USERS' GUIDE TO OSIAC SIMULATOR (SIM)
A Logic Simulator for ECE 5362

ECE Department, The Ohio State University
Last updated in July 2016

I. OVERVIEW

Once the data paths are determined for a given processor, the machine instructions must be implemented by writing the register transfers or microinstructions for each instruction. While some instructions have an obvious implementation, some are more involved. It would be useful to use a computer to test whether your microinstructions really correctly implement each instruction. This is the purpose of the OSIAC Simulator (called SIM in short hereinafter), a logic simulator written for use in ECE 5362.

You will be given an architecture and SIM will have a file for the contents of the memory (program and data) of the machine that is being simulated. Basically you will provide SIM with your file of specifications for register transfers to implement the machine instructions, the associated control lines which must be activated, and the next state to be executed. SIM will use these microinstructions and execute each state and show you the changes made to the registers. When the processor executes the HALT machine instruction, you will get a memory dump of the simulated machine which will show you whether your implementation is correct. It is also possible for you to load a test program in the memory of the machine being simulated so that you can do your own diagnostic tests.

II. INPUT

The input to the computer simulation program consists of the microinstructions which pro- vide the specification of the control lines to be activated during specified states and information on branching of states. Comments are also allowed and recommended.

Comments

A comment is any line that starts with a non-blank character. The rest of such lines is not used by the simulator. Conversely, all other records should not start in the first column. A completely blank line, however, will also be treated as a comment.

The file consists of two kinds of records, one for control line specification and the other for branching information. They cannot be combined in one line.

Control Line Specification

Control line specification is done in records of the following form:

```
rt='string' list_of_control_lines
```

Here string is a string of characters of length up to 16 characters and describes the register transfer and list of control lines is a list of control lines. The control lines can optionally be assigned a value. If no value is assigned, the value is assumed to be a 1. Example:

```
st=3   rt= 'pc+1 -> pc'      imar
```

Here the state number st is assigned a 3, and imar is asserted, i.e, it is assigned a 1. Variables not in the list are given the value 0 so you only have to list asserted lines.

In general the control lines are different for each simulated system but there are several key lines that are common.

- st (State). This variable indicates the state number for this record. Although presented as a control line, it is actually a directive to the simulation program for record keeping. Each control line record must have a state number. The state numbers must be defined in increasing order, although states can be skipped. The first state of your fetch cycle must be state 0 and every time the fetch cycle is performed, the first state executed must be state 0. State numbers are decimal integers without a decimal point. The simulator has a limited table size so the maximum state number must be kept below a specified maximum.
- halt. This control line makes the computer halt. In your problems be sure to imple-ment the HALT instruction which asserts this control line in its execution cycle.

Branch Records

Each branch record has the following form:

```
cond = 'x'                value = val      nst  =  nextstate
```

where x is an allowable condition for a given processor. If the condition cond has the value val then the next state is nextstate.

- cond. A typical condition might be ir0, meaning the contents of Bit 0 of the Instruction Register is used to determine if branching occurs. The condition 'unc' means an unconditional branch.
- value. This is the number that the condition must match for a conditional branch to

occur. The value must be compatible with the condition and is expressed as a base 10 integer without a decimal point.

- nst. This is the next state executed if the condition matches the specified value.

Each of the branch records for a given state is evaluated in order to see if a branch should occur. If none of the branches are unconditional or none of the conditional branches have a matching value, then no branch occurs.

If cond is omitted, 'unc' defaults. If value is omitted, 1 is the default value. For example the record

```
nst = 3
```

is an unconditional branch to state 3 and the record

```
cond = 'z'     nst = 4
```

is a branch to 4 if z = 1. Note that even simple branch records must be on their own line and cannot be combined with control line records, even if the control line record is only a state number!

III. OUTPUT

The simulation program prints the output values of all registers in each state into a file. It is important to understand the timing convention used in all these computer designs. All flip-flops, unless specified otherwise, are negative-going edge-triggered flip-flops. See Figure 1 for a definition of the states. The change requested by a control line during state i occurs at the end of state i and can be seen in the register values for state (i + 1).

The program will print a dump of memory contents before and after execution of the simulated architecture. The memory contents are shown in both hex and decimal formats. In general, a decimal point indicates a decimal value and the absence of one indicates a hex value.

IV. ERROR MESSAGES

To make it easier for you to debug a design, a number of conditions are checked for mistakes.

- The input file must start with a control line record, not a branch record.
- The highest state number must be less than a specified maximum. The simulator has simple tables so this constraint is based on the limited table size.

