Writing ARMSim# Plugins– A Short Introduction

© W.D. Lyons, 2007
Department of Computer Science
University of Victoria

1. Overview

An ARMSim# Plugin is a .NET assembly that exposes an ARMPlugin Interface. At startup, the ARM-Sim# simulator scans the application directory for assemblies that expose this interface and loads them into the simulator domain. These plugins are then available to extend the simulator functionality.

This document details what functionality can be extended by a plugin and the steps necessary to create one of these plugins. A detailed walkthrough is provided to create a plugin, make it available to the simulator and interact with the user while running the program being simulated.

1.1 Basic Requirements

Before writing a plugin, it is required that ARMSim# v1.5 be installed on the system. In addition one requires Visual Studio 2005 to use the supplied Starter kit and create the plugin.

1.2 Installation of the Starter Kit

The installationl directory of ARMSim# 1.5 is, by default, located in:

```
   C:\Program Files\University of Victoria\ARMSim.150
```

Browse into the **PluginStarterKit** subdirectory and double-click the file **SamplePluginLib.vsi**

The Visual Studio starter kit installer will now run. Follow the instructions to install the starter kit. This will integrate the ARMSim# plugin starter kit into the Visual Studio development environment. You are now ready to build your first plugin.

2. ARMSim# Extensions

ARMSim# can be extended in three ways which are described below. A single plugin can extend in only one way, or any combination of the three.

2.1 User Interface

The plugin can request a panel to be created in the simulator user interface. This panel is then available to the plugin to display controls and allow the user to interact with these controls. By utilizing the Panel object, the plugin can exert full control over the user interface allocated to it. This is the most useful and interesting extension for users.

2.2 Instruction Extension

The plugin can declare an opcode that is an “extension” to the standard ARM instruction set. The plugin establishes a request to the ARMSim# simulator, such that when this extra opcode is encountered, the plugin is notified. When this happens, the plugin can then execute the instruction and make whatever state changes are required in the simulator engine. This allows the creation of new instructions that do not exist in the standard ARM instruction set. Additionally the plugin can create a new mnemonic to be added to the parsing tables. This allows the programmer to use the new instruction as an assembler instruction instead of coding a binary value in the code.

This extension requires finding an unused opcode in the ARM instruction set. There are many gaps in the ARM instruction formats. A good description detailing how to extend the instruction set can be found in the Architecture Reference Manual, A3-27.

2.3 MemoryMapped IO

The plugin can declare that a range of memory is to be reserved for itself and when this memory is accessed the plugin must be informed. This allows the plugin to simulate hardware devices that use Memory Mapped IO. When the simulator encounters a memory operation on the reserved range, the plugin is notified and the plugin can make whatever state changes are required in the ARMSim# engine.

3. A Plugin Walkthrough

In this example we will create a plugin that extends the ARM instruction set by implementing a new hypothetical register in the ARM processor. We will call this new register, `RegisterX`. It will be identical to a general purpose ARM register. To support this new register we will create 2 new ARM instructions, MRX and MXR that will behave as follows:

`MRX{cond} <Rd>` transfer the X register to the specified ARM general purpose register Rd

`MXR{cond} <Rm>` transfer an ARM general purpose register Rm to the X register.

where:

`{cond}` is the standard ARM condition code specifier(optional), `<Rd>` is the destination ARM general purpose register, and `<Rm>` is the source ARM general purpose register

In addition, we will create a user interface component that will display the current contents in hexadecimal of register X to the user. To test these instructions, we will create a short ARM program to exercise the functionality of the new instructions.

3.1 Creating the Project

When a new project is created, one must specify its name to set the root namespace, assembly name, and project name, and ensure that the default component will be placed in the correct namespace. Here are the steps in Visual Studio to create the armSample control library and the armSample control.

- Start Visual Studio 2005
- On the **File** menu, point to **New**, and then click **Project** to open the **New Project** dialog box.

- •From the list of Visual C# projects, select the **SamplePluginLib** project template, type **armSampleLib** in the **Name** box, and then click **OK**.

  The project name, `armSampleLib`, is also assigned to the root namespace by default. The root namespace is used to qualify the names of components in the assembly. For example, if two assemblies provide components named `armSample`, one can specify this `armSample` component using `armSampleLib.armSample.`
- •In Solution Explorer, right-click **Class1.cs**, and then click **Rename**. Change the file name to **armSample.cs**.
- •On the **File** menu, click **Save All** to save the project.

