gpsim

T. Scott Dattalo

24 DECEMBER 1999

Contents

Introduction

gpsim is a full-featured software simulator for Microchip PIC microcontrollers distributed under the GNU General Public License (see the COPYING section).

gpsim has been designed to be as accurate as possible. Accuracy includes the entire PIC - from the core to the I/O pins and including ALL of the internal peripherals. Thus it's possible to create stimuli and tie them to the I/O pins and test the PIC the same PIC the same way you would in the real world.

gpsim has been designed to be as fast as possible. Real time simulation speeds of 20Mhz pics are possible.

gpsim has been designed to be as useful as possible. The standard simulation paradigm including breakpoints, single stepping, disassembling, memory inspect & change, and so on has been implemented. In addition, gpsim supports many debugging features that are only available with in-circuit emulators. For example, a continuous trace buffer tracks every action of the simulator (whether you want it or not). Also, it's possible to set read and write break points on values (e.g. break if a specific value is read from or written to a register).

gpsim is not fancy. It's a straight-forward text based command line interface. The CLI uses the same library as gdb (the GNU debugger). Thus command history, command editing, and command completion are all available. Other projects are beautifying gpsim with gui wrappers.

Chapter 1

gpsim - An Overview

If you don't care to wade through details, this chapter should help you get things up and running. The INSTALL and README files will provide more up-to-date information then this document, so please refer to those first.

1.1 Making the executable

gpsim's executable is create in a manner that's consistant with many of the other open source software:

command	description
tar -xvzf gpsim-0.x.0.tar.gz	expand the compressed tar file
./configure	Create a 'makefile' unique to your system
make	compile gpsim
make install	install gpsim

The last step will require root privileges.

1.1.1 Make Details - ./configure options

gui-less

The default configuration will provide a gui (graphical user interface). The cli (command line interface) is still available, however many people prefer just to use the cli. These hardy souls may build a command-line only interface by configuring gpsim:

```
./configure --disable-gui
```

As of version 0.17.0, you'll still need the gui to compile (I know, that sucks).

debugging

If you want to debug gpsim then you'll probably use gdb. Consequently, you'll want to disable shared libraries:

```
./configure --disable-shared
```

This will create one, huge monolithic executable with symbolic information.

1.2 Running

The executable created above is called: gpsim. The following command line options may be specified when gpsim is invoked.

```
gpsim [-h] [-p<device> [<hex_file>]] [-
c<stc_file>]
   -h           : this help list
   -p<device>   : processor (e.g. -
pp16c84 for the 'c84)
   <hex_file>   : input file in "intelhex16" for-
mat
   -c<stc_file> : startup command file
   -s<cod_file> : .cod symbol file
   -v           : gpsim version
```

1.3 Requirements

gpsim has been developed under Linux. It should build just fine under the popular Linux distributions like Redhat. I know of no port to windows. Two packages gpsim requires that may not be available with all Linux distributions are readline and gtk (the gimp tool kit). The ./configure script should tell you if these packages are not installed on your system or if the revisions that are installed are too old.

There are no minimum hardware requirements to run gpsim. Faster is better though!

gpasm, the gnupic assembler, is also very useful. gpsim will accept straight hex files, but if you want to do any symbolic debugging then you'll want to use the .cod files that gpasm produces. The .cod files are in the same format as the .cod files MPASM produces. Unfortunately, it's not possible to use MPASM produced .cod files with gpsim. This is because the .cod file has os dependent directory information (If I get a serious request, then I'll fix this limitation).

Chapter 2

Command Line Interface

The command line interface is fairly straight-forward. If you're familiar with gdb's cli, then you'll fill fairly comfortable with gpsim's cli. The table below summarizes the available commands.

command	summary
attach	Attach stimuli to nodes
break	Set a break point
clear	Remove a break point
disassemble	Disassemble the current cpu
dump	Display either the RAM or EEPROM
echo	echo "text"
help	Type help "command" for more help on a command
list	Display source and list files
load	Load either a hex or command file
node	Add or display stimulus nodes
processor	Add/list processors
quit	Quit gpsim
run	Execute the pic program
step	Execute one or more instructions
stimulus	Create a stimulus
symbol	Add/list symbols
trace	Dump the trace history
version	Display gpsim's version
x	examine and/or modify memory

The built in 'help' command provides addtional online information.