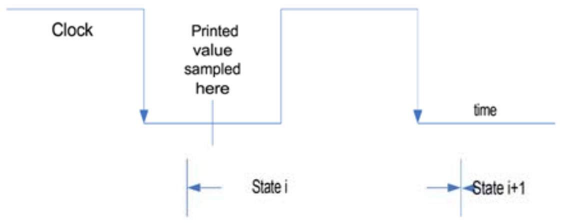

Figure 1: Timing convention used in simulator

- The actual number of states executed must be less than a preset maximum.
- Simulated memory is limited. An error indicates whether you have tried to access memory that is not being simulated.
- Executing an un-initialized state causes a termination. It is assumed that if you try to execute a state that you have never defined, that is clearly a mistake and the simulator stops. However, there is nothing wrong with executing null states. A null state is one that has been defined with an st value but has no control lines activated. These are useful in many cases.
- Each state must be numbered explicitly and the states must be defined in increasing order. The simulator will complain if a state number has been repeated or is less than the last state compiled. These features have been added because in the past typos in the state numbers caused the misnumbered states to overwrite some other states without any warning to the user.
- A number of syntax errors are checked. Examples include misspelled control lines or cond values, strings that have no closing, decimal values with non-numeric characters in them, etc. It is important to note that the simulator is rewritten each term with a different set of control lines, so the simulator only accepts control lines for your machine problems.
- In most cases an explicit line number is given in the error message so that you can use a text editor to go specifically to that line. In some cases the line count could be off by one. In counting lines, comments are counted so the line number should be the true line number in your specs.dat file.
- Like a high level language, the simulator has a compile and an execution phase. The simulator will try to complete the compile phase even if syntax errors occur but if any syntax errors occur, it won't try to do any execution. If an error is found in any line, that line is discarded and is not entered in the internal tables. Therefore, after the first round of errors are corrected, new errors may be disclosed. Many errors can only be caught in the execution phase.
- In addition, you may be able to make the simulation crash with some types of errors that are not anticipated, and although a great deal of work has gone into writing the simulator, there may still be bugs in the program. In many cases you may be able to see what data condition causes this situation and will be able to make them go away. Please let your instructor know of any such situations. In particular if you get any error message from the system about access violation or subscripts being out of range, please report them.

V. EXAMPLE

A simple example will help clarify how this system works. Suppose a single computer has control lines opc, imar, read, omdr, iir, and incpc to do the respective operations of putting the

PC on the bus, setting the MAR from the bus, reading memory, putting the MDR on the bus, setting the IR from the bus and incrementing the PC. Let's look at a sample input file:

```
st=0      rt='[pc] -> mar'        opc=1   imar=1
st=1      rt='[[mar]] -> mdr'     read=1
st=2      rt='[mdr] -> ir'        omdr=1 iir=1
st=3      rt='[pc]+1 -> pc'       incpc=1
        cond='ir'   value=1    nst=5
        cond='ir'   value=2    nst=10
        cond='ir'   value=3    nst=15
        cond='unc'             nst=50
```

The fi records are control line records and the last 4 are branch records for state 3. In state 3 the IR is decoded and the next state will be 5, 10, 15, or 50 depending on the opcode. Now look at the resulting output.

```
 st   pc   ac    x   sp   t1    z   t2   t3  mar  mdr   ir  reg xfer
  0    0    0    0    0    0    0    0    0    0    0    0  [pc] -> mar
  1    0    0    0    0    0    0    0    0    0    0    0  [[mar]] -> mdr
  2    0    0    0    0    0    0    0    0    0    1    0  [mdr] -> ir
  3    0    0    0    0    0    0    0    0    0    1    1  [pc]+1 -> pc
  5    1    0    0    0    0    0    0    0    0    1    1  --
State is un-initialized.      I quit.
 Hex memory dump (least significant word on left)
 Only lines with at least one nonzero value printed
     0 :     1    1    1    0    1    2    3    4
     8 :     5    6    7    0    0    0    0    0
```

Look at state 2 as an example. Here the user specified [mdr] → ir by causing the control lines omdr and iir to be asserted. Since the change occurs at the clock edge marking the end of the state, the new value for the IR can be seen in state 3. Because the input fi has only implemented the fetch cycle, an un-initialized state, state 5, is encountered which stops the simulation. This is an error which should go away when the machine instructions are completed and the HALT instruction is properly implemented.

The memory dump shows the contents of memory. The number before the colon is an address and the numbers after the colon are the contents of successive words of memory. In this example words 0, 1, and 2 all have 1's in them, M(3)=0 , M(4)=1, M(5)=2, M(6)=3, M(7)=4, etc.

VI. FILE USAGE BY SIM

input:

- specs.dat is your file of specifications to the simulator. The format was discussed above.
- memory.dat is the file for simulated memory contents of the processor. Logically these are the compiled instructions that you want to test. The format is limited but very simple. All lines in a memory.dat file fit one of two formats. The first kind of line is one which has a valid hex number in the first 4 columns. Call this type of line a hex format line. The other kind of line is one that does not have a valid hex number in the first 4 columns. Call this second kind of line a delimiter line.

  The first rule of memory.dat file is that they consist of 1 or more test sets. A test set is a contiguous set of hex format lines. The lines of a test set describe the original values of the AC, X, SP, PC and Condition Code registers, and the successive words of memory starting from address 0.