You now have a complete framework for a functional plugin. The next step is to define some functionality for the plugin to perform.

3.2 Selecting an Opcode and Mask

For this example, the first step is to select an unused opcode from the ARM instruction set. The ARM contains gaps in the defined instruction space. This space is defined in the Architecture Reference Manual pA3-28 and is defined as follows:

```
cond | 0 1 1 x | x x x x | x x x x | x x x x | x x x x | x x x 1| x x x x
```

For this example we will select the following patterns for the opcodes:

`0x06000010`	for the MXR instruction and
`0x07000010`	for the MRX instruction

Next, we will construct a bitmask. This mask defines which bits are significant and which bits are optional. Any bit eqaul to “1” in the mask implies that the correspoding bit on the opcode must be the same. A “0” bit in the mask means that the correspoding bit in the opcode can be either a 1 or a 0.

For this example the mask require for both instructions is `0x0ffffff0` .

3.3 Implementing IARMPlugin

In **Solution Explorer**, double-click the file **armSample.cs** to open the file. The project template has created a skeleton plugin for you. Locate the init method:

```
public void init(IARMHost ihost)
{
     _ihost = ihost;
     _ihost.Load += onLoad;
}
```

This method is first called by the simulator when the plugin is located and initially created. In it we copy the reference to the simulator host. We then subscribe to the **onLoad** event. This event is fired once after all the plugins have been loaded and are ready to start. It is best to take a detailed look at what happens in the onLoad method. First of all, locate the onLoad method which should look as follows:

```
public void onLoad()
{
     //To request an opcode range to control, use the following call
     //_ihost.RequestOpcodeRange(0x016f0e10, 0x0fff0ff0, onExecute);
}//onLoad
```

(Some commented out code has been removed.)

This piece of sample code illustrates an example of subscribing to the simulator a specific opcode to be handled by the plugin. For our example we will use the base opcode and mask calculated above. Uncomment the “`RequestOpcodeRange`” call above and replace it with the following code:

```
_ihost.RequestOpcodeRange(0x06000010, 0x0ffffff0, onExecuteMXR);
```

This line of code requests the opcode for the `MXR` instruction. Now lets add the line of code to request the `MRX` instruction as well.

```
_ihost.RequestOpcodeRange(0x07000010, 0x0ffffff0, onExecuteMRX);
```

Now that we have requested the opcodes for the two new instructions,we can implement their execution. Notice that when we requested the opcodes we specified a function to be executed when these opcodes are encountered by the simulator engine. Lets first create the execution function for the `MXR` instruction, which we specified was the “`onExecuteMXR`”. Create the `onExecuteMXR` functions as follows:

```
//Parameters
//uint opcode : opcode encountered by simulator engine
//Return: The number of clock cycles required to execute
public uint onExecuteMXR(uint opcode)
{
     //extract source register(Fm)
     uint Fm = (opcode & 0x0000000f);

     //retrieve register Fm from the ARM cpu registers and store in
RegisterX
     _RegisterX = _ihost.getReg(Fm);

     //update the user interface
     _lbl.Text = "0x" + _RegisterX.ToString("x8");

     //return the number of clock cycles this instruction uses.
     return 3;
}
```

Notice that we are encoding the source register in the bottom 8 bits of the opcode. Once we have the source register we can make a request to the simulator engine to retrieve the register contents from the ARM cpu registers via the `getReg` function. Then we store the resulting register value into the `RegisterX` local variable which we will create now.

Add the following variable declaration to the class:

```
private uint _RegisterX=0;
```

This variable will hold the current value of the `RegisterX` which has an initial value of 0. Notice that the function returns a constant 3. The return value for the execute function is the number of clock cycles that this instructions requires to execute. In this case we are assume a constant 3 cycles.

Now we mustdefine the execution body for the `MRX` instruction:

```
//Parameters
//uint opcode : opcode encountered by simulator engine
//Return: The number of clock cycles required to execute
public uint onExecuteMRX(uint opcode)
{
     //extract destination register(Fd)
     uint Fd = (opcode & 0x0000000f);

     //store the current RegisterX to the ARM cpu register
     _ihost.setReg(Fd, _RegisterX);

     //return the number of clock cycles this instruction uses.
     return 3;
}
```

Now that we have requested the opcodes for the 2 new instructions and defined their behavoir, we need to create some useful mnemonics to represent these new instructions. Once we create these new mnemonics, the assembler parser will generate the opcode we have specified for the new instructions.