attach

```
attach node1 stimulus1 [stimulus2 stimulus_N]
```

attach is used to define the connections between stimuli and nodes. At least one node and one stimulus must be specified. If more stimuli are specified then they will all be attached to the node examples:

```
gpsim> node  n1                # De-
fine a new node.
gpsim> attach n1 porta4 portb0 # Con-
nect two I/O pins to the node.
gpsim> node                    # Dis-
play the new "net list".
```

break

The break command is used to set and examine break points:

```
break [c | e | w | r | wv | rv | wdt [loca-
tion] [value] ]
  options:
      c   - cycle
      e   - execution
      w   - write
      r   - read
      wv  - write value
      rv  - read value
      wdt - wdt timeout
          - no argument, dis-
play the break points that are set.
  examples:
      break e 0x20     # set an execu-
tion break point at address 0x20
      break wv 0x30 0  # break if zero is writ-
ten to register 0x30
      break c 1000000  # break on the one mil-
lion'th cycle
      break            # dis-
play all of the break points
```

clear

The clear command is used to clear break points:

```
clear bp_number
where bp_number is the number as-
signed to the break point
```

```
when it was created. (type "break" with-
out any arguments to
display the currently set break points.
```

disassemble

```
disassemble [[start | length] [end]]

  no arguments:  disassembles 10 instructions be-
fore and 5 after the pc.
  one argument:  disassemble [length] instruc-
tions after the pc.
  two arguments: disassem-
ble from [start] to [end].
```

disasemble does not use symbols. However, special function registers like status will be displayed by name rather than value.

dump

```
dump [r | e]

  dump r or dump with no options will dis-
play all of the file
  registers and special function registers.
  dump e will display the contents of the eep-
rom (if the pic
  being simulated contains any)
```

See the 'x' command for examining and modifying individual registers.

echo

The echo command is used like a print statement within configuration files. It just lets you display information about your configuration file.

help

By itself, help will display all of the commands along with a brief description on how they work. 'help <command>' provides more extensive online help.

list

```
list [[s | l] [*pc] [line_number1 [,line_number2]]]
```

```
Display the contents of source and list files.
Without any options, list will use the last spec-
ified options.
list s will dis-
play lines in the source (or .asm) file.
list l will display lines in the .lst file
list *pc will display ei-
ther .asm or .lst lines around the pc
```

The list command allows you to view the source code while you are debugging.

load

The load command is used to load either hex, configuration, or .cod files. A hex file is usually used to program the physical part. Consequently, it provides no symbolic information. .cod files on the other hand, do provide symbolic information. Unless you suspect there's something wrong with the way you're assembler or compiler is generating hex files, I'd recommend always using .cod files for debugging.

Configuration files are script files containing gpsim commands. These are extremely useful for creating a debugging enviroment that will be used repeatedly. For example

node

```
node [new_node1 new_node2 ...]
     If no new_node is speci-
fied then all of the nodes that have been
    defined are displayed. If a new_node is spec-
ified then it will be
    added to the node list. See the "at-
tach" and "stimulus" commands
    to see how stimuli are added to the nodes.

    examples:

    node           // display the node list
    node n1 n2 n3  // cre-
ate and add 3 new nodes to the list
```

processor

```
processor [new_processor_type [new_processor_name]] | [list] | [dump]
 If no new processor is specified, then the cur-
rently defined processor(s) will be dis-
played. To see a list of the processors sup-
```

```
ported by gpsim, type 'processor list'. To de-
fine a new processor, specify the proces-
sor type and name. To dis-
play the state of the I/O processor, type 'pro-
cessor pins' (For now, this will dis-
play the pin numbers and their current state.
examples:
processor // Display the processors you've al-
ready defined. proces-
sor list                   // Dis-
play the list of processors supported. proces-
sor pins
          // Display the processor pack-
age and pin state processor p16cr84 fred
          // Create a new processor. proces-
sor p16c74 wilma
          // and another. processor p16c65
          // Create one with no name.
```

quit

Quit gpsim.

run

Start (or continue) simulation. The simulation will continue until the next break point is encountered.

