PennSim Simulator Manual

Overview

This document describes the Java-based PennSim simulator (and assembler) developed at the University of Pennsylvania. It does not teach assembly programming or debugging techniques. Instead, it simply describes the features available in the simulator.

PennSim provides an interface to executing LC-3 programs on a simulated LC-3 machine. The interface allows users to observe and effect changes to devices (such as the graphical video display, console text output, and keyboard input). It also allows users to control or limit the execution of a programming running on the simulator (for example, we might want to execute a program one instruction at a time) and observe or modify the state (memory and registers) of the machine.

PennSim provides both a graphical and text-based interface (the later is specified via the '-t' flag). Our expectation is that only the graphical interface will be used, so this document does not describe the text interface explicitly. Nevertheless, most of the functionality of the graphic interface is available (via the "Command Line") to the text interface.

Running PennSim

For instructions on downloading and running the simulator, please see the PennSim Guide.

PennSim Summary

the PennSim window consists of five components: Menus, Controls, Registers, Memory, and Devices. Each is described below.

Menus

There are only two menus available ("File" and "About"). The "Open .obj File" menu item under "File" is used to load an object file into the machine. Note that this can also be achieved via the "Command Line" (see below). The "Open Command Output Window" menu item under "File" opens a window that mirrors the context of the "Command Line Output" panel (see Controls section, below). This is often useful because the Command Output Panel is somewhat small. Finally, the "Quit" menu item under "File" causes the simulator to terminate (after confirmation).

The "About" menu has only one item - "Simulator Version." Use this to make sure you're running the most current version of the simulator that has all the latest bug fixes, features, etc.

Controls

The "Controls" section of PennSim (appearing at the top of the simulator window) is used to control and monitor the simulated machine. It contains four components: Execution Buttons, Machine Status Indicator, Command Line, and Command Line Output panel. Each are described below.

Execution Buttons

The Execution Buttons control the execution of a program in PennSim.

The "Next" button executes the instruction at the current PC and stops when the PC == (current PC + 1). If the instruction at the current PC is a subroutine call instruction (i.e., JSR, JSRR) or a trap instruction, the machine will execute instructions until the PC points to the instruction *after* the JSR/JSRR/TRAP at which "Next" was called. "Next" is useful for walking through a program one instruction at a time, without going into subroutines and trap handlers.

The "Step" button is similar to "Next", but it will stop execution again after one instruction has been executed. Thus, "Step" follows execution *into* called functions and traps.

Both "Next" and "Step" follow branch instructions.

As an example of the results of "Next" versus "Step", consider the following code snippet:

Address	Label	Instruction
x3000	START	AND R0,R0,#0
x3001		JSR FUNCTION
x3002		NOT R0,R0
x3003		BRz START
x3004		HALT
...		
x4000	FUNCTION	ADD R0,R0,#1
x4001		RET

If we set the `PC` to `x3000` and hit "Next", we will execute the `AND` go to the `JSR` instruction. Hitting "Next" again will execute everything in the function call, but the simulator won't stop until we reach the `NOT` instruction, i.e. after the function has returned. Hitting "Next" again will follow the branch and take us to the START label.

If we set the `PC` to `x3000` and hit "Step", we will go to the `JSR` instruction, as before. Hitting "Step" again will take us to the `ADD` instruction inside the function call. Hitting "Step" one more time takes us to the `RET` instruction, and we have to hit "Step" a fourth time to get to the `NOT` instruction, after the function call. Hitting "Step" again will follow the branch and take us to the START label, just like "Next" did.

The "Continue" button starts execution with the instruction at the current PC. The machine will continue to execute instructions until the machine halts, a breakpoint (see below) is encountered, or the "Stop" button is pressed.

The "Stop" button stops execution of the machine.