Defining a mnemonic requires four parameters as follows:

- •The mnemonic string
- •The base opcode
- •The register parameters
- •A function to form the resulting opcode

In the `onLoad` function, add the following code:

```
_ihost.InstallOpcode("mxr", 0x06000010, "R", formOpcodeMXR);
_ihost.InstallOpcode("mrx", 0x07000010, "R", formOpcodeMRX);
```

The first parameter is the mnemonic label, followed by the base opcode computed before. Next we specify the register parameters. Recall that the syntax we had decided for these instructions should be:

`MRX{cond} <Rd>`	to transfer the X register to the specified ARM general purpose register Rd
`MXR{cond} <Rm>`	to transfer a ARM general purpose register Rm to the X register.

Both instructions require one register parameter. This requires that we pass in the string “R” as the register parameter. The last parameter specifies the function that will be called when the assembler parser encounters this mnemonic. We need to define those functions now. Add the following code to the class:

```
//Parameters
//ref uint code : is the base opcode with the condition codes set. We need
   to
//encode the destination register in this opcode.
//int[] params : the register parameter parsed
//Return: True if successful, False if error
public bool formOpcodeMRX(ref uint code, int[] parms)
{
        //make sure the number of parameters is correct
        if (parms.Length != 1)
             return false;

        //encode destination register into bottom 8 bits
        code |= (uint)parms[0];    // Rd

        //return success
        return true;
}

//Parameters
//ref uint code : is the base opcode with the condition codes set. We need
   to
//encode the source register in this opcode.
//int[] params : the register parameter parsed
//Return: True if successful, False if error
public bool formOpcodeMXR(ref uint code, int[] parms)
{
        //make sure the number of parameters is correct
        if (parms.Length != 1)
             return false;

        //encode source register into bottom 8 bits
        code |= (uint)parms[0];    // Rm

        //return success
        return true;
}
```

These 2 functions form the final opcode that will be part of the final binary image of the program. Notice that we are encoding both the source and destination registers into the bottom 8 bits of the opcode and the onExecute functions extract the registers from the same location.

Now that the instructions are fully defined, we need to build the user interface elements that will display the `RegisterX` contents in the ARMSim# board controls view. The `onLoad` function is the place to setup for drawing. First we will request from the simulator a Panel object. This is the same as the.NET *Panel object* in the **System.Windows.Forms** namespace. Documentation on this object is available in the Visual Studio documentation.

Add the following variable to the class:

```
      private Panel _panel;
```

And add the following code into the onLoad function to request a Panel:

```
   //request a panel from the simulator boardview
   _panel = _ihost.RequestPanel(new Size(183,32));
   _panel.BackColor = Color.Red;
```

This code will request a Panel object from the simulator user interface that is located within the Board-View display area. In addition the requested size must be specified. Once we have the Panel object we can manipulate it like any other .NET forms object. We will specify a background colour of Red to highlight the Panel.

To display text within our panel we will need to create a Label object.

Add the following variable to the class:

```
      private Label _lbl;
```

Add the following code into the onLoad function to create the Label object, initialize it and add it to the Panel form requested earlier:

```
   _lbl = new Label();
   _lbl.Font = new Font("Microsoft Sans Serif", 16.2F, FontStyle.Bold,
      GraphicsUnit.Point, ((byte)(0)));

   _lbl.Text = "0x00000000";
   _lbl.Dock = System.Windows.Forms.DockStyle.Fill;

   //and add all the created controls to the panel
   _panel.Controls.Add(_lbl);
```

We now have a fully functional plugin that implements the two new instructions that we wish to simulate. The next steps will be to build the project and test the new instructions.

3.4 Building the Project

On the **Build** menu, point to **Rebuild Solution**, and then click. The project will now be compiled and deployed to the ARMSim# 1.50 install directory. The plugin is now ready to use.

If any compiler error or warnings are generated, fix these now.

3.5 Write a Test ARM Program

The last step is to write an ARM test program that will use the new instructions. We will create some test conditions where we can manipulate RegisterX and observe the results.

