Documentation for CPU Simulators

Brent Seidel
Phoenix, AZ

April 15, 2024

Contents

Chapter 1

Introduction

This project includes simulators for some existing processors. It can be embedded into other code as a library, or used stand-alone with a command line interpreter. The intent of these simulators are to provide instruction level simulation and not hardware level. Generally, no attempt has been made to count clock cycles - instructions may not not even take the same relative amount of time to execute.

Interfaces are provided to allow the simulator to be controlled by and display data on a simulated control panel (see the Pi-Mainframe (`https://github.com/BrentSeidel/Pi-Mainframe`) project. These may be stubbed out or ignored if not needed.

Chapter 2

Simulators

2.1 General

Several simulators are available for use. Each simulator may also have variation. So, one simulator may provide variations for different processors in a family of processors.

Each simulator is based on an object that derives from the `simulator` object defined in the `BBS.Sim_CPU` package. A generic simulator interface is defined with some procedures or functions that must be defined by a specific simulator and some that may be defined, if needed. There are also a number of utility functions that are not expected to be overridden by a specific simulator.

The current design has memory included in the simulator instead of being an external device. Any I/O decoding is also handled inside the simulator. This means that any reading and writing of memory has to be done via routines defined by the simulator.

2.1.1 Data Types

Currently, processors with address and data busses up to 32 bits wide are supported. For the address bus, use the data type `addr_bus` and for the data bus, use `data_bus`. These are both defined as a 32 bit unsigned integer. Each simulator may use as many or as few of these bits as are needed. This means that if (as in most cases) a simulator doesn't define a full 4 GB of memory, the external program can try to read or write non-existent memory. This will probably cause an exception.

2.1.2 Initialization

```
--
--  Called first to initialize the simulator
--
```

```
procedure init(self : in out simulator) is abstract;
```

This should be called once at the beginning of the host program to initialize the simulator. The implementation is up to the simulators and it is possible that some simulators may not need any initialization.

2.1.3 Loading Data to Memory

The main routines for reading and writing simulator memory are:

```
--
--  Called to set a memory value
--
procedure set_mem(self : in out simulator;
                  mem_addr : addr_bus;
                  data : data_bus) is abstract;
--
--  Called to read a memory value
--
function read_mem(self : in out simulator;
             mem_addr : addr_bus) return
  data_bus is abstract;
```

The actual addressing and data bus used are defined by the specific simulator.

For loading bulk data into memory use:

```
--
--  Called to load data into the simulator.
--
procedure load(self : in out simulator; name : String)
   is null;
```

Its implementation is defined by the simulator. Typically it loads an Intel Hex file or Motorola S-Record file representing a memory image.

2.1.4 Running a Simulation

The `start` procedure is called first to specify a starting address for program execution.

```
--
--  Called to start simulator execution at a specific
--  address.  This is made null rather than abstract
--  so that simulators that don't use it don't need to
--  override it.
--
procedure start(self : in out simulator; addr : addr_bus)
   is null;
```

Then, each instruction is individually executed using the `run` procedure.

```
--
-- Called once per frame when start/stop is in the start
-- position and run/pause is in the run position.
--
procedure run(self : in out simulator) is abstract;
```

Certain conditions or instruction can cause a simulator to halt. The `halted` and `continue_proc` routines can be used to test for this condition and clear it.

```
--
-- Check if simulator is halted
--
function halted(self : in out simulator) return Boolean
   is (False);
--
-- This clears the halted flag allowing processing to
-- continue.
--
procedure continue_proc(self : in out simulator) is null;
```

From all of this, one can write the core of a simulator as follows:

```
--
-- A bunch of other stuff including defining some
-- simulator as "sim"
--
begin
   sim.init;
   --
   -- I/O devices can be added here...
   --
   sim.load("image.hex");
   sim.start(0); -- Start execution at address 0
   while not sim.halted loop
      sim.run;
   end loop;
end
```

2.1.5 Variants

Each simulator can support variants. This enables one simulator to support multiple CPUs in a family. Since the variants supported are unique to each simulator a universal data type cannot be used. Variants are identified by a `Natural` number. The following routines are used to get the number of variants supported by a simulator, the name of each variant, the currently selected variant, and to select a variant:

```
--
```

```
--  Called to get number of variants
--
function variants(self : in out simulator)
   return Natural is (1);
--
--  Called to get variant name
--
function variant(self : in out simulator; v : Natural)
   return String is abstract;
--
--  Called to get current variant index
--
function variant(self : in out simulator)
   return Natural is abstract;
--
--  Called to set variant
--
procedure variant(self : in out simulator; v : Natural)
   is abstract;
```

2.1.6 Other

A number of other functions are defined to support I/O devices, interfaces with a front panel, and other things. The I/O support routines will be discussed more in chapter 3 on I/O devices.

