ARM-Architecture Simulator

Archulator

Contents
Purpose............................................................................................................................................ 2
Description...................................................................................................................................... 2
Block Diagram............................................................................................................................ 2
Component Description .............................................................................................................. 3
  Buses .................................................................................................................................................... 3
  μP .......................................................................................................................................................... 3
  Caches ................................................................................................................................................ 3
  Memory................................................................................................................................................ 3
  CoProcessor ..................................................................................................................................... 3
Using the Archulator ....................................................................................................................... 4
  Internal Directory Structure ........................................................................................................ 4
  Building the project and example files ....................................................................................... 4
  Requirements .................................................................................................................................... 4
  Build Procedure ................................................................................................................................ 4
The libsim Library .......................................................................................................................... 5
Example Application ...................................................................................................................... 6
Creating and testing user applications ............................................................................................ 7
  Software applications .................................................................................................................. 7
  Hardware-Software Partitioned applications .............................................................................. 8
CoProcessor’s SC_THREAD function ............................................................................................. 10
Purpose

The Archulator or the ARM-Architecture Simulator is a simulation framework that provides the means to simulate user programs for an ARM target system consisting of a processor, instruction and data caches, system bus, memory device, and an optional hardware coprocessor.

Description

The Archulator is built by integrating SimpleScalar cache memory simulator (ARM version) within a Transaction Level Modeling (TLM) interface. Different system components like the processor core, the instruction and data caches, the system bus, the memory device, and the hardware coprocessor are all modeled using SystemC. Some of the features of Archulator are as mentioned below:

- Provides details like the number of instructions executed by the processor, and the number of load/store operations initiated by the processor.
- Comprises actual set-associative and user-configurable instruction and data caches.
- Provides detailed cache statistics that include the number of cache hits, misses, and replacements.
- Includes an optional and easily customizable hardware coprocessor that is particularly useful to simulate hardware-software partitioned applications.

Block Diagram

Figure 1 shows the various internal system components that make up the Archulator.

![Archulator Block Diagram](image-url)
Component Description

Buses
I-Bus and D-Bus are the SystemC channels which model the instruction and data cache buses respectively. M-Bus is a SystemC channel that models the system bus for performing memory operations on the memory device. All the three buses implement common bus-master and bus-slave interfaces. S-Bus is a special SystemC channel that facilitates the processor to execute system calls. System Calls are requests made by the application program and these operations require operating system’s intervention. The requests may include file I/O, console I/O or other operations. The S-Bus implements the S-Bus master interface and has a port for accessing the D-Bus.

µP
µP represents the SystemC module that wraps the ARM-processor core which is responsible for Fetch-Decode-Execute operations. µP also implements a Hardware-Interrupt interface using which a hardware coprocessor can raise an interrupt on the processor. µP also uses ports to connect to the I-Bus, S-Bus, and D-Bus for the execution of instructions, system calls, and load/store operations respectively. In other words, the µP acts as the bus master to the three buses.

Caches
I-Cache and D-Cache are SystemC modules that model the instruction and data caches respectively. Both the instruction and data caches are set associative caches which have write-through cache policy. The associativity, number of sets, block size, and replacement policy of each cache can be configured by the user as command-line arguments. I-Cache and D-Cache act as bus-slaves to the I-Bus and D-Bus respectively, while they both act as bus-master to the M-Bus.

Memory
Memory represents the SystemC module that models the memory device. Memory has the complete ownership of the entire simulator memory i.e. the memory required to simulate the application program’s stack, heap, data, and text segment. Memory acts as a bus-slave to the M-Bus.

CoProcessor
CoProcessor represents the SystemC module that models the hardware coprocessor. During normal execution, the CoProcessor functions as a bus slave to the data cache bus, listening to all the read/write events over the D-Bus. When µP performs a write operation on a special memory location, the CoProcessor starts functioning as the bus master to the D-Bus. When it is done with executing all its instructions, the CoProcessor relinquishes control by raising an interrupt (in the case of SLEEP mode) on the µP using the µP’s hardware interrupt interface and starts functioning as a slave device connected to the D-Bus.
Using the Archulator

Internal Directory Structure

The internal directory structure of Archulator is as follows:

- **ss**
  - This folder contains the simulator project files that need to be built. After a successful build, the *sim-cache* executable is generated.

- **libs**
  - This folder contains the header and library files necessary to create a hardware-software partitioned application.

- **example**
  - This folder contains two versions of an example application – a pure software version and a hardware-software partitioned version.

Building the project and example files

Requirements

1. archulator.tar.gz
2. g++ compiler to build the simulator project files

The following are the additional requirements if you are trying to build on your own LINUX/UNIX machine.

