Documentation for CPU Simulators

Brent Seidel
Phoenix, AZ

April 2, 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 nee
--  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/
--  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
```

2.2 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.

2.3 8080 Family

2.4 68000 Family

Chapter 3

I/O Devices

3.1 Clock

3.2 Serial Ports

3.3 Disk Interfaces

Chapter 4

Command Line Interface

4.1 Commands

4.2 Lisp Programming