2.2 Example

The example simulator provides and example of using the simulator object interface. Its primary purpose is to blink the lights in interesting ways in the Pi-Mainframe (`https://github.com/BrentSeidel/Pi-Mainframe`) project. There are a number of different patterns selectable. Variants are defined for "Copy Switches", "Count", "16-Bit Scan", "16-Bit Bouncer", "Fibonacci Counter", "32-Bit Scan", and "32-Bit Bouncer".

This simulator is unusual in that it has no memory defined, but instead has several registers defined that act as memory. When reading or writing memory, the address is ignored and the value returned depends on the pattern (variant) selected.

2.3 8080 Family

The 8080 simulator has variants defined for the 8080, 8085, and Z-80 processors. Only the 8080 and 8085 are currently implemented. The Z-80 is for future development. These are 8 bit processors with an 8 bit data bus and a 16 bit address bus. In addition to a memory bus, these processors also include an I/O bus with 8 bit I/O port addressing.

Currently, the 8080 family does not have interrupts enabled or memory mapped I/O. These may be added at some time in the future.

When loading data using the `load` routine, the specified file is assumed to be in Intel Hex format.

2.4 68000 Family

The 68000 simulator has variants defined for the 68000, 68008, 68010, and CPU32. Only the 68000 and 68008 are currently implemented. The 68010 and CPU32 are for future development. Internally, these are 32 bit processors with 32 bit data and 32 bit address busses. The external address and data bus sizes depend on the variant selected.

The interrupt code is interpreted with the low order bits (7-0) representing the vector number and the next 8 bits (15-8) representing the priority. Interrupts with vectors that match internally defined exception vectors are ignored. Thus only interrupt vectors in the range 25-31 and 64-255 are processed.

When loading data using the `load` routine, the specified file is assumed to be in Motorola S-Record format.

Chapter 3

I/O Devices

Each I/O device is based on an object that derives from the `io_device` object defined in the `BBS.Sim_CPU` package. A generic I/O device interface is defined with some procedures or functions that must be defined by a specific I/O device and some that may be defined, if needed. There are also a number of utility functions that are not expected to be overridden by a specific I/O device.

To be used by a simulator, an I/O device must first be attached to the simulator using the `attach_io` routine.

```
--
--  Called to attach an I/O device to a simulator at a
--  specific address.  Bus is simulator dependent as
--  some CPUs have separate I/O and memory space.
--
procedure attach_io(self : in out simulator;
                    io_dev : io_access;
                    base_addr : addr_bus;
                    bus : bus_type) is abstract;
```

Many of the I/O devices will have an initialization routine specific for that device. Since this may have device specific parameters, it can't be defined as part of the `io_device` object.

Along with attaching the I/O device to the simulator, the base address needs to be set in the I/O device object. The value for `base` in this routine should match the value for `base_addr` in the `attach_io` routine. Bad things may happen if they don't match.

```
--
--  Set the I/O device base address
--
procedure setBase(self : in out io_device;
                  base : addr_bus) is abstract;
```

The I/O device `read` and `write` functions are called by the simulator when an address within the devices address range is accessed. Note that the address range starts at the `base_addr` specified in the `attach_io` call and extends through the number of addresses specified in the `getSize` function. If needed, the simulator can query the I/O device for its base address using the `getBase` function. If a device needs to do DMA or interrupt a simulator, the `setOwner` procedure must be called first to give the device a reference to the simulator. For interrupts, the `setException` routine is used to define a `long` value that the device passes to the simulator on exception.

```
--
--  Set which exception to use
--
procedure setException(self : in out io_device;
                       except : long);
```

If a device does not use interrupts, this routine can be declared as `null`.

3.1 Clock

The main function of the clock device is to provide a periodic interrupt. It can be enabled or disabled and the time between interrupts can be set in $\frac{1}{10}$th of a second. The base time can be changed by editing the code. Note that the timing is only approximate and depends on many things. Also since the simulation is not running at the same speed as actual hardware the number of instructions between each interrupt will be different from actual hardware.

3.2 Serial Ports

Serial ports provide a way to provide a terminal interface to a simulator. Unlike real serial ports, output buffering is done by the host operating system and it is not obvious to the I/O device if the output has been completed or not. As a result, currently no status is set or interrupts generated on output. The output is assumed to complete instantly.

3.2.1 Basic Serial Port

The basic serial port was developed in early testing to send output to and get input from the terminal controlling the simulator. It worked well enough to test some initial concepts, but has been superseded by the single line telnet port.

3.2.2 Single Line Telnet Port

The single line telnet port provides a replacement for the basic serial port. This device has an `init` routine that is used to set the TCP port. Using `telnet` to connect to this port connects to the device and will allow communication with software running on the simulator. If enabled, this device can provide interrupts to the simulator when characters are received.

