Abstract

This document describes the classes in version 2.4a of the computer architecture library, their functions, the relationships between the classes, and provides some simple examples of how to use the library. The library is designed to simulate the different hardware components used in modern architectures and their interactions. This architecture library is meant to be used with the Computer Architecture Course at RIT.
1. Fundamental Structure

The “arch” library is a C++ class hierarchy (see Figure 1) which provides components for use in writing CPU simulators. The library contains classes which describe the major components of typical CPUs, including memory, registers, buses, and an ALU.

2. Class Summary

The following is a brief description of some of the classes available within the library. A more detailed description is provided later in the document.

2.1. Class Hierarchy

At the root of the hierarchy is the CPUObject class. This abstract class provides attributes and methods which are common to most objects in the program, including the object’s ID (used internally and in trace output to identify the object).

Two classes exist outside the hierarchy: Clock and ArchLibError. Clock is never instantiated; all of its data members and methods are static, and exist throughout the lifetime of the program. (See below for a more complete description of Clock.)

ArchLibError is used by the library as an exception class. Many library components will throw ArchLibError exceptions when fatal runtime errors are encountered. ArchLibError is derived from the C++ standard runtime_error class; when thrown, it contains a string which can be printed with the what() method.

The CPUObject class will throw an ArchLibError exception if an instance is created after the first Clock::tick() is issued, an attempt to create too large a CPUObject
occurs, or if a CPUObject is passed as a non-reference parameter to a function.

Below CPUObject are the three main branches in the library: Flow, ClockedObject, and Connector.

Flow
A “placeholder” superclass for the InFlow and OutFlow classes (see below).

ClockedObject
ClockedObjects are components which are inherently sequential in nature - that is, they operate synchronously, based on a common clock. All ClockedObjects will operate at the same time, under the control of the Clock.

Connector
Provides the communication mechanism used by other classes to transfer information. (See InFlow and OutFlow, below.)

2.2. Classes Not Commonly Used
Several of the classes in the hierarchy are typically never instantiated by programmers using the library, but rather exist to provide services to the other classes in the library.

InFlow
An object which accepts input from another library component (e.g., a StorageObject). An InFlow can only be connected to one source at a time during any given clock cycle, but can be connected to any number of sources through the life of the program. This is the “input side” of a connection. An InFlow is connected to a source by invoking the InFlow's pullFrom(StorageObject &src) method. If an attempt to use an InFlow is made but the pullFrom() method is not invoked, an ArchLibError exception will be thrown.

OutFlow
An object which provides output to another library component (e.g., a StorageObject). An OutFlow can be connected to any number of destinations at one time. This is the “output side” of a connection. An OutFlow is connected to a destination by invoking the destination’s latchFrom(OutFlow &dst) method (see StorageObject, below).

3. Descriptions of Commonly-Used Classes
In this section, we will discuss the data types and most important methods available in library classes used in most simulation programs.

3.1. Bus
A Bus is the standard Connector, containing one InFlow and one OutFlow. It just passes data straight through. This is useful when an architecture requires passing data between different simulated pieces of hardware.
3.1.1. Commonly-Used Methods

Bus( const char *id, int numbits)

The constructor for the standard Bus. The arguments are its ID for referencing an instantiation and the capacity, in bits, of the Bus.

InFlow &IN()

Returns a reference to the Bus’s InFlow object. If an object’s OutFlow is connected to the Bus’s InFlow when a clock tick occurs, that object’s contents will be moved through the Bus.

OutFlow &OUT()

Returns a reference to the Bus’s OutFlow object. If another object’s InFlow is connected to the Bus’s OutFlow when a clock tick occurs, that object will take on the data being moved through the Bus.

3.2. BusALU

The BusALU is a special kind of Bus that can do arithmetic and logical operations as well as passing data to and from other objects. Unlike a normal Bus, it has two InFlows, and it provides three separate OutFlows. The InFlows (accessed with the OP1() and OP2() functions, below) provide operand 1 and operand 2 inputs for the operation; the OutFlows (OUT(), CARRY(), and OFLOW()) provide the result and status information.