Machine Status Indicator

The Machine Status Indicator (located to the right of the Execution Buttons) indicates the current state of the machine. "Halted" means the machine is halted (i.e., the "clock enable bit" of the Machine Control Register is cleared). "Suspended" means that the machine is not halted, but neither is it running. In this state you can examine and change the state of the machine, then resume execution via the "Next", "Step", or "Continue" buttons. "Running" means that the machine is actively executed instructions.

Pressing the "Next", "Step", or "Continue" buttons will cause the machine to leave the "Halted" machine state (i.e., the Machine Status Indicator will change to "Running" then "Suspended"). If "Stop" is pressed while the Machine Status Indicator is "Running", it will change to "Suspended."

Command Line

The Command Line (located immediately below the Execution Buttons) is used to specify commands to control and monitor the machine. Commands are typed on the command line and the command's output (if any) appears in the Command Line Output panel (below). See the Control Commands section (below) for a description of all the available commands.

Command Line Output panel

The Command Line Output panel displays the output (if any) of the commands entered into the Command Line. If you resize the PennSim window, the Command Line Output panel grows and shrinks. This is useful for making the Command Line Output panel larger so that you may better view the output of commands that produce a lot of output (e.g., "as"). If you are using a small monitor (e.g., 1024x768), open the "Command Output Window" (via the "File" menu). The Command Output Window simply mirrors the text in the Command Line Output panel.

Registers

The register panel is located on the left half of the screen. The processor has eight general purpose registers, and has a few special registers (PC, MPR, PSR, CC). The value of each register is right next to its label, and the value can be modified by double-clicking on the value and typing in a new value. Values can either be in decimal, or in hexadecimal (hexadecimal numbers must be prefixed with an 'x').

The general purpose registers are freely accessible throughout the entire program. Whenever an executed instruction changes the value of a register, it will automatically be updated in the Registers display panel. It is convenient to be able to monitor the values of registers for debugging purposes.

The special registers on the other hand, cannot be directly referenced by the program and special instructions must be used to work with them. It is possible to modify their values by hand in the simulator though.

The **PC**, or Program Counter, indicates the address of the next instruction to be executed.

NOTE: When PennSim first starts up, the PC is automatically set to address x0200. This will generally be the location where the operating system begins. Generally we will give you an operating

system to load into the machine, but we may also ask you to write parts of the operating system on your own. When the operating system finishes executing, it will then transfer control to the user program by jumping to location x3000. This means that **all your user programs should begin execution at address x3000!** More specifically, unless you are writing the operating system, make sure the first line of your program is .ORIG x3000.

The **MPR** is the Memory Protection Register. Each bit in the MPR controls whether instructions in a given memory range can be executed while in user mode (see **PSR** below) - 1 means that execution is allowed in a memory range in user mode, 0 means that it is only allowed in supervisor mode. Trying to execute code in a region for which the MPR doesn't allow execution results in an exception and will halt execution. Since the MPR is 16 bits, and the address space has 2^16 = 65,536 memory locations, each bit of the MPR controls 4096 (x1000) memory locations. The table below shows which regions of memory each MPR bit controls.

MPR Bit	Memory Region
MPR[0]	x0000 - x0FFF
MPR[1]	x1000 - x1FFF
MPR[2]	x2000 - x2FFF
...	...
MPR[15]	xF000 - xFFFF

The **PSR**, or Process Status Register, indicates whether the machine is operating in *supervisor mode* or *user mode*. If supervisor mode is enabled, PSR[15] is 1. Supervisor mode is enabled only for the operating system code, and it allows access to the different devices available to the machine (by allowing access to their memory-mapped regions - see MPR above). PSR[10:8] specify the priority level of the process being executed. PSR[2:0] contain the bits for the condition codes (CCs). PSR[2] is N, PSR[1] is Z, PSR[0] is P.

The **CCs**, or condition codes, are the 3 low-order bits of the PSR that give sign information of the most recently executed instruction that updated the codes, letting you determine whether the value was Negative, Zero, or Positive. These are used by the BR instruction in determining when to branch. The instructions that update the CCs are: ADD, AND, LD, LDI, LDR, LEA, and NOT.