3. arm-linux-gcc (or arm-linux-g++) compiler tool chain to build user applications including the example applications
4. SystemC environment.

Build Procedure

1. Untar and expand archulator.tar.gz into a single folder, for e.g. `~/ece576/archulator`. Now the *archulator* folder should contain *ss*, *libs*, and *example* sub-folders.
2. If you are building on your own UNIX/LINUX machine, add the location of your arm-linux compiler to the PATH variable.
3. If you are trying to build the files on the Embedded ECE server, you just need to run the following command
   
   ```bash
   ~/ece576/archulator$ ./setup OR
   ~/ece576/archulator$ source ./setup
   ```
4. Run the following command to build both the simulator and example applications.
   ```bash
   ~/ece576/archulator$ make
   ```
   Alternately, to build only the simulator, run the following command
   ```bash
   ~/ece576/archulator$ make simulator
   ```
   If you want to build only the example applications, you can run the following command
   ```bash
   ~/ece576/archulator$ make examples
   ```
Note: If you are running the above commands on your own LINUX/UNIX machine, you have to add the location of the SystemC’s include and library path inside the Makefile of the simulator. For example, if you have set up SystemC under the home/ece576/SystemC folder, make the following changes to the Makefile present in home/ece576/archulator/ss folder
SYSINCS = -I /home/ece576/SystemC/systemc-2.2.0/include
SYSLIBS = -L/home/ece576/SystemC/systemc-2.2.0/lib-linux –lsystemc

The libsim Library

The libsim library is provided to facilitate the creation of hardware-software partitioned applications. The Application Programmer Interfaces exported in this library can be used to coordinate the execution of the microprocessor and the hardware coprocessor. The microprocessor and the hardware coprocessor can execute either in parallel or in a mutually exclusive manner. The table below describes the APIs exported in the libsim library.

<table>
<thead>
<tr>
<th>API</th>
<th>API Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get_HwRegs_Count():unsigned int</td>
<td>Returns the number of memory-mapped hardware registers available with the hardware coprocessor.</td>
</tr>
<tr>
<td>Get_HwReg_Data(...):int</td>
<td>Returns the data stored in the indexed hardware register. The index value can vary from 0 to 1 less than the value returned by the Get_HwRegs_Count() API. 0xffffffff is returned if the index range is violated.</td>
</tr>
<tr>
<td>Set_HwReg_Data(...):int</td>
<td>Sets the data in the indexed hardware register. The index value can vary from 0 to 1 less than the value returned by the Get_HwRegs_Count() API. The API also returns the data value set. 0xffffffff is returned if the index range is violated.</td>
</tr>
<tr>
<td>Sleep()</td>
<td>Puts the microprocessor into sleep mode. The microprocessor wakes up only after an interrupt that is raised by the hardware coprocessor</td>
</tr>
<tr>
<td>CoExecute()</td>
<td>Puts the microprocessor into co-execution mode. In this mode, the coprocessor and microprocessor can execute in parallel.</td>
</tr>
</tbody>
</table>
Example Application

The example application can be found in the `~/ece576/archulator/example` folder. The example application executes one of three sorting operations (bubble sort, insertion sort, and selection sort) to sort a set of 512 integers present in `input.dat` file. The user can specify his/her choice of sorting operation as a command line argument. There are two versions of this application:

1. **sort_orig** – The software version which is entirely simulated on the microprocessor. To simulate this application on Archulator, first build the application as described in the earlier section, and follow the steps below:
   a. `$ cd ~/ece576/archulator/example/sort_orig`
   b. `~/ece5/archulator/example/sort_orig$ ../../ss/sim-cache sort_orig -<option>`
   where option is `b` for bubble sort,
      i for insertion sort,
      s for selection sort,

   Check the contents of the `output.dat` file generated to see if the integers are sorted in ascending order. Also, observe the important statistics thrown by the simulator, such as the number of instructions executed by the microprocessor, the number of load/store operations initiated by the microprocessor and the number of data cache accesses. Repeat the steps for all the three sorting algorithms.

2. **sort_hwsw** – The hardware-software partitioned version which is executed by the microprocessor and the hardware coprocessor. This version is generated by partitioning the sorting loops to the coprocessor. As an example, consider the BubbleSort() function from `sort_hwsw.c` file. Compare this function with the BubbleSort() function in `sort_orig.c` file and notice in Figure 2 how the sorting loop is partitioned to the hardware coprocessor.