Create a text document and type the following ARM program:

```
           .text
           .global _start
   _start:
           @load a constant value into r5
           ldr   r5,=0x12345678

           @move r5 into RegisterX
           mxr   r5

           @check the user interface!

           @clear r5
           mov   r5,#0

           @move RegisterX back into r5
           mrx   r5

           @end program
           swi   0x11

      .end
```

Load this program into ARMSim# and single step over the `mxr` instruction and notice the result which is written into register 5. Check in the user interface that the correct value of `RegisterX` is displayed. Step over the next few instructions that clear register 5 and move `RegisterX` into it. Did register 5 update properly?

Below is a complete code listing for `armSample.cs`. Some commented lines have been removed:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using System.Drawing;

using ARMPluginInterfaces;

namespace SamplePluginLib
{
    public class armSample : IARMPlugin
    {
        private IARMHost _ihost;
        private Panel _panel;
        private Label _lbl;
        private uint _RegisterX = 0;

        //This function is called once the plugin has been loaded.
        //From this function you can subscribe to the events the
        //simulator supports.
        public void init(IARMHost ihost)
        {
            _ihost = ihost;
            _ihost.Load += onLoad;
        }

        //This function is called after all the plugins have been loaded
   and their
        //init methods called.
        public void onLoad()
        {
           //uses the undefined instruction space specified in ARm manual
   pA3-28
            _ihost.RequestOpcodeRange(0x06000010, 0x0ffffff0,
   onExecuteMXR);
            _ihost.RequestOpcodeRange(0x07000010, 0x0ffffff0,
   onExecuteMRX);

            _ihost.InstallOpcode("mxr", 0x06000010, "R", formOpcodeMXR);
            _ihost.InstallOpcode("mrx", 0x07000010, "R", formOpcodeMRX);

            //request a panel from the simulator boardview
            _panel = _ihost.RequestPanel(new Size(183, 32));
            _panel.BackColor = Color.Red;

            _lbl = new Label();
            _lbl.Font = new Font("Microsoft Sans Serif", 16.2F,
   FontStyle.Bold, GraphicsUnit.Point, ((byte)(0)));
```

```
        _lbl.Text = "0x00000000";
        _lbl.Dock = System.Windows.Forms.DockStyle.Fill;


        //and add all the created controls to the panel
        _panel.Controls.Add(_lbl);

    }//onLoad

    //Parameters
    //ref uint code : is the base opcode with the condition codes set.
    // We need to encode the destination register in this opcode.
    //int[] params : the register parameter parsed
    //Return: True if successful, False if error
    public bool formOpcodeMRX(ref uint code, int[] parms)
    {
        //make sure the number of parameters is correct
        if (parms.Length != 1)
            return false;

        //encode destination register into bottom 8 bits
        code |= (uint)parms[0];                 // Rd

        //return success
        return true;
    }

    //Parameters
    //ref uint code : is the base opcode with the condition codes set.
    //We need to encode the source register in this opcode.
    //int[] params : the register parameter parsed
    //Return: True if successful, False if error
    public bool formOpcodeMXR(ref uint code, int[] parms)
    {
        //make sure the number of parameters is correct
        if (parms.Length != 1)
            return false;

        //encode source register into bottom 8 bits
        code |= (uint)parms[0];                 // Rm

        //return success
        return true;
    }

    //Parameters
    //uint opcode : opcode encountered by simulator engine
    //Return: The number of clock cycles required to execute
    public uint onExecuteMXR(uint opcode)
    {
            //extract source register(Fm)
            uint Fm = (opcode & 0x0000000f);

           //retrieve register Fm from the ARM cpu registers and store in
   //RegisterX
            _RegisterX = _ihost.getReg(Fm);

            //update the user interface
            _lbl.Text = "0x" + _RegisterX.ToString("x8");

            //return the number of clock cycles this instruction uses.
            return 3;
        }

        //Parameters
        //uint opcode : opcode encountered by simulator engine
        //Return: The number of clock cycles required to execute
        public uint onExecuteMRX(uint opcode)
        {
            //extract destination register(Fd)
            uint Fd = (opcode & 0x0000000f);

            //store the current RegisterX to the ARM cpu register
            _ihost.setReg(Fd, _RegisterX);

            //return the number of clock cycles this instruction uses.
            return 3;
        }
    }
}
```