step

```
step [over | n]

     no arguments:  step one instruction.
 numeric argument:  step a number of instruc-
tions
  "over" argument:  step over the next instruc-
tion
```

symbol

stimulus

```
stimulus [[type] options]
  stimulus will create a sig-
nal that can be tied to an io port.
```

```
  Note that in most cases it is easier to cre-
ate a stimulus
  file then to type all this junk by hand.

      Supported stimuli:

square_wave | sqw  [pe-
riod p] [high_time h] [phase ph] [ini-
tial_state i]
  port port_name bit_pos end
      creates a square wave with a pe-
riod of "p" cpu cycles.
      If the high time is speci-
fied then that's the number of cycles
      the square wave will be high.
      The phase is with respect to the cpu's cy-
cle counter.
      The "port_name" and "bit_pos" de-
scribe where the stimulus
      will be attached.
asynchronous_stimulus | asy  [pe-
riod p] [phase ph]  [initial_state i]
   d0 [d1 d2 ... dn] port port_name bit_pos end
      creates an asyn-
chronous square wave with a period of "p" cpu
      cycles.  The phase is with re-
spect to the cpu's cycle counter.
      The "port_name" and "bit_pos" de-
scribe where the stimulus
      will be attached.


  examples:
      stimulus sqw pe-
riod 200 high_time 20 phase 60 port portb 0 end
      create a square wave stimulus that re-
peats every 200 cpu cycles,
      is high for 20 cpu cycles (and low for 200-
20=180 cycles). The
      first rising edge will occur at cy-
cle 60, the second at 260,...
      Bit 0 of portb will receive the stimulus
```

trace

```
trace [dump_amount]
      trace will print out the most re-
cent "dump_amount" traces.
      If no dump_amount is speci-
fied, then the entire trace buffer
      will be displayed.
```

version

```
version
    Display gpsim's version.
```

x

```
x [file_register] [new_value]
  options:
    file_register -
ram location to be examined or modified.
    new_value -
the new value written to the file_register.
    if no options are specified, then the en-
tire contents
    of the file registers will be dis-
played (dump).
```

Chapter 3

Graphical User Interface

gpsim also provides a graphical user interface that simplifies some of the drudgery associated with the cli. It's possible to open windows to view all the details about your debug environment. To get the most out of your debugging session, you'll want to assemble your code with gpasm (the gnupic assembler) and use the symbolic .cod files it produces.

3.1 Main window

Only settings/windows and help/about are usable.

3.2 Source Browsers

gpsim provides two views of your source: *'.asm'* and *'.obj'* browsers. The *'.asm'* browser is a color coded display of your pic source.

3.2.1 .asm Browser

When a .cod file with source is loaded, there should be something in this display.

There is an area to the left of the source, where symbols representing the program counter and breakpoints are displayed. Double clicking in this area toggles breakpoints. You can drag these symbols up or down in order to move them and change the PC or move a breakpoint.

A right button click on the source pops up a menu with six items (the word 'here' in some menu items denote the line in source the mouse pointer was on when right mouse button was clicked.):

Menu item **Description**

Select symbol. This menu item is only available when some text is selected in the text widget. What it does is search the list of symbols for the selected word, and if it is found it is selected in the symbol window. Depending of type of symbol other things are also done, the same thing as when selecting a symbol in the symbol window:

- If it is an address, then the opcode and source views display the address.
- If it's a register, the register viewer selects the cell.
- If it's a constant, address, register or ioport, it is selected in the symbol window.

Find PC This menu item will find the PC and changed page tab and scroll the source view to the current PC.

Run here This sets a breakpoint 'here'and starts running until a breakpoint is hit.

Move PC here This simply changes PC to the address that line 'here'in source has.

Breakpoint here Set a breakpoint 'here'.

Find text This opens up a searching dialog. Every time you hit the 'Find' button, the current notebook page is found and the source in that page is used. This dialog is similar to the one in netscape navigator, except for a combo widget containing previous search strings.

These are the keyboard bindings:

Key	**command**
s,S,F7	step
o,O,F8	step over
r,R,F9	run. (currently the only way to stop running is to press Ctrl-C in the terminal window where the cli is)
q,Q	quit

3.2.2 Opcode view - the .obj Browser

This a gui version of the disassemble command.

Double click on a line to toggle breakpoints.

This window has the same keyboard commands as the source browser.

3.3 Register views

There are two similar register windows. One for the RAM and one for the EEPROM data, when available.

Here you see all registers in the current processor. Clicking on a cell displays it's name and value above the sheet of registers. You can change values by entering it in the entry (or in the cell).

The following things can be done on one register, or a range of registers. (Selecting a range of registers is done by holding down left mouse button, moving cursor, and releasing button.)

