Tutorial1: Your First Personality

Initial Author: Kevin Townsend ktown@iastate.edu

Note—This was intended to be a series of tutorials one building on the next. For example Tutorial2 could be "Multi-cycle Instructions and Memory Controllers". I included my name and e-mail incase people had questions about this tutorial or request for the original doc file. There is no need to include my name on derivative work. Feel free to copy this to the wiki or other medium.

This tutorial will walk you through the steps to create a Convey personality.

Run the "newCnyProject" script. Name your project. This project will use "First" as its name. Pick a number for your project. This number can be between 65000 and 65535. This tutorial will use 65001. Don't worry about this number conflicting with other personalities. The script creates a new personality directory for your personality.

```
cd
cd  $CNYSCRIPTS/newCnyProject
```

Try making and running your application.

```
cd caeFirst/appFirst
make
.\run
```

Now let's decided what we want our personality to do. This tutorial will create a personality that takes in 1, 8-byte word and modifies the word and then returns it.. Specifically it will swap the first byte with the last byte of the word.

First let's start at the program that will be running on the host processor. We want to ask for a number, process it and return it. Edit appFirst.cpp with your favorite text editor. Then replace:

```
#TODO: replace with own cp call
cout << "@user:calling coprocessor" << endl;
copcall_fmt(sig, cpTalk, "");
cout << "@user:calling coprocessor" << endl;
```

with:

```
uint64 Input;
uint64 Shuffle;
//Ask for a number
cout << "What is the answer to life the universe and
everything?:";
cin >> Input;
cout << "@user:calling shuffler" << endl;
```

```
//TODO: call coprocessor
cout << "@user:shuffler finished" << endl;
//print output to screen
cout << hex <<"Input:0x" << Input << endl;
cout << hex << "Output:0x" << Shuffle << endl;
```

Now we want an assembly function to run on the coprocessor that the host can call. Edit cpFirst.s. Go to the cpTalk function and replace its name with shuffler. Arguments are usually passed using the registers and the mov instruction. The same method is used for the return values. We end up with the following:

Add the following to cpFirst.s

```
    .globl shuffler
    .type shuffle. @function
    .signature pdk=65001
shuffler:
    mov.ae0 %a8, $0, %aeg
    caep00.ae0 $0
    mov.ae0 %aeg, $0, %a8
    rtn
```

This will run but since we have not written anything for caep00 the instruction does nothing (nop).

Go back to FirstApp.cpp
Add the function reference

```
Extern "C" long shuffler
```

At the TODO statement add:

```
Shuffle = l_copcall_fmt(sig, shuffler,"A",Input);
```

You should have:

```
    uint64 Input;
    uint64 Shuffle;
    cout << "What is the answer to life, the universe and
everything:";
    cin >> Input;
    Shuffle = l_copcall_fmt(sig, shuffler, "A", Input);
    cout << hex << "input:0x" << Input << endl;
    cout << "Output:0x" << Shuffle << endl;
```

Make and see if it compiles.

Before we play with the Verilog let's make an emulator.

Emulator

The emulator is designed to do the same thing as the coprocessor but is easier to write and debug since it is written in C++. The emulator will help debug the verilog code. Replace the case statement with the following:

```
case 0x20:
    {
    uint64 tmp = ReadAeg(aeId, 0);
    WriteAeg(aeId, 0, (tmp<<56)|(tmp &
           0x00FFFFFFFFFFFF00)|(tmp>>56));
    break;
    }
```

Now run the application again:

```
cd ../appFirst
make
run
```

The output should be something like the following:

```
[ktown@xilinx-1 appFirst]$ run
What is the answer to life, the universe and everything:42
  Waiting for remote AE to connect on port 45341...
connected.
Hello World from emulated ae0
input:0x2a
Output:0x2a00000000000000
coprocessor calls: 1
AeSim: Connection from CpSim closed, exiting.
```