Memory

The memory locations are on the right half of the screen. Each row represents a location in memory, and the row will tell you: if there is a breakpoint set at the location, the actual address (and any labels that might exist there), the value, and what instruction the value represents. Only the value of the memory can be changed by double clicking on the current value (similar to changing register values).

The following table summarizes how memory space is mapped in the machine:

Address Range	Usage
x0000 - x00FF	Trap Vector Table
x0100 - x01FF	Interrupt Vector Table
x0200 - x2FFF	**Operating System**
x3000 - xBFFF	**User code & stack**

xC000 - xFDFF	Video output
xFE00 - xFFFF	Device register addresses

Devices

A number of devices are available to the simulator. The simulator uses memory-mapped device architecture, so accessing a device is just like accessing any other memory location. Following is a table that summarizes the device locations:

Address	Device Register	Usage
xFE00	KBSR	Keyboard Status Register: when KBSR[15] is 1, the keyboard has received a new character.
xFE02	KBDR	Keyboard Data Register: when a new character is available, KBSR[7:0] contains the ASCII value of the typed character.
xFE04	DSR	Display Status Register: when DSR[15] is 1, the display is ready to receive a new character to display.
xFE06	DDR	Display Data Register: when the display is ready, the display will print the ASCII character contained in DDR[7:0].
xFE08	TMR	Timer Register: TMR[15] is 1 if the timer has gone off, and 0 otherwise.
xFE0A	TMI	Timer Interval Register: the number of milliseconds between timer ticks. Setting this to 0 disables the timer, and setting it to 1 sets the timer to generate "ticks" from "." (period) characters read from the current Text I/O Device (either user input or a file)
xFE12	MPR	Memory Protection Register: see **Registers** above.
xFFFE	MCR	Machine Control Register: see **Registers** above.

In addition to these devices, the **video output** is also memory-mapped from address location xC000 to xFDFF. The video display is 128 by 124 pixels (15,872 pixels total) and the coordinate system starts from (0,0) at the top left corner of the display.

Since each row is 128 pixels long, in order to find the location exactly one row below a given location, at x0080 to it. For example, if you are outputting to pixel (3, 10), whose memory location is xC18A, then one row immediately below it would be xC20A (=xC18A + x0080).

As a general rule, this is the formula to find the memory location associated with a given (*row*, *col*):

addr = xC000 + *rowx0080 + *col***

Each video output memory location represents one pixel, which means that the value it contains must be formatted as a pixel would be (i.e. RGB format):

[15]	[14:10]	[9:5]	[4:0]
0	RED	GREEN	BLUE

A value like x7FFF (or xFFFF would work - bit 15 is actually ignored) in location xC000 would output a white dot at (0,0). Here are a few common colors:

Pixel Code	Color
x7FFF	White
x0000	Black
x7C00	Red
x03E0	Green
x001F	Blue
x3466	Puce

Control Commands

Below are all the commands that are available on the Command Line.

`h[elp] usage: h[elp] [command]`
Prints a list of all available commands and their syntaxes. Specifying a command will give you a brief description of the command.
Example: `help list` will display help using the `list` command.

`as - usage: as [-warn] <filename>`
Assembles <filename> showing errors and (optionally) warnings, and leaves a .obj file in the same directory.
Example: `as -warn breakout.text`
Example: `as -warn breakout.asm`

`b[reak] - usage: b[reak] [ set | clear ] [ mem_addr | label ]`
Sets or clears break point at specified memory address or label.
Example: `break set x3000` will set a break point at memory location `x3000`.
Example: `break clear LABEL` will clear a previously set breakpoint at the location pointed to by `LABEL`.

`c[ontinue] - usage: c[ontinue]`
Continues running instructions until next breakpoint is hit.