- Set and clear breakpoints. Use the right mousebutton menu to pop up a menu where you can select set read, write, read value and write value breakpoints. You can also "clear breakpoints", notice the s in "clear breakpoints", every breakpoint on the registers are cleared.
- Copy cells. You copy cells by dragging the border of the selected cell(s).
- Fill cells. Move mouse to lower right corner of the frame of the selected cell(s), and drag it.
- Watch them. Select the "Add Watch" menu item

The cells have different backgroung colors depending on if they represent:

- File Register (e.g. RAM): light cyan.
- Special Function Registers (e.g. STATUS,TMR0): dark cyan
- aliased register (e.g. the INDF located at address 0x80 is the same as the one located at address 0x00): gray
- invalid register: black. If all sixteen registers in a row are invalid, then the row is not shown.
- a register with one or more breakpoints:red

gpsim dynamically updates the registers as the simulation proceeds. Registers that change contents between pauses in the simulation are highlighted with a blue foreground color.

3.4 Symbol view

This window, as its name suggests, displays symbols. All of the special function registers will have entries in the symbol viewer. If you're using .cod files then you'll additionally have file registers (that are defined in cblocks), equates, and address labels.

You can filter out some symbol types using the buttons in the top of the window, and you can sort the rows by clicking on the column buttons (the ones reading 'symbol', 'type' and 'address').
The symbol viewer is linked to the other windows. For example, if you click on a symbol and:

- If it is an address, then the opcode and source views display the address.
- If it's a register, the register viewer selects the cell.

3.5 Watch view

This is not a output-only window as the name suggests (change name?). You can both view and change data. Double-clicking on a bit toggles the bit. You add variables here by marking them in a register viewer and select "Add watch" from menu. The right-click menu has the following items:

- Remove watch
- Set register value
- Clear Breakpoints
- Set break on read
- Set break on write
- Set break on read value
- Set break on write value
- Columns...

"Columns..." opens up a window where you can select which of the following data to display:

- BP
- Type
- Name
- Address
- Dec
- Hex
- Bx (bits of word)

You can sort the list of watches by clicking on the column buttons. Clicking sorts list backwards.

Chapter 4

Controlling the Flow: Break Points

One of gpsim's strong features is the flexibility provided with break points. Most simulators are limited to execution type break points.

If you want to set break points on registers, on execution cycles, invalid program locations, stack over flows, etc. then you're usually forced to debug your code with an ICE.

4.1 Execution Break Points

An execution break point is one that will halt a running program when the program memory address at which it is set is encountered. For example, if you were debugging a mid-ranged PIC and wished to stop execution when ever an interrupt occurs, you could set a break point at program memory address 0x04:

```
gpsim> break e 4
```

(To be more precise, an interrupt doesn't have to occur for this break point to be encountered - errant code could have branched here too).

The break point occurs BEFORE the instruction executes. Other simulators such as MPLAB break after the instruction executes. In many cases this distinction is insignificant. However, if the break is set on a 'goto' or 'call' instruction, then it's convenient to stop before the branch occurs. This way it's easy to determine from where a brance occurred.

4.1.1 Invalid Instruction Break Points

gpsim automatically will halt execution if a program attempts to venture beyond its bounds. Program memory locations that are not defined by your source code will be

initialized with an 'Invalid Instruction'. These are quite visible when you disassemble the program.

4.2 Register Break Points

gpsim provides the ability to break whenever a register accessed, either read or written or both. Furthermore, it's possible to break whenever a specific value is written to or read from a register.

4.3 Cycle Break Points

Cycle break points allow the program to be halted at a specific instruction cycle. Suppose you have a 20 Mhz pic and want to break after one second of simulation time. You could set a break at the 5 millionth instruction cycle.[1]

[1] There are 4 clock cycles per instruction. Also, a future feature of gpsim will provide you with the ability to set break points in terms of seconds.

Chapter 5

What has happen?: Trace

Inspecting the current state of your program is sometimes insufficient to determine the cause of a bug. Often times it's useful to know the conditions that led up to the current state. gpsim provides a history or trace of everything that occurs - whether you want it or not - to help you diagnose these otherwise difficult to analyze bugs.

What's traced	notes
program counter	adresses executed
instructions	opcode
register read	value and location
register write	value and location
cycle counter	current value
skipped instructions	addresses skipped
status register	during implicit modification
interrupts	
break points	type
resets	type

The 'trace' command will dump the contents of the trace buffer.

A large circular buffer (whose size is hard coded) stores the information for the trace buffer. When it fills, it will wrap around and write over the old history. To sustain high performance, gpsim encodes all trace information into a 32 bit integer.

Chapter 6

Simulating the Real World: Stimuli

Stimuli are extremely useful, if not necessary, for simulations. They provides a means for simulating interactions with the real world.