   ```
   int* data;
   int data_length;

   BubbleSort()
   {
      /*
         Read from the input file and populate the data items read in the data pointer.
         data_length contains the length of the data items (integers) read.
      */

      Set_HwReg_Data( 0, (unsigned int) data_length ); // Store the length of the data array in HW REG 0
      Set_HwReg_Data( 1, (unsigned int) data ); // Store the source address (address of the starting array element) in HW REG 1
      Set_HwReg_Data( 2, (unsigned int) `b` ); // HW REG 2 is for opcode. ‘b’ for bubble sort, ‘i’ for insertion sort, and ‘s’ for selection sort
      Sleep(); // Microprocessor enters the sleep mode. The Coprocessor takes over the execution.
   }
   ```

   Figure 2: A BubbleSort() function where the loops are partitioned to hardware

   To simulate this application on Archulator, build the application as explained in the earlier section and follow the steps below:
a. `~$ cd ~/ece576/archulator/example/sort_hwsw`

b. `~/ece576/archulator/example/sort_hwsw$ ..../ss/sim-cache sort_hwsw -<option>`, where option is:
   - `b` for bubble sort,
   - `i` for insertion sort,
   - `s` for selection sort.

Check the contents of the `output.dat` file generated to see if the integers are sorted in ascending order. Also, observe and compare the important statistics with those obtained with `sort_orig` application.

**Note:** You can check the README file inside the example folder for more information on how to run the example applications.

### Creating and testing user applications

#### Software applications

Such applications are simulated solely on the microprocessor. To build the C/C++ source files for such applications, use the `Makefile` present in the `~/ece576/archulator/example/sort_orig` folder as a template. Just make sure that for a C-application, the compiler is `arm-linux-gcc` while for a C++-application, the compiler is `arm-linux-g++`. For software applications, there is no requirement for a hardware coprocessor.

If a binary called `test` is generated in the `~/ece576/archulator/example/testapp` folder, the application can be simulated by following the steps below: -

- `~$ cd ~/ece576/archulator/example/testapp`
- `~/ece576/archulator/example/testapp$ ..../ss/sim-cache <optional cache configurations> test <argv1 argv2 ...>`

Cache configuration can be set for both instruction and data caches as shown below:

- `-cache:dl2 <name>:<nsets>:<bsize>:<assoc>:<repl>` (for data cache)
- `-cache:il2 <name>:<nsets>:<bsize>:<assoc>:<repl>` (for instruction cache)

Or, it is possible to specify a unified instruction and data cache as shown below:

- `-cache:dl2 <name>:<nsets>:<bsize>:<assoc>:<repl>` (for data cache)
- `-cache:il2 dl2`

where,
- `name` - the name of the cache being defined
- `nsets` - number of sets in the cache
- `bsize` - block size of the cache
- `assoc` - associativity of the cache
Example:
-cache:dl2 dl2:4096:32:1:1
-cache:il2 il2:1024:64:2:1

or a unified cache,
-cache:dl2 ul2:1024:64:2:1
-cache:il2 dl2

Hardware-Software Partitioned applications

Building hardware-software partitioned applications is similar to building software applications except that the Makefile present in the ~/ece576/archulator/example/sort_hwsw folder has to be used as a template. Such applications need to link to the libsim library (present in ~/ece576/archulator/libsim/lib folder), and use the exported APIs listed in the hwswsim.h header file (present in ~/ece576/archulator/libsim/include folder).

1. To know how to partition the critical loop/s to the hardware coprocessor, refer to the sort_hwsw application’s source file.
2. In general, all the information required for the coprocessor to execute the loop is written into one or more memory-mapped hardware registers of the coprocessor.
3. Then the Sleep()/CoExecute() API is called. This means that the coprocessor has to be customized for every user application. This can be done by making alterations to the coprocessor.h and coprocessor.cpp files present in the ~/ece576/archulator/ss folder. After making the following changes, make sure you do a clean build on the simulator.
   
   [1] Changes to coprocessor.h
   Declare the application specific methods as public member functions of the CoProcessor class.

   [2] Changes to coprocessor.cpp
   1. Provide the definition for the application specific member functions of the CoProcessor class
   2. Call these application specific member functions when the hardware coprocessor becomes the bus master. This is indicated by the CoProcessor::isCoprocessorMode flag as shown in Figure 5.