3.2.1. Defined Types

enum Operation

This type defines the set of operations which can be performed by a BusALU. These values are passed as parameters to the perform() function (described below).

    op_none
    
    Do nothing.

    op_add, op_sub

    Compute OP1 + OP2, or OP1 – OP2.

    op_and, op_or, op_xor

    Bitwise AND, OR, or XOR of OP1 & OP2.

    op_not

    Bitwise complement of OP1.

    op_extendSign

    Extend the sign bit of OP1; OP2 is a mask which is non-zero in the position of the sign bit.

    op_lshift, op_rshift

    Left or right shift OP1; OP2 is the shift count.

    op_rashift

    Arithmetic (signed) right shift OP1; OP2 is the shift count.
op_rop1, op_rop2
   Just pass OP1 or OP2 through.

op_zero, op_one
   Output the constant 0 or the constant 1.

3.2.2. Important Methods

BusALU( const char *id, int numbits)
   The constructor for the arithmetic Bus. The arguments are its ID and the capacity
   (in bits) of the Bus.

void perform( BusALU::Operation op )
   Perform the specified operation using OP1 and OP2 on the next clock pulse. Note
   that OP1 and OP2 must be connected to a source before the clock pulse occurs
   using pullFrom().

InFlow &OP1()
   Returns a reference to the OP1 InFlow. Used when setting up source connections.

InFlow &IN1()
   Returns a reference to the OP1 InFlow. Used when setting up source connections.
   Included for consistency with Bus naming convention.

InFlow &OP2()
   Returns a reference to the OP2 InFlow. Used when setting up source connections.

InFlow &IN2()
   Returns a reference to the OP2 InFlow. Used when setting up source connections.
   Included for consistency with Bus naming convention.

OutFlow &OUT()
   Returns a reference to the OUT OutFlow. Used when transferring the result of
   the operation to another object.

OutFlow &CARRY()
   Boolean path that tells whether there was a carry resulting from the operation
   performed in the previous clock cycle.

OutFlow &OFLow()
   Boolean path that tells whether there was a signed overflow resulting from the
   operation in the previous clock cycle.

3.2.3. Exceptions

BusALU will throw an ArchLibError exception if it is given an illegal BusALU::Operation
   to perform, or if an attempt is made to use its result without telling it to perform
   an operation.
3.3. Clearable

A StorageObject with additional methods allowing it to clear and set its contents.

3.3.1. Defined Types

define Operation
    Used in calls to perform() (see below) to indicate which operation should be performed.

    none
        Do nothing.
    clearOp
        Clear the Clearable (assigns a value of 0 to each bit).
    setOp
        Set the Clearable (assigns a value of 1 to each bit).

3.3.2. Important Methods

Clearable( const char *id, int numBits, long value=0 )
    The constructor for the Clearable object. The arguments are its ID, its capacity (in bits), and its initial contents (by default, 0).

void perform( Clearable::Operation op )
    Perform the specified operation on the next clock pulse.

void set()
    This is a shorthand function; it is equivalent to perform(setOp).

void clear()
    This is a shorthand function; it is equivalent to perform(clearOp).

3.3.3. Exceptions

Clearable will throw an ArchLibError exception if it is given an illegal operation to perform.

3.4. Clock

The "system clock" of the simulation, controlling and counting clock ticks. It regulates the actions of all StorageObjects. Clock is never instantiated; its member functions are all declared static, and are accessed globally.

3.4.1. Important Methods

static void tick()
    Increments the system clock, and sends the clock pulse to each synchronous component in the program. This should be called when everything has been set up for the next clock cycle.
static long getTime()
    Returns the number of clock cycles since the beginning of the simulation – i.e.,
    the number of times tick() has been called.

3.5. Counter
A StorageObject with the ability to clear its contents, or to increment or decrement
its contents by 1, 2, 4, or 8.

3.5.1. Defined Types

enum Operation
    These values are used in the perform() function.
    clear0
        Clear the contents.
    decr8, decr4, decr2, decr1
        Subtract 8, 4, 2, or 1 from the current contents.
    none
        Do nothing.
    incr1, incr2, incr4, incr8
        Add 1, 2, 4, or 8 to the current contents.