The gpsim stimuli capability is designed to be accurate, efficient and flexible. The models for the PIC's I/O pins mimic the real devices. For example, the open collector output on port A of an PIC16C84 can only drive low. Multiple I/O pins may tied to one another so that the open collector on port A can get a pull up resistor from port B. The overhead for stimuli only occurs when a stimulus changes states. In other words, stimuli are not polled to determine their state.

Analog stimuli are also available. It's possible to create voltage references and sources to simulate almost any kind of real world thing. For example, it's possible to combine two analog stimuli together to create signals like DTMF tones.

6.1 How They Work

In the simplest case, a stimulus acts a source for an I/O pin on a pic. For example, you may want to simulate a clock and measure its period using TMR0. In this case, the stimulus is the source and the TMR0 input pin on the pic is the load. In gpsim you would create a stimulus for the clock using the stimulus command and connect it to the I/O pin using the node command.

In general, you can have several 'sources' and several 'loads' that are interconnected with nodes[1]. A good analogy is a spice circuit. The spice netlist corresponds to a node-list in gpsim and the spice elements correspond to the stimuli sources and loads. This general approach makes it possible to create a variety of simulation environments. Here's a list of different ways in which stimuli may be connected:

[1] Although, gpsim is currently limited to 'one-port' devices. In other words, it is assumed that ground serves as a common reference for the sources and the loads.

1. Stimulus connected to one I/O pin
2. Stimulus connected to several I/O pins
3. Several stimuli connected to one I/O pin
4. Several stimuli connected to several I/O pins
5. I/O pins connected to I/O pins

The general technique for implementing stimuli is as follows:

1. Define the stimulus or stimuli.
2. Define a node.
3. Attach the stimuli to the node.

More often then not, the stimulus definition will reside in a file.

6.1.1 Contention among stimuli

One of the problems with this nodal approach to modeling stimuli is that it's possible for contention to exist. For example, if two I/O pins are connected to one another and driving in the opposite directions, there will be contention. gpsim resolves contention with attribute summing. Each stimulus - even if it's an input - has an effect on the node. This effect is given a weight. When a node is updated, gpsim will simply add the weights of all the stimuli together and assign that numeric value to the node. A weight value of zero corresponds to no load. A large positive weight is used by a stimulus to drive the node positive, while a large negative weight is used to drive it negative.

Attribute summing is useful for pull up resistors. In the port A open collector / port B weak pull-up connection example, gpsim assigns a relatively small weight to the pull up resistor and a large negative weight to the open collector if it is active or no weight if it's not driving. Capacitive effects (which are not currently supported) can be simulated with dynamically changing weight values.

6.2 I/O Pins

gpsim models I/O pins as stimuli. Thus anywhere a stimulus is used, an I/O pin may be substituted. For example, you may want to tie two I/O pins to one another (for example, a port B pull up resistor to a port A open collector). gpsim automatically creates the I/O pin stimuli whenever a processor is created. All you need to do is to specify a node and then attach the stimuli to it. The names of these stimuli are formed by concatenating the port name with the bit position of the I/O pin. For example, bit 3 in port B is called portb3.

Here's a list of the types of I/O pin stimuli that are supported:

I/O Pin Type	Function
INPUT_ONLY	Only accepts input (like MCLR)
BI_DIRECTIONAL	Can be a source or a load (most I/O pins)
BI_DIRECTIONAL_PU	PU=Pullup resistor (PORTB)
OPEN_COLLECTOR	Can only drive low (RA4 on c84)

There is no special pin type for analog I/O pins. All pic analog inputs are multiplexed with digital inputs. The I/O pin definition will always be for the digital input. gpsim automatically knows when I/O pin is analog input

6.3 Synchronous Stimuli

Synchronous stimuli are periodic functions that are synchronized to the cpu's clock[2]. They are defined by the following parameters:

parameter	function
phase	The # of cycles (after start) before the stimulus starts
period	The # of cycles for one period
start	The cycle the stimulus becomes active
initial_state	The initial state of the stimulus
type	Analog or Digital

One cycle of the periodic function can be written like so

$$func(t) = something\ for\ t = 0 \ldots period$$

And then can be made periodic by:

$$func(t + period) = func(t)$$

which just says that the function has the same value at t and $t + period$.

The $phase$ allows the periodic function to be shifted by an arbitrary amount.

In some instances, you don't want the stimulus to be active until after a certain amount of time. The parameters $start$and $initial\ state$ describe how long the stimulus should wait before becoming active and what state the stimulus should be in during that waiting period.

