

USERS' GUIDE TO SIM

A Logic Simulator for ECE 662

Prof. C. Klein

(Modified for Autumn Quarter 2007 by Prof. D. Orin)

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 SIM, a logic simulator written for use in ECE 662.

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 provide 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 implement 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.

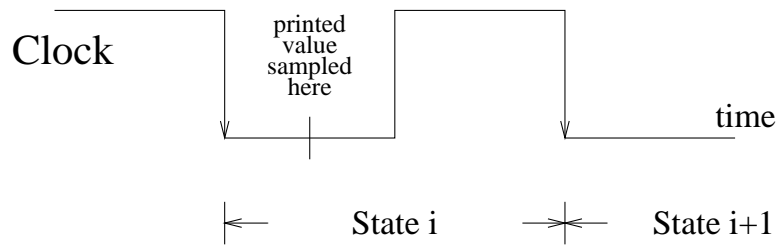


Figure 1: Timing convention used in simulator

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.
- 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 I didn't anticipate 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. I would appreciate it if you would let me 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, I would definitely like to see 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 first 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 $[mdr] \rightarrow ir$ 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 file 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. This file should consist of hex contents of successive words of memory stored one per line. These are interpreted as successive contents starting in location 0 and continuing upward. The simulator expects the 4 digit hex number to be in the first four columns. You must either enter the 4 hex characters or leading zeros can be suppressed but the remaining digits must be right justified in the first 4 columns.

output:

- Screen output is mostly status information and error messages. The main part of the output is directly written to files.
- `details.dat` is the output file showing the results of all register transfers. 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.
- `summary.dat` is the output file showing a summary of what each instruction did. 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. MISCELLANEOUS NOTES

- 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. From that directory then enter:

```
~orin/osiac/sim4
```