3.5.2. Important Methods

Counter( const char *id, int numBits, long value=0 )
    The constructor for the Counter object. The arguments are its ID for refer-
   encing an instantiation, the capacity (in bits) of the Counter object, and its initial
    contents (by default, 0).

void perform( Counter::Operation op )
    Perform the specified operation.

void incr()
    Shorthand for perform(incr1); adds 1 to the current contents.

void decr()
    Shorthand for perform(decr1); subtracts 1 from the current contents.

void clear()
    Shorthand for perform(clear0); clears the current contents.

int overflow()
    Check to see if a roll-over occurred (FFF..FF --> 000..0). This should be used
    after executing a clock tick.
3.6. Constant

A Connector that simply provides the same contents to its OutFlow every time. It is not connected to any InFlow. Equivalent to a Bus which has its InFlow permanently connected to a source for this value. Note that you cannot connect a Constant to either of the inputs of an instance of BusALU.

3.6.1. Important Methods

Constant( const char *id, int numBits, long value=0 )

The constructor for the the Constant connector. The arguments are its ID, the capacity (in bits) of the Constant object, and its permanent contents (by default, 0).

void printOn( ostream &o ) const

The print interface: prints its register name (id) and contents to the specified stream.

3.7. Memory

The Memory object is a collection of memory cells, arranged as an array, and individually addressable. It provides access to the memory cells through an internal StorageObject which functions as a Memory Address Register (MAR), a WRITE InFlow, and a READ OutFlow. Multi-unit transfer and detection of improper addresses are supported.

3.7.1. Defined Types

enum Operation

These values are used in the perform() function.

none

Do nothing.

loadOp

Read in an object file.

readOp

Read data in memory from the current address.

writeOp

Save data to memory at the current address.

3.7.2. Important Methods

Memory( char* id, int bitsInAddr, int bitsPerUnit, unsigned long maxAddr=(unsigned long)(-1), int unitsInDataPath=8 )

The constructor for the the Memory module. The arguments are:

char* id

The name of the module.
int bitsInAddr
    Width of an address; this Memory object will contain \(2^{bitsInAddr}\) units.

int bitsPerUnit
    Number of bits in each addressable unit.

unsigned long maxAddr=(unsigned long)(-1)
    The maximum valid address. The default is -1, meaning all addresses are valid.

int unitsInDataPath=1
    The number of units transferred each time a read or write occurs.

bool littleEndian=false
    Is this Memory object big-endian or little-endian. The default is big-endian.

StorageObject &MAR()
    Returns a reference to the memory unit’s memory address register.

InFlow &WRITE()
    Returns a reference to the memory unit’s ingoing data path for writing to the location specified by the MAR contents.

OutFlow &READ()
    Returns a reference to the memory unit’s outgoing data path for reading from the location specified by the MAR contents.

void perform( Memory::Operation op )
    Perform the specified operation on the next clock tick.

void read()
    Shorthand for perform(readOp).

void write
    Shorthand for perform(writeOp).

void load( char *fileName, long defaultValue=0 )
    Read in an object file. The filename parameter is the pathname of the object file; it is opened and its contents loaded into the Memory object. Each line in the object file except the last has the format addr N word1 ... wordN; each item on the line is a hex number. The final line in the object file contains a single hex number which is the entry point for the object program; it is made available immediately on the READ() OutFlow so that it may be used to initialize the program counter. The rest of Memory is initialized to the defaultValue.

void dump( unsigned long startAddr, unsigned long endAddr, ostream &o=cout )
    Diagnostic memory dump for debugging. Output the contents of memory from startAddr through endAddr to the provided ostream.
void dumpLite( unsigned long startAddr, unsigned long endAddr, ostream &o=cout )

Like dump(), but does not print the addresses, or any newline characters.

int badAddress()

Reflects the status of the most recent read() or write() operation. Can be called after every such operation to allow detection of invalid memory addresses.

3.7.3. Exceptions

Memory will throw an ArchLibError exception if an error occurs during its creation, or during its use (e.g., load() cannot open the specified object file).

3.8. PseudoInput

A StorageObject that gets its contents from stdin every time it is connected to an InFlow.

3.8.1. Functions

PseudoInput( const char* id, int numbits )

The constructor for the the PseudoInput object. The arguments are its ID and the size of each word to be passed through the object.

3.8.2. Exceptions

PseudoInput will throw an ArchLibError exception if an attempt is made to assign to it, or if an input error occurs.

3.9. PseudoOutput

A StorageObject that prints, on stdout, every new value given to it.

3.9.1. Important Methods

PseudoOutput( const char* id, int numbits )

The constructor for the the PseudoOutput object. The arguments are its ID and the size of each word to be passed through the object.