  The values of the AC, X, SP, and PC registers and the memory contents are 4 digit hex numbers. The four condition codes are put in one 4 digit number interpreted as a 4 bit binary number. The bits of this field are assigned from left to right to the CVZN bits, respectively.

  The second rule of this file format is that any number of delimiter lines can appear before, after, or between test sets, but not within a test set. At least one delimiter must appear between test sets as a separator. Additional ones can serve as comments.

  Spaces have a predefined interpretation by the Linux operating system. Spaces can be used in place of leading zeros in the 4 digit hex numbers, but the number must be right adjusted in the first four columns. A blank line is treated as a hex zero and cannot be used as a separator between test sets.

  Anything past the first four columns is ignored and therefore, can be used as comments. Therefore, it is possible to comment the text instructions within the lines of the test.

  Here is a sample memory.dat file:

```
----------cut here-------------------
this is a sample memory.dat file
starting test 1
0001 AC
0002 X
00FF SP
0000 PC
1100 CVZN
1001
0000
this is end of test 1
******************
```

```
this is test 2
FFFF AC
FFFF X
00FF SP
0000 PC
0000 CVZN
1001 ADD AC, X
0000
this is end of test 2
------cut here-----------------------
```

This sample file has two tests. Suppose 1001 is the instruction to ADD AC to the X register. The value of 0 in location 1 is the halt instruction. In the first test, by presetting the AC and X registers to 1 and 2, one can look for the answer 3 in the X register. Without the ability to preset registers, tests would have to be longer. For example, for this test, it would have required additional instructions to generate non-zero values to add. A second part of this first test is to see if the initial values of 1 for C and V are properly cleared, since no carry or overflow is generated in this add.

Test 2 is the same except for the operands. Here we should expect to see -3 in the X register (FFFD Hex) and the N condition code set.

The format of this file is, unfortunately, deceptively simple. In this example, the characters "AC" in the first line of the test set are actually a comment. Putting the AC value here is not an option since the first 5 lines are always the values to go into the AC, X, SP, PC, and condition codes, whether the comments are included or not. However, it is strongly recommended the comments especially to remember the order in which the condition code bits must be presented.xs

All test sets must have the first five lines to preset registers and condition codes, even though in most cases all zeros is sufficient. It should be easy however, to make a skeleton test file with these 5 lines already included. By copying this skeleton to a different name and filling in the actual instructions in the test, you can save a lot of typing. By including the comments "AC, X, SP, PC, and CVZN" in this skeleton file you will be able to read the resulting test file a great deal easier.

output:

- Screen output is mostly status information and error messages. The main part of the output is directly written to files.
- details.x is the output file showing the results of all register transfers in test x. This file provides a detailed description of how the simulated machine is operating at the microinstruction level. This file also provides a dump of memory before and after your microinstructions are executed. For successive tests, x will range through the single digit integers, then upper case letters, and then the lower case letters. This format was partially chosen because

shell commands will operate on files in this order. There is a maximum of 61 test sets in one test file.

- summary.x is the output file showing a summary of what each instruction did in test x. The simulator is actually taking a snapshot of the register contents every time state 0 is entered. This is the main reason why you must begin your fetch cycle in state 0. This file also shows memory writes when they happen.

VII. HOW TO RUN SIM ON ECE LINUX SERVERS

- File names are case sensitive. The correct names are all lower case.
- To run the simulator, first create specs.dat, the file of microinstructions, and memory.dat, the test program. Place them in one of your directories in your Linux account.
- A shell script has been written which you can use to simplify running the simulator. This script allows you to specify arbitrary file names for your microinstruction and test files rather than using the fixed names specs.dat and memory.dat. The script makes a copy of the names you specify into these fixed names, so if your first microinstruction file was named specs.dat and you later use a file specs1.dat, the original specs.dat would be overwritten.

Here is the format of the script:

```
/share/ECE/wang.3596/osiac/dosim4 -options microinstruction_file test_file
```

The three options are -debug, -g, and -wcc. One of the three must appear.

The *debug* option is the original version of the program and it generates a full set of details.dat and summary.dat files. All registers are shown.

The *g* option is the one is used for grading. It suppresses printing the details.dat files and it only prints general purpose registers and condition code values. The other registers don't have a defined value at the end of instructions.

The *wcc* option is a version of the grading option that prints output without the condition codes. In some cases, this option has been used to see if an instruction works while ignoring whether the condition codes are right.

The microinstruction file is the file with the format previously described for specs.dat. This field must be present.

The test_file is the file with the format described for memory.dat. If this last field is omitted, the script assumes you already have a memory.dat file.

Because repeated use of this script can generate a large number of files, the script will prompt you to see if it should remove all summary.* and details.* files from previous runs.

However, even if you say "n" the new files will overwrite the old ones.

You can make a copy of this script into your own account with the command (note that there is a dot at the end)

```
cp /share/ECE/wang.3596/osiac/dosim4 .
```

Then you don't have to type the tilde and my name each time. Of course, if instructor makes corrections to the script, you won't be running the latest version. You can even change it if you want to modify features. For example, if you always want to remove old versions, take out the prompt.