Verilog

The fun part, some of the Verilog has been written to help you create your personality. This includes some instruction decoding and aeg register logic. We need to add logic to use aeg0. Also, we need to add logic to execute caep00. Let's start with the aeg register logic. Go the following section of code:

```
   //
   // AEG[0..NA-1] Registers
   //
```

```
    localparam NA = 51;
    localparam NB = 6;       // Number of bits to represent
NAEG

    assign cae_aeg_cnt = NA;

    //output of aeg registers
    wire [63:0]  w_aeg[NA-1:0];

    genvar g;
    generate for (g=0; g<NA; g=g+1) begin : g0
      reg [63:0] c_aeg, r_aeg;

      always @* begin
        case (g)
//TODO: add cases for registers to be written to
            default: c_aeg = r_aeg;
        endcase
      end

      wire c_aeg_we = inst_aeg_wr && inst_aeg_idx[NB-1:0]
== g;

      always @(posedge clk) begin
        if (c_aeg_we) begin
            r_aeg <= cae_data;
            $display("writing: %x", cae_data);
        end
        else
            r_aeg <= c_aeg;
        end
      assign w_aeg[g] = r_aeg;
    end endgenerate
```

Add the following code in bold:

```
//*************************************************************
*****************
   //               PERSONALITY SPECIFIC LOGIC

//*************************************************************
*****************
    reg [63:0] c_shuffle_out;
    reg return_shuffle
```

```
   //
   // AEG[0..NA-1] Registers
   //
    localparam NA = 51;
    localparam NB = 6;       // Number of bits to represent
NAEG

    assign cae_aeg_cnt = NA;

    //output of aeg registers
    wire [63:0]  w_aeg[NA-1:0];

    genvar g;
    generate for (g=0; g<NA; g=g+1) begin : g0
      reg [63:0] c_aeg, r_aeg;

      always @* begin
        case (g)
            0: begin
                if(return_shuffle)
                    c_aeg = c_shuffle_out;
                else
                    c_aeg = r_aeg;
            end
            default: c_aeg = r_aeg;
        endcase
      end

      wire c_aeg_we = inst_aeg_wr && inst_aeg_idx[NB-1:0]
== g;

      always @(posedge clk) begin
        if (c_aeg_we) begin
            r_aeg <= cae_data;
            $display("writing: %x", cae_data);
        end
        else
            r_aeg <= c_aeg;
        end
      assign w_aeg[g] = r_aeg;
    end endgenerate
```

Now add the logic to correctly stall the processor.

```
   //logic for using cae IMPORTANT. cae_idle should be 0
when executing a custom instruction and 1 otherwise.
   //cae_stall should be 1 when when exectuting a custom
instruction and 0 otherwise.
wire c_caep00;
reg r_caep00;
assign c_caep00 = inst_caep == 5'd0 && inst_val;
always @(posedge clk) begin
    r_caep00 <= c_caep00;
end

   assign cae_idle  = !r_caep00;
   assign cae_stall = c_caep00 || r_caep00;
```

Now add the logic to shuffle the bits around:

```
    always @(posedge clk) begin
        if(inst_caep == 5'd0 && inst_val) begin
            $display("@simulation:Shufflin' from simulated
ae%d", i_aeid);
            return_shuffle <= 1;
            c_shuffle_out <= (w_aeg[0] << 56) | (w_aeg[0] &
64'H00FFFFFFFFFFFF00) | (w_aeg[0] >> 56);
        end
        else
            return_shuffle <= 0;
    end
```

Now run the application using the simulation:

```
cd ../appFirst
./run -vsim
```

You should see at the end of the simulation "no errors". This means the simulation matched what the emulator was doing. If you don't see this, check your code.

Bitfile

Now you can create a bitfile to be loaded on the coprocessor.

```
cd ../phys
make
```

This should take 1 to 2 hours, so use NX client and suspend the session and come back later to finish.

```
make release
#takes about 5 minutes
cd ..
cp ../caeFirst.release/12_07_08_35/cae_fpga.tgz
personalities/65001.1.1.1.0/ae_fpga.tgz
```

Finally you can run your application on the actual hardware.

```
cd appFirst
./runcp
```

That's all.

Conclusion

You should now be able to create a personality on your own. The following is an exercise you can try on your own:

Odd Even Segregation:

This personality has one instruction. It takes 2, 64 bit arguments. Store the even bits of the first argument into the least significant 4 bytes of the return value. Store the even bits of the second argument into the most significant 4 bytes.

Bonus:
Also return a 64bit value with the odd bits. Note, you may need to have 2 assembly functions.