   ```
   void coprocessor::coproc_thread_function()
   {
       while(true)
       {
           if( isCoprocessorMode )
           {
               
               /**
               Coprocessor is now bus master.
               Call the application specific member functions based on program conditions
               */
               ...
           }
       }
   }
   ```

   Figure 3: Changes to Coprocessor’s thread function.
4. The Coprocessor is connected to the D-Bus in order to access the simulator memory i.e. the memory device via the D-cache. The member variable (of `CoProcessor` class) `dcache_port` can be used to read/write data to the memory device over the D-Bus using the method `dcache_port->Access(...)`. The signature of `Access(..)` method is shown in Figure 4. Note that none of the instructions executed by the Coprocessor is profiled by the Archulator. The `dcache_port->Access(..)` calls are reflected in the load store operations, and the D-cache statistics.

```c
/**
 * Method to perform Read/write operation.
 * The Bus Master uses this method to perform read/write operation on the
 * memory mapped slave devices over the bus.
 * @param cmd read/write command. Can have values Read or Write.
 * @param addr address where the read/write operation needs to be performed.
 * @param vp data pointer casted as void*, pointing to the actual data meant
 * for read/write operation
 * @param nbytes number of bytes of data meant for read/write operation.
 * @ret @li md_fault_none if no error. Otherwise, refer md_fault_type
 * enumerators in machine.h
 */
virtual enum md_fault_type
Access( enum mem_cmd cmd,
       md_addr_t addr,
       void* & vp,
       int nbytes ) = 0;
```

Figure 4: Signature of D-Bus access method
CoProcessor’s SC_THREAD function

The CoProcessor’s SC_THREAD function is shown in Figure 5.

```c
void CoProcessor::coproc_thread_function()
{
    enum mem_cmd cmd;
    md_addr_t addr;
    word_t* word_ptr = NULL;
    void *vp;
    vp = (void*) word_ptr;
    int nbytes;

    while( true )
    {
        if( isCoprocessorMode ) {
            wait(SC_ZERO_TIME);
            /* Coprocessor mode reached */
            /* Add Application Specific code here */

            wait(SC_ZERO_TIME);
            isCoprocessorMode = false;
            mp_port->Hw_Hinterrupt();
        }

        /* Coprocessor functions as a slave device, listening to events over the bus */
        slave_port->WaitOnEvent();
        slave_port->ServiceEvent( cmd, addr, vp, nbytes );

        /* Coprocessor's registers are actually memory mapped locations */
        if(( sizeof(word_t) == nbytes ) ) {
            /* General purpose registers of Coprocessor used by uP to store information */
            if( (addr >= HW_REG_START_ADDR) &
                (addr < (HW_REG_START_ADDR + HW_REGS_COUNT * sizeof(word_t)) ) ) {
                int index = ((addr - HW_REG_START_ADDR) / 4);
                if( cmd == Write )
                    _REG[index] = *( (unsigned int*) (vp) );
                else
                    *( (unsigned int*) (vp) ) = _REG[index];
            }
        } /* Special memory mapped locations of Coprocessor.
        1. If something is written into HW_SLEEP_REG (i.e Sleep() call ),
            Coprocessor enters sleep mode
        2. If something is written into HW_COEXEC_REG (i.e. CoExecute() call),
            Coprocessor enters co-execution mode */
        else if ( (addr == HW_SLEEP_REG) || (addr == HW_COEXEC_REG) )
            isCoprocessorMode = true;
    }
}
```

Figure 5: CoProcessor’s SC_THREAD function
a. It can be seen that CoProcessor::isCoprocessorMode decides the mode of operation of CoProcessor.

b. By default, CoProcessor keeps listening to the bus events on the D-Bus.
   I. If the Microprocessor tries to read/write to the memory locations of the hardware registers, the CoProcessor responds like a memory device as shown from line 24 to 37. Therefore, before the CoProcessor becomes the master of D-Bus, the Microprocessor can write all the useful data required by the CoProcessor (like loop counters, source and destination operand addresses, special op-codes and etc) to the latter’s hardware registers like normal load/store operations.
   II. If the Microprocessor performs the memory access operation on one of the two special memory locations i.e. either HW_SLEEP_REG or HW_COEXEC_REG, the CoProcessor sets the CoProcessor::isCoProcessor to true indicating that it is entering the coprocessor mode, thus becoming the master of D-Bus as shown in lines 45 and 46. The Microprocessor performs this memory access operation in response to the API call Sleep() or CoExecute().

c. On entering the coprocessor mode (line 12), the CoProcessor can execute its own code and also access data on the memory device via the D-cache over the D-Bus. That is, application specific member function/s of the CoProcessor class should be called after line 13.

d. After the CoProcessor has finished executing its code i.e. after the application specific member functions have returned, the CoProcessor::isCoprocessor flag is set to false (line 19) indicating that it is entering the slave device mode.

e. Then a hardware interrupt is raised on the Microprocessor indicating that the CoProcessor is relinquishing its rights as the master to the D-Bus (line 20). This hardware interrupt will have no effect if the Microprocessor was executing in co-execution mode (CoExecute() API call). If the Microprocessor was in sleep mode (Sleep() API call), the hardware interrupt will wake up the Microprocessor so that it becomes the master of the D-Bus again.