By default, the synchronous stimulus is assumed to be digital. This can be overridden by specifying *analog.* In many cases, the context in which the stimulus is used makes the analog/digital choice clear. For example, a square wave is usually digital and a sine wave is usually analog. There are instances where you might want to change this. For example, [how would you create a sinusoidally PWM'd wave form??? add it here...]

[2]This will probably be made optional in the future. In other words, you'll get the choice to specify whether or not the 'synchronous' stimulus parameters are synchronized to the cpu or are absolute time values.

6.3.1 Square Wave

The simplest example of a synchronous stimulus is a square wave. All of the parameters described above still apply. There's one additional parameter for the square wave:

parameter	function
duty cycle	'high time' for a square wave

The synchronous stimulus begins with the 'initial state' at cycle 0. After 'phase' cycles happen, the stimulus changes states. After 'high time' more cycles, the state changes again. The frequency of the square wave is:

$$f_{sq} = \frac{1}{period} * \frac{f_{osc}}{4}$$

Where f_{osc} is the simulated oscillator frequency. The duty cycle is:

$$DutyCycle = \frac{hightime}{period}$$

Synchronous stimuli are useful as clocks. For example, it's possible to create a synchronous stimulus and attach it to the TMR0 input pin. Or another example may be the clock line in an I^2C interface.

Square Wave example

Here's a sequence of commands to create a square wave:

```
stimulus sqw
initial_state 1
start 10000
period 1000
high_time 300
phase 100
port portb 0
end
```

There's really just one command, 'stimulus', spread over several lines. In this example, the square wave has a period of 1000 cpu cycles and a 30% duty cycle (it's high for 300 out of the 1000 cycles). It will start off high and remain high until cycle 10000, the start cycle, after which it will go low. Then 100 cycles later (phase delay) it will go high. This is the 'phase shift'. 300 more cycles later, or cycle 14000 it will go low. Since the period is 1000 cycles, the square will repeat this sequence at cycle 11000, 12000, etc. So in terms of absolute cpu cycles, the stimulus starts off high and then goes low at cycle 10000, high at cycle 10100, low at 10400, high at 11100, low at 11400...

Or if you prefer, the state of the square wave can be expressed:

```
if(cpu cycle < start)
  state = initial_state;
else {
 cycle_in_period = (cpu cycle - start) % period;
 if( (cycle_in_period > phase) && (cy-
cle_in_period < (phase+high_time)) )
   state = 1;
 else
   state = 0;
}
```

6.3.2 Triangle Wave

In it's simplest form, you can described a triangle wave as a periodic wave that is composed of two line segments, one with a positive slope the other with a negative slope. It's sometimes also called a 'sawtooth' wave because the repeated rising and falling edges look similar to the cutting teeth on a hand saw. gpsim defines triangle waves the same way as square waves and adds two new parameters. In other words, all of the parameters that apply to square waves also apply to triangle waves and in addition you may also specify the positive and negative peaks. The traingle wave can be most easily envisioned (PICTURE!!!) when superimposed on a square wave. When the square wave is 'high' the triangle wave has a positive slope and when it's low the triangle wave has a negative slope. If the triangle wave has the same amplitude excursions as a square wave, then the triangle wave peaks exactly coincide with the square wave's high-to-low and low-to-high transitions.

6.3.3 Sine Wave

Doesn't exist yet.

6.4 Asynchronous Stimuli

Asynchronous stimuli are analog or digital stimuli that can change states at any given instant (limited to the resolution of the cycle counter). They can be defined to be repetitive too.

parameter	function
start	The # of cycles before the stimulus starts
cycles[]	An array of cycle #'s
data[]	Stimulus state for a cycle
period	The # of cycles for one period
initial state	The initial state of data[0]

When the stimulus is first initialized, it will be driven to the 'initial state' and will remain there until the cpu's instruction cycle counter matches the specified 'start' cycle. After that, the two arrays 'cycles[]' and 'data[]' define the stimulus' outputs. The size of the arrays are the same and correspond to the number of events that are to be created. So the event number, if you will, serves as the index into these arrays. The 'cycles[]' array define when the events occur while the 'data[]' array defines the states the stimulus will enter. The 'cycles[]' are measured with respect to the 'start' cycle. The asynchronous stimulus can be made periodic by specifying the number of cycles in the 'period' parameter.