3.10. ShiftRegister

A StorageObject with shifting capability.

3.10.1. Defined Types

enum Operation

These values are used in the perform() function.

none=0

Do nothing.
right
Shift right one position, filling with a zero.

rightArith
Shift right one position, filling with a copy of the leftmost bit.

left
Shift left one position, filling with a zero.

### 3.10.2. Important Methods

`ShiftRegister( const char* id, int numbits, long initVal=0 )`

The constructor for the `ShiftRegister` object. The arguments are its ID, the size of each word to be passed through the object, and its initial contents (by default, 0).

`void perform( ShiftRegister::Operation op )`

Perform the specified operation.

`void rightShift()`

Shorthand for `perform(right)`.

`void rightArithShift()`

Shorthand for `perform(rightArith)`.

`void leftShift()`

Shorthand for `perform(left)`.

`void rightShiftInputIs( StorageObject& obj )`

Use the contents of this `StorageObject` as the value to shift in on the left end.

`void leftShiftInputIs( StorageObject& obj )`

Use the contents of this `StorageObject` as the value to shift in on the right end.

### 3.10.3. Exceptions

`ShiftRegister` will throw an `ArchLibError` exception if it is given inputs for both shift directions at once, or if it is given an input for an arithmetic shift.

### 3.11. StorageObject

The base class of all components of the simulated CPU capable of receiving and holding binary data. It contains only elementary operations allowing it to interact with other components like data paths and the system clock.

#### 3.11.1. Important Methods

`StorageObject( const char* id, int numbits, long initVal=0 )`

The constructor for the `StorageObject` object. The arguments are its ID, the size of the object, and its initial contents (by default, 0). It creates a `numbBits`-wide register.
virtual long value() const
    Return the current contents of the StorageObject.

unsigned long uvalue() const
    Read the current contents as an unsigned number.

int zero() const
    Boolean check to see if the contents equal zero.

void connectsTo( Flow &f )
    Establish bus-register connection. Must be done prior to the first clock pulse
    of the simulation. It establishes the virtual wire between objects; without this,
    objects cannot pass data to and from each other.

void latchFrom( OutFlow &o )
    Establish the source of data for this object on the next clock cycle. The OutFlow
    must have been specified in a call to this StorageObject’s connectsTo() method
    prior to the first clock pulse of the simulation.

long operator()(int startBit, int stopBit) const
    Extract a substring from the startBit to the stopBit. The most significant bit is
    numBits-1; the least significant bit is 0.

long operator()(int bit) const
    Returns the value of the given bit.

long operator()() const
    Returns the value of the most significant bit.

void backDoor( long x )
    Force the StorageObject’s contents to x, without following the normal data flow
    protocol (i.e., asynchronously, and without any source object). Use of backDoor()
    is recorded and reported at the end of the simulation. Provided for use only as a
    last resort.

3.11.2. Exceptions

StorageObject will throw an ArchLibError exception if it is told to latchFrom()
    something it is not connected to, if a late connectsTo() call is made, if it is used
    in a pullFrom() operation for something it is not connected to, or if an invalid
    bit range is used with the ( ) operators.

4. Examples

4.1. Miscellaneous Details

4.1.1. Programming Using the Library

Every library component is described by a C++ header file containing its class decla-
ration. To use a particular component, you must include its header file; e.g., to use a
Memory object:

```cpp
#include <Memory.h>
```

Typically, header files for class X have the name X.h. Note that any program which uses any of the synchronous components must include the Clock header file, even though Clock is never instantiated.

### 4.1.2. Debugging

The arch library contains a facility which can be used to aid debugging efforts. The CPUObject class contains a static data member named debug which is examined by every descendent of CPUObject on every clock tick. This data member is public, and can be set by the programmer to enable three types of debugging output.

CPUObject defines a data type named DebugMode which contains three values:

- **create**
  - Enable tracing of object creation. Each CPUObject descendent will announce its existence at the time of creation.

- **trace**
  - Enable data flow tracing. During execution, every clock pulse will cause each CPUObject descendent being manipulated in that clock pulse to print trace output showing the movement of data between components.

- **memload**
  - Causes the Memory object to print out the contents of each unit which was initialized from the specified object file.

