gpsim

T. Scott Dattalo

30 October1998

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 ent ire 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

Commands

The command line interface is fairly straight-forward.

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
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
x	examine and/or modify memory

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                 # Define a new node.
gpsim> attach n1 porta4 portb0 # Connect two I/O pins to the node.
gpsim> node                     # Display the new "net list".
```

break

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

```
break [c | e | w | r | wv | rv | wdt [location] [value] ]
  options:
      c   - cycle
      e   - execution
      w   - write
      r   - read
      wv  - write value
      rv  - read value
      wdt - wdt timeout
          - no argument, display the break points that are set.
  examples:
      break e 0x20     # set an execution break point at address 0x20
      break wv 0x30 0  # break if zero is written to register 0x30
      break c 1000000  # break one the one million'th cycle
      break            # display all of the break points
```

clear

The clear command is used to clear break points:

```
clear bp_number
where bp_number is the number assigned to the break point
when it was created. (type "break" without any arguments to
display the currently set break points.
```

disassemble

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

  no arguments:  disassembles 10 instructions before and 5 after the pc.
  one argument:  disassemble [length] instructions after the pc.
  two arguments: disassemble 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 display all of the file
  registers and special function registers.
  dump e will display the contents of the eeprom (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.

load

Load

node

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

     examples:

     node           // display the node list
     node n1 n2 n3  // create and add 3 new nodes to the list
```

processor

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 instructions
  "over" argument:  step over the next instruction
```

symbol

stimulus

```
stimulus [[type] options]
  stimulus will create a signal that can be tied to an io port.
  Note that in most cases it is easier to create a stimulus
  file then to type all this junk by hand.

      Supported stimuli:

square_wave | sqw  [period p] [high_time h] [phase ph] [initial_state i]
  port port_name bit_pos end
      creates a square wave with a period of "p" cpu cycles.
      If the high time is specified then that's the number of cycles
      the square wave will be high.
      The phase is with respect to the cpu's cycle counter.
      The "port_name" and "bit_pos" describe where the stimulus
      will be attached.
asynchronous_stimulus | asy  [period p] [phase ph]  [initial_state i]
   d0 [d1 d2 ... dn] port port_name bit_pos end
      creates an asynchronous square wave with a period of "p" cpu
      cycles.  The phase is with respect to the cpu's cycle counter.
      The "port_name" and "bit_pos" describe where the stimulus
      will be attached.


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

trace

```
trace [dump_amount]
      trace will print out the most recent "dump_amount" traces.
      If no dump_amount is specified, then the entire trace buffer
      will be displayed.
```

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 entire contents
    of the file registers will be displayed (dump).
```

Chapter 2

Simulating the Real Worl: Stimuli

Stimuli are extremely useful, if not necessry, 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.

2.1 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 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. Stimulus connected to one I/O pin
2. Stimulus connected to several I/O pins

[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.

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, stimulus definition will reside in a file.

2.1.1 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.

2.2 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.

2.3 Synchronous Stimuli

Synchronous stimuli are square waves. They're called synchronous because their frequency is related to the cpu's cycle counter.[2] They are defined by the following parameters[3]:

parameter	function
phase	The # of cycles before the stimulus starts
high time	The # of cycles the stimulus is 'high'.
period	The # of cycles for one period of the square wave
initial state	The initial state of the stimulus

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.

2.4 Asynchronous Stimuli

Asynchronous stimuli are 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
data[]	An array of cycle #'s
period	The # of cycles for one period
initial state	The initial state of data[0]

Not complete...

[2]The cycle counter is an internal counter gpsim use to track of all of the PIC's timing. It's frequency is the same as the PIC instruction cycle frequency: Fcpu/4.

[3]Caution: these (misleading) names WILL e changing...

Chapter 3

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.

Chapter 4

Theory of Operation[1]

4.1 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.

4.2 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.)

[1]The TOO comes mostly from a document I wrote around version 0.0.2. While the basic concepts are fairly accurate, many of the inner details have changed.

4.3 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
```

4.4 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 below). It's sufficiently
  //to only store the opcode. However, even this may be unnecessary since
  //the progam counter is also traced. Expect this to disappear in the
  //future...
  trace.instruction(opcode);

  // 'source' is a pointer to a 'file_register' object. It is initialized
  //by reading the 'registers' array. Note that the index depends on the
```

```
    //'rp' bits (actually just one bit) in the status register. Time is
    // saved by caching rp as opposed to decoding the status register.
    source = cpu->registers[cpu->rp | opcode&REG_IN_INSTRUCTION_MASK];

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

    // If the destination is W, then the constructor has already initialized
    //'destination'. Otherwise the destination and source are the same.
    if(opcode&DESTINATION_MASK)
      destination = source;      // Result goes to source

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

    // The movf instruction 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();

  }
```

4.5 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.

4.6 Breakpoints

Breakpoints fall into two categories: execution & register.

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.

COPYING

Index