the end of your program or is interrupted by your typing control-C.

We describe a few of the more useful debugging commands below. It is important to note, however, that other commands are available and brief documentation is available with the help command.

The Help Command

The *help* command lists each of the commands available in the debugger, along with a line of documentation. At the time of this printing, the mice help function generated the following output:

```
The following commands are available in mice.  CAPS indicate abbreviations.

Help            - print this message
```

[2]Students will find a number of useful assembly programs in the directory `/usr/cs-local/share/cs237/`. See the `README` file in that location.

```
Continue        - run the microprogram
Last n          - print last n instructions executed (up to 99)
Registers r     - dump register a
Registers l h   - dump registers between l and h
Memory a        - dump memory at address a
Memory a b      - dump memory between addresses a and b
RWatch r        - watch register r for change
RWatch r t      - watch register r for change to target t
MWatch l        - watch memory location l for change
MWatch l t      - watch memory location l for change to target t
LIst a b        - list program between addresses a and b
Profile a b     - dump profile information for lines a-b
Trace           - trace instructions as they’re executed (currently off)
Verbose         - trace verbosely (currently on)
Quit/HAlt/Exit  - leave mice
Break           - list breakpoints
Break a         - set (or clear) breakpoint at mpc a

Values may be expressed in decimal (49), octal (043),
hexadecimal ($fffe), or binary (ˆ1001).
Symbols for registers and line labels may also be used.
If symbol begins with a decimal digit, it must be preceded
by a backslash (\63).
```

The Step Command

The `step` command instructs the debugger to execute a specified number of micro-instructions and then return to debug mode. The general form of the command is

`step` *n*

where *n* is the number of instructions to execute. If the operand is omitted one instruction is executed. *Simply pressing return in response to the debug prompt is equivalent to the command* `step 1`.

The Continue Command

The `continue` command instructs the debugger to resume normal execution of the micro-program. Actually, to avoid wasting time running infinite loops, the `continue` command is treated as a step command with a very large operand.

The Trace Command

The `trace` command is used to enable or disable micro-program tracing. If tracing is disabled, execution of this command enables it. If tracing is enabled, this command disables tracing.

The Verbose Command

The `verbose` command is used to control the form of output produced when tracing is enabled. Like the `trace` command the `verbose` command toggles the state of the interpreter between verbose and non-verbose output modes. In verbose mode, the trace output includes the source code for the line executed. In non-verbose mode, only the micro-instruction number is printed.

The Register Command

The `register` command is used to display the values of *Mice* machine registers. The general form of the command is

`register` *low-reg high-reg*

This command causes the debugger to display the current values in the scratchpad registers numbered between *low-reg* and *high-reg*. If only one operand is included only that register is displayed. Within this command, the interpreter treats the `MAR` as register number 32 and the `MBR` as register number 33. A request to display `R[31]` causes the debugger to print the value of `R[RP]`.

The Memory Command

The `memory` command is used to display the values of words in the machine's main memory (i.e. not the micro-store). Its general form is

`memory` *low-addr high-addr*

This command causes the debugger to display the current values in memory words *low-addr* through *high-addr*. If only one address or label is included, the word at that address is displayed.

The Breakpoint Command

The `breakpoint` command is used to set and remove breakpoints in the micro-program. Its general form is

`breakpoint` *micro-instr-number*

Its operand is interpreted as the address or label of a micro-instruction in micro-store. If this operand is omitted, the address of the next micro-instruction to be executed is used. If no breakpoint is currently set on the instruction specified by the operand, this command sets a breakpoint at that instruction. If a breakpoint is already set at the selected instruction, this command removes the breakpoint.

The Watchpoint Commands

The `mwatch` and `rwatch` commands set breakpoints that occur when the indicated ranges of memory or registers change value. These commands are useful for debugging logic associated with registers or memory locations appear to change unexpectedly. Their forms are:

`mwatch` *low-addr high-addr*

`rwatch` *low-reg high-reg*

Use of these commands on large numbers of locations can slow the simulator substantially. Setting a watchpoint on a location that is currently being watched will cause the watchpoint to be canceled. When either command is used with no arguments, associated watchpoints are listed.

The Last Command

While executing a micro-program, the *Mice* interpreter remembers the last instructions executed. In debug mode, the command

`last` *count*

causes it to display a trace of the last ‘count’ instructions executed. If *count* is omitted, it displays a trace of all of the instructions it has saved.

The Halt Command

This command terminates the execution of the interpreter.