`check - usage: check [ count | cumulative | reset | PC | reg | PSR | MPR | mem_addr | label | N | Z | P ] [ value | label ]`
Verifies that a particular value resides in a register or in a memory location, or that a condition code is set. `check count` prints the number of checks that passed and failed since the last invocation of `check reset`.
Samples:
`check r0 17` checks that register `r0` has the value `17`.
`check PC LABEL` checks if the PC points to wherever `LABEL` points.
`check LABEL x4000` checks if the value stored in memory at the location pointed to by `LABEL` is equal to `x4000`.
`check x4000 LABEL` checks if the value stored in memory at `x4000` is equal to the location pointed to by `LABEL` (probably not very useful). To find out where a label points, use `list` instead.

`clear - usage: clear`
Clears the commandline output window. Only available in GUI mode.

`d[ump] - usage: d[ump] [-check | -coe | -readmemh] from_mem_addr to_mem_addr dumpfile`
Dumps a range of memory values (range endpoints can be expressed in hex or decimal) to dumpfile as raw values (1 per line). The memory values themselves are in hex. Various flags modify the dumped output format:
`-check`: dump memory as a series of `check` commands that can be run directly as a sript.
`-coe`: dump memory as a .coe file, suitable for passing to the Xilinx CORE generator as the initial contents of a memory.
`-readmemh`: dump memory in a format that can be read by the Verilog system call $readmemh().
Example: `dump -check x100 x150 addresses.txt`
Example: `dump -coe 20 42 memory.coe`

`input - usage: input <filename>`
Specifies a file to read the input from instead of keyboard device (simulator must be restarted to restore normal keyboard input).
Example: `input keystrokes.txt`

`l[ist] - usage: l[ist] [ addr1 | label1 [addr2 | label2] ]`
Lists the contents of memory locations (default address is PC. Specify range by giving 2 arguments).
Example: `list LABEL` will list the contents of memory that is pointed to by `LABEL`.
Example: `list x3000 x300A` will list the contents of the 11 memory locations between `x3000` and `x300A`.

`l[oa]d - usage: l[oa]d <filename>`
Loads an object file into the memory.
Example: `load breakout.obj`

`n[ext] - usage: n[ext]`
Executes the next instruction.

`p[rint] - usage: p[rint]`
Prints out all registers, PC, MPR and PSR.

`quit - usage: quit`
Quit the simulator.

`reset - usage: reset`
Resets the machine and simulator.

`s[tep] - usage: s[tep]`
Steps into the next instruction.

`script - usage: script <filename>`
Specifies a file from which to read commands.
Example: `script myscript.txt`

`set - usage: set [ PC | reg | PSR | MPR | mem_addr | label ] [ value | N | Z | P ]`
Sets the value of a register/PC/PSR/label/memory_location or set the condition codes individually.
Example: `set PC x3000` sets the `PC` to point to `x3000`.
Example: `set r1 17` sets register `r0` to have the value `17`.
Example: `set x4000 -5` sets memory location `x4000` to the value `-5`.
Example: `set LABEL x0A` sets the value of the memory location pointed to by `LABEL` to be `x0A`.
Example: `set N` sets the `N` condition code.

`stop - usage: stop`
Stops execution. You can enter this command after you have started execution with `continue` to stop execution.

`trace - usage: [on trace_file | off]`
For each instruction executed, this command dumps a subset of processor state to a file, to create a trace that can be used to verify correctness of execution. The state consists of, in order, (1) PC, (2) current insn, (3) regfile write-enable, (4) regfile data in, (5) data memory write-enable, (6) data memory address, and (7) data memory data in. These values are written in hex to trace_file, one line for each instruction executed. Note that trace files can get very large very quickly! Sometimes a signal may be a don't-care value - if we're not writing to the regfile, the `regfile data in' value is undefined - but the write-enable values should allow don't-care signals to be determined in all cases.
Example: `trace on my.trace` enables tracing. Next, run some instructions. When you're done, run the command `trace off` to finalize the trace file.