Here's an example that generates three pulses and then repeats:

```
stimulus asyn-
chronous_stimulus # or we could've used asy
# The initial state AND the state the stimu-
lus is when
# it rolls over
initial_state 1
# all times are with respect to the cpu's cy-
cle counter
start_cycle 100
# the asynchronous stimu-
lus will roll over in 'period'
# cy-
cles. Delete this line if you don't want a roll over.
period 5000
# Now the cycles at which stimu-
lus changes states are
# specified. The initial cycle was speci-
fied above. So
# the first cycle specified below will tog-
gle this state.
# In this example, the stimulus will start high.
# At cycle 100 the stimulus 'begins'. How-
ever nothing happens
# until cycle 200+100.
200 0
300 1
400 0
600 1
1000 0
3000 1
# Give the stimulus a name:
name asy_test
# Finally, tell the command line inter-
face that we're done # with the stimulus
end
```

ds

6.5 Analog Stimuli

Analog Stimuli are really a subset or type of asynchronous or synchronous stimuli.

Chapter 7

Symbolic Debugging

Early versions of gpsim provided hard coded symbols like 'status' or 'porta'. As a version 0.0.12, gpsim provides symbolic debugging capabilities. So in addition to the hard coded symbols, you can now have access to all of the symbols defined by your assembler. This includes cross referencing the program memory to source and list files, access to registers defined within cblocks, access to program memory labels, and access to all of the constants that are defined (either with equates or #defines) by your program.

Chapter 8

Hex Files

The target code simulated by gpsim comes from a hex file, or more specifically an Intel Hex file. gpsim accepts the format of hex provided by gpasm and mpasm. The hex file does not provide any symbolic information. It's recommended that hex files only be used if 1) you suspect there's a problem with the way .cod files are generated by your assembler or compiler OR 2) your assembler or compiler doesn't generate .cod files.

Chapter 9

Theory of Operation

This section is only provided for those who may be interested in how gpsim operates. The information in here is 'mostly' accurate. However, as gpsim evolves so do the details of the theory of operation. Use the information provided here as a high level introduction and use the (well commented :]) source to learn the details.

9.1 Background

gpsim is written mostly in C++. Why? Well the main reason is to easily implement a hierarchical model of a pic. If you think about a microcontroller, it's really easy to modularize the various components. C++ lends itself well to this conceptualization. Furthermore Microchip, like other microcontroller manufacturers, has created families of devices that are quite similar to one another. Again, the C++ provides 'inheritance' that allows the relationships to be shared among the various models of pics.

9.2 Instructions

There's a base class for the 14-bit instructions (I plan to go one step further and create a base class from which all pic instructions can be derived). It primarily serves two purposes: storage that is common for each instruction and a means for generically accessing virtual functions. The common information consists of a name - or more specifically the instruction mnemonic, the opcode, and a pointer to the processor owning the instruction. Some of the virtual functions are 'execute' and 'name'. As the hex file is decoded, instances of the instructions are created and stored in an array called program_memory. The index into this array is the address at which the instruction resides. To execute an instruction the following code sequence is invoked:

```
program_memory[pc.value]->execute();
```

which says, get the instruction at the current program counter (pc.value) and invoke via the virtual function execute(). This approach allows execution break points to be easily set. A special break point instruction can replace the one residing in the program memory array. When 'execute' is called the break point can be invoked.

9.3 General File Registers

A file register is simulated by the 'file_register' class. There is one instance of a 'file_register' object for each file register in the PIC. All of the registers are collected together into an array called 'registers' which is indexed by the registers' corresponding PIC addresses. The array is linear and not banked like it is in the PIC. (Banking is handled during the simulation.)

9.4 Special File Registers

Special file registers are all of the other registers that are not general file registers. This includes the core registers like status and option and also the peripheral registers like eeadr for the eeprom. The special file registers are derived from the general file registers and are also stored in the 'registers' array. There is one instance for each register - even if the register is accessible in more than one bank. So for example, there's only one instance for the 'status' register, however it may be accessed through the 'registers' array in more than one place.