The values of these are assigned in such a way that any or all of them can be enabled at a time:

```cpp
CPUObject::debug |= CPUObject::trace;
```

This turns on the trace flag by ORing that bit into the debug variable. To enable more than one, OR them together, as in these two examples which turn on both the trace and the memload flags:

```cpp
CPUObject::debug |= CPUObject::trace;
CPUObject::debug |= CPUObject::memload;
CPUObject::debug |= CPUObject::trace | CPUObject::memload;
```

Note that it is suggested that you use the |= operator rather than just the = operator to set these bits, to preserve the state of the other bits in debug.
4.1.3. Compilation

When compiling your simulator, you must make sure that the C++ compiler is able to locate the header files for the library modules you’re using. Under UNIX, this is typically accomplished with the -I command-line option given to the C++ compiler:

```
CC -c -IPathToHeaderFileDirectory prog.C
```

Next, the linker must be told where to find the precompiled version of the library. Again, under UNIX, this is typically accomplished with the -L and -l options:

```
CC prog.o -LPathToLibraryDirectory -lLibraryFileName
```

4.2. A Simple Example: Data Transfer

In this example, we will illustrate the use of a Bus to transfer data between two StorageObjects. Note that this will work for other variants of StorageObject, too.

First, we must include the appropriate header files. We’ll be creating a Bus and two StorageObjects; because they are synchronous, we must also include the Clock header file. In addition, we’ll want to do output, so we must include the iostream header:

```cpp
#include <iostream>
#include <Clock.h>
#include <Bus.h>
#include <StorageObject.h>

using namespace std;
```

Next, we declare the components:

```cpp
Bus b( "Bus", 8 );
StorageObject r1( "Register 1", 8, 5 );
StorageObject r2( "Register 2", 8 );
```

Note that while we now have three architectural components, we haven’t yet connected them; that must be done at runtime, with calls to the connectsTo() function of each StorageObject.

For simplicity, in this example we’ll put all our executable code into our main() function. (In practice, we would normally keep main() as simple as possible, and have it invoke other functions in which the actual simulation would be performed.)

We will have main() enable runtime tracing, make the connections, print the components out, move the contents of r1 into r2, print the components again, and exit. Here is our program:

```cpp
int main( ) {

    CPUObject::debug |= CPUObject::trace;

    try {

        
```

14
Let's analyze each group of statements.

CPUObject::debug |= CPUObject::trace;

Sets the trace flag in the debug variable; every clock pulse will result in output tracing the flow of data.

try {
...
}
catch( ArchLibError ) {
...
}

We use a try block to ensure that we regain control if an ArchLibError exception is thrown.

r1.connectsTo( b.IN() );
r2.connectsTo( b.OUT() );

Set up connections between the components. These connections allow us to move data from r1 through the Bus (b) and into r2. Note that these connections don't let us move data the other way; for that, we would need to also connect r1 to b.OUT(), and r2 to b.IN().

cout << "Registers: " << r1 << ", " << r2 << endl;
Produce a line of output showing the current contents of the two registers.

```c
b.IN().pullFrom( r1 );
r2.latchFrom( b.OUT() );
```

Enable the data transfer from r1 to r2. We must enable the connections we will use every time we want to manipulate data.

```c
Clock::tick();
```

Issue a clock pulse. All the connections and actions which have accumulated since the last clock pulse will occur now, after which these connections are broken – if we want to do them again, we must re-enable them.

```c
cout << "Registers: " << r1 << "", " << r2 << endl;
```

Again, print the contents of the registers.

```c
return 0;
```

Return from main(), which ends the simulation.

Next, we must compile and link our program, which we have saved in a file named prog.C. Assuming we are using version 2-5a of the library on a DCS Sun workstation, we would use these commands:

```c
CC -I/home/course/vcsg720/include/arch2-5a -c prog.C
CC -L/home/course/vcsg720/lib.solaris -o prog prog.o -larch2-5a
```

This will compile prog.C into prog.o, and will then link prog.o against version 2-5a of the library to produce an executable named prog. We run this executable to produce the following output:

```
CPU "ARCH" Simulator, 2.5a(Mar 10 2008)
-----------------------------------------
Registers: Register 1[5], Register 2[0]
-------
-----/0000000 \-----
Register 1[5]-->Bus-->5-->Register 2
Registers: Register 1[5], Register 2[5]

Simulated time 1 cycle
```

LAST CPUObject DESTROYED; END OF SIMULATION