3.2.3 Multi-Line Telnet Port

The multi-line telnet port combines 8 ports into a single simulated device. The `init` routine specifies the starting TCP port. It and the next seven ports can be accessed by `telnet` to communicate with software running on the simulator. This uses a single exception and requires fewer addresses than having 8 single line telnet ports.

3.3 Disk Interfaces

The initial `disk_ctrl` device was designed with 8 inch floppy disks in mind for use by CP/M. It has been extended to allow other disk geometries, but follows the model of tracks, sectors per track, and heads for defining a disk. It currently supports 16 bit DMA, but may be extended to support 32 bit DMA. All data transfers occur between the request and the next instruction processed by the simulator.

A `hd_ctrl` device is under development that will support a simpler model. Disks are simply a linear sequence of blocks. It will support 32 bit block addressing and 32 bit DMA.

Chapter 4

Command Line Interface

The command line interface first requests which simulator to use. The choices are currently 8080 and 68000. This will change as more simulators are added. Depending on which simulator is selected, a number of I/O devices are attached to the simulator. Currently, changing this will require editing and rebuilding the program.

4.1 Commands

A number of commands are provided to allow the user to run and interact with software running on the simulator. The commands and their exact operation may change depending on feedback from usage.

; – A comment. Everything on the line is ignored.

BREAK <addr> – Sets a breakpoint at the specified address.

CONTINUE – Clear a simulation halted condition.

DEP <addr> <byte> – Deposit a byte into memory.

DUMP <addr> – Dump a block of memory starting at the specified address. This can be abbreviated to `D`.

EXIT – Exit the simulation (synonym for `QUIT`).

GO <addr> – Sets the execution address.

INTERRUPT <ON/OFF> – Can be abbreviated to `INT`. Used to enable or disable interrupt processing by the simulated CPU.

LISP – Starts the Tiny-Lisp environment.

LOAD <filename> – Loads memory with the specified file (typically Intel Hex or S-Record format).

QUIT – Exit the simulation (synonym for `EXIT`).

REG – Display the registers and their contents.

RESET – Calls the simulator's `init` routine.

RUN – Runs the simulator. While running, it can be interrupted by a control-E character. This can be abbreviated to `R`.

STEP – Execute a single instruction, if the simulation is not halted.

TRACE <value> – Sets the trace value. The value is interpreted by the simulator and typically causes certain information to be printed while processing.

UNBREAK <addr> – Clears a breakpoint at the specified address. For some simulators, the address is ignored and may not need to be specified. In this case, all or the only breakpoint is cleared.

4.2 Lisp Programming

The Lisp environment is based on Ada-Lisp, available at `https://github.com/BrentSeidel/Ada-Lisp`, with a few commands added to interface with a simulator. Refer to the Ada-Lisp documentation for details about the core language. This section just describes the additional commands.

4.2.1 Go

A number representing the starting address for program execution.

Outputs

None

Description

Sets the starting address for program execution. The next `(sim-step)` command executes the instruction at this location.

4.2.2 Halted

Inputs

An optional boolean that if set true, clears the halted state of the simulator. If set false, does nothing.

Outputs

If no input is provided, it returns a boolean representing the halted state,

Description

This is used to clear to test the halted state of a simulator.

4.2.3 Memb

Inputs

A memory address and an optional byte value.

Outputs

If no byte value is provided, it returns the byte at the memory address.

Description

Reads or writes a byte value in memory.

4.2.4 Meml

Inputs

A memory address and an optional long value.

Outputs

If no long value is provided, it returns the long at the memory address.

Description

Reads or writes a long value in memory.

4.2.5 Memw

Inputs

A memory address and an optional word value.

Outputs

If no word value is provided, it returns the word at the memory address.

Description

Reads or writes a word value in memory.

4.2.6 Num-reg

– (num-reg)

Inputs

None

Outputs

The number of registers defined by the simulator.

Description

Returns the number of registers defined by the simulator. This could be used for iterating through the registers and displaying their values.

4.2.7 Reg-val

Inputs

A register number

Outputs

The value of the specified register, or an error if the register number is not in range.

Description

Returns the value of the specified register.

4.2.8 Sim-init

Inputs

None.

Outputs

None.

Description

Calls the initialization routine to the current simulator. The action is simulator dependent, but typically clears the registers. The memory may, or may not be cleared.

4.2.9 Sim-step

Inputs

None.

Outputs

None.

Description

Executes one instruction or step of the simulator.

4.2.10 Examples

The Lisp environment is useful for writing automated tests and for writing utilities to enhance the CLI.

Watch a Memory Location

The following Lisp program will execute instructions until the specified memory location changes.

```
(defun watch (addr)
  (setq old-value (meml addr))
  (dowhile (= old-value (meml addr))
    (sim-step)))
```