All file registers are accessed by the virtual functions 'put' and 'get'. This is done for two main reasons. First, it conveniently encapsulates the breakpoint overhead (for register breakpoints) in the file register and not in the instruction. Second, and more important, it allows derived classes to implement the put and get more specifically. For example, a 'put' to the indf register is a whole lot different than a put to the intcon register. In each case, the 'put' initiates an action beyond simply storing a byte of data in an array. It also allows the following code sequence to be easily implemented:

```
movlw   trisa   ;Get the address of tris
movwf   fsr
movf    indf,w  ;Read trisa indirectly
```

9.5 Example of an instruction

Here's an example of the code for the movf instruction that illustrates what has been discussed above. Somewhere in gpsim the code sequence:

```
program_memory[pc.value]->execute();
```

is executed. Let's say that the pc is pointing to a movf instruction. The ->execute() virtual function will invoke MOVF::execute. I've added extra comments (that aren't in the main code) to illustrate in detail what's happening.

```
void MOVF::execute(void)
{
  unsigned int source_value;

  // All instructions are 'traced' (discussed be-
low). It's sufficient
  //to only store the opcode. How-
ever, even this may be unnecessary since
  //the progam counter is also traced. Ex-
pect this to disappear in the
  //future...
  trace.instruction(opcode);

  // 'source' is a pointer to a 'file_register' ob-
ject. It is initialized
  //by reading the 'registers' ar-
ray. Note that the index depends on the
  //'rp' bits (actually just one bit) in the sta-
tus register. Time is
  // saved by caching rp as opposed to decod-
ing the status register.
  source = cpu->registers[cpu-
>rp | opcode&REG_IN_INSTRUCTION_MASK];

  // We have no idea which register we are try-
ing to access and how it
  //should be accessed or if there's a break-
point set on it. No problem,
  //the virtual function 'get' will re-
solve all of those details
  // and 'do the right thing'.
  source_value = source->get();

  // If the destination is W, then the construc-
tor has already initialized
  //'destination'. Otherwise the destina-
tion and source are the same.
  if(opcode&DESTINATION_MASK)
    destination = source;      // Re-
sult goes to source

  // Write the source value to the destina-
```

```
tion. Again, we have no idea
  // where the destination may be or
  // or how the data should be written there.
  destination->put(source_value);

  // The movf instruc-
tion will set Z (zero) bit in the status register
  //if the source value was zero.
  cpu->status.put_Z(0==source_value);

  // Finally, advance the pc by one.
  cpu->pc.increment();

}
```

9.6 Trace

Everything that is simulated is traced - *all* of the time. The trace buffer is one huge circular buffer of integers. Information is or'ed with a trace token and then is stored in the trace buffer. No attempt is made to associate the items in the trace buffer while the simulator is simulating a PIC. Thus, if you look at the raw buffer you'll see stuff like: cycle counter = ..., opcode fetch = ..., register read = ..., register write = ..., etc. However, this information is post processed to ascertain what happened and when it happened. It's also possible to use this information to undo the simulation, or in other words you can step backwards. I don't have this implemented yet though.

9.7 Breakpoints

Breakpoints fall into three categories: execution, register, and cycle.

Execution:

For execution breakpoints a special instruction appropriately called 'Breakpoint_Instruction' is created and placed into the program memory array at the location the break point is desired. The original instruction is saved in the newly created breapoint instruction. When the break point is cleared, the original instruction is fetched from the break point instruction and placed back into the program memory array.

Note that this scheme has zero overhead. The simulation is only affected when the breakpoint is encountered.

Register:

There are at least four different breakpoint types that can be set on a register: read any value, write any value, read a specific value, or write a specific value. Like the execution breakpoints, there are special breakpoint registers that replace a register object. So when the user sets a write breakpoint at register 0x20 for example, a new breakpoint object is created and insert into the file register array at location 0x20. When the simulator attempts to access register location 0x20, the breakpoint object will be accessed instead.

Note that this scheme too has zero overhead, accept when a breakpoint is encountered.

Cycle:

Cycle breakpoints allow gpsim to alter execution at a specific instruction cycle. This is useful for running your simulation for a very specific amount of time. Internally, gpsim makes extensive use of the cycle breakpoints. For example, the TMR0 object can be programmed to generate a periodic cycle break point.

Cycle break points are implemented with a sorted doubly-linked list. The linked list contains two pieces of information (besides the links): the cycle at which the break is to occur and the call back function[1] that's to be invoked when the cycle does occur. The break logic is extremely simple. Whenever the cycle counter is advanced (that is, incremented), it's compared to the next desired cycle break point. If there's NO match, then we're done. So the overhead for cycle breaks is the time required to implement a comparison. If there IS a match, then the call back function associated with this break point is invoked and the next break point in the doubly-linked list serves as the reference for the next cycle break.

[1] A call back function is a pointer to a function. In this context, gpsim is given a pointer to the function that's to be invoked (called) whenever a cycle break occurs.

COPYING

Index