Tutorial 3
RISC-V and Debugging

Quan Nguyen
(Moonlights as an amateur instruction set evangelist)

Outline
• RISC-V processor (from lab 5 onwards)
• Debugging
RISC-V Processor

and the SCE-MI Infrastructure

RISC-V Interface

```verilog
interface Proc;
  method Action hostToCpu(Addr startpc);
  method ActionValue#(CpuToHostData) cpuToHost;
interface MemInitIfc iMemInit;
interface MemInitIfc dMemInit;
endinterface

typedef struct {
  CpuToHostType c2hType;
  Bit#(16) data;
} CpuToHostData deriving(Bits, Eq);

typedef enum {
  ExitCode, PrintChar, PrintIntLow, PrintIntHigh
} CpuToHostType deriving(Bits, Eq);
```
RISC-V Interface

Host (Testbench)

RISC-V Interface - CpuToHost

- **Write mtohost CSR:** `csrw mtohost, rs1`
  - `rs1[15:0]`: data
    - 32-bit Integer needs two writes
  - `rs1[17:16]`: `c2hType`
    - 0: Exit code
    - 1: Print character
    - 2: Print int, low 16 bits
    - 3: Print int, high 16 bits

```haskell
typedef struct {
    CpuToHostType c2hType;
    Bit#(16) data;
} CpuToHostData deriving(Bits, Eq);
```
RISC-V Interface - Others

- hostToCpu
  - Tells the processor to start running from the given address
- iMemInit/dMemInit
  - Used to initialize iMem and dMem
  - Can also be used to check when initialization is done
  - Defined in MemInit.bsv

```haskell
interface MemInitIfc;
  interface Put#(MemInit) request;
  method Bool#(MemInit) done();
endinterface
```

Connecting the RISC-V interface

- Previous labs: software testbenches
- Want more advanced programs
  - Load multiple programs and display their outputs
- How?
  - C++ testbench connected to BSV module with SceMi
SceMi Interface

Load Program

• Bypass this step in simulation
Load Program

Simulation: load with mem.vmh (fixed file name)
- Copy <test>.riscv.vmh to mem.vmh

Start Processor

Starting PC
0x200
Print & Exit

Get reg
c2hType:
1,2,3: print
0: Exit
Data == 0
   PASSED
Data != 0
   FAILED

SceMi Interface: Simulation

Compiled Program: tb

Compiled BSim Executable: bsim_dut
The same SceMi Testbench can be used to test hardware FPGA designs too!

Compiled Program: tb

Debugging

(and how to stay sane)
Syntax?

• Bluespec compiler very helpful!
  ■ Line/column number information

• Check your types
  ■ Reg? EHR? Wire?
  ■ Be careful of `let` statements

Things to try

1. Reason carefully about your design
   ■ (If you can’t, rethink your design’s modularity)

2. Use $display() within rules, Action/ActionValue methods
   ■ In functions, either see #1 or place $display in rule containing function

3. Use waveforms (up next)

4. When debugging processors, use PC and assembly code

5. Get some rest and come back later
$display

- Prints text to console
  - $display("hello");
  - $display("the value is %d", 42);
  - $display("struct contents: ", fshow(s));
- Formats:
  - %d: decimal
  - %x: hexadecimal
  - %o: octal
  - %b: binary
- Use %0d (zero), %0x, %0o, %0b to print without whitespace padding

Ways to Display Values

fshow

- fshow is a function in the FShow typeclass
  - (see Tutorial 2 for typeclasses)
- It can be derived for enumerations and structs
  - In structs, can define recursively

```haskell
typedef enum {Red, Blue} Colors deriving(FShow);
Color c = Red;
$display("c is ", fshow(c));
```

Prints “c is Red”
Manually specifying your own

- Use \$format (similar to \$display) to return Fmt
  - Combine multiple fshows, Strings together

```
instance FShow#(Color);
  function Fmt fshow(Color c);
    case (c)
      Red: return fshow("Rojo");
      Blue: return fshow("Azul");
      default: return fshow("?");
    endcase
  endfunction
endinstance
```

BSV Debugging
Waveform Viewer

- Simulation executables can dump VCD waveforms
  - ./bsim_dut -V test.vcd
- Produces test.vcd containing the values of all the signals
  used in the simulator
  - Not the same as normal BSV signals
- VCD files can be viewed by a waveform viewer
  - Such as gtkwave
- The signal names and values in test.vcd can be hard to understand
  - Especially for structures and enumerations
  - But try your best
Bluespec GUI Example

- gtkwave: waveform viewer
  - installed on vlsifarm machines
- SSH with -X
  - ssh -X you@vlsifarm-0x.mit.edu
  - Windows: might need Cygwin/X
  - Mac OS X: might need XQuartz

Bluespec project

- Create Bluespec GUI project file
  - project.bld: workstation-project-file

```bash
[onecycle]
extends-target: bsim_dut
bsv-define: PROC_FILE-OneCycle SIM
workstation-project-file: onecycle.bspec
```

- $ cd scemi/sim
- $ build -v onecycle
- onecycle.bspec is the project file
Simulation: generate VCD file

```bash
$ cp ../../../programs/build/assembly/vmh/simple.riscv.vmh mem.vmh
$ ./bsim_dut -V out.vcd > log.txt &
$ ./tb
```

Open Bluespec GUI

```bash
$ bluespec onecycle.bspec
```
Open Module Browser

- Window → Module Browser

Module Browser

Top level module
Load VCD

Waveform Viewer
Add Signals to View

- Add clock

Add Signals to View

- Add PC
Add Signals to View

- Add `csrf.wr` method

Add Signals to View

- Add `can_fire`, `will_fire` for rule `doProc`
Problem

- No internal signals
  - Optimized away during compilation
  - But I want to see them for debugging, e.g., dInst
Solution – Probe

- Writes to mkProbe will be recorded in VCD file
  - Instantiate mkProbe for each signal that we want to record

```verilog
interface Probe#(type t);
    method Action _write(t x);
endinterface

module mkProbe(Probe#(t)) provisos(Bits#(t, sz));
```

Probe Example

```verilog
import Probe::*;
...
module mkProc(Proc);
    mkProbe#(DecodedInst) dInstProbe <- mkProbe;
    ...
    rule doProc;
    ...
    let dInst = decode(inst);
    dInstProbe <= dInst;
    ...
endrule
...
endmodule
```
After Adding Probe
Combine with Assembly

- Assembly code of compiled tests are in
  - programs/build/*/dump
- Refer to the assembly code for debugging

Debugging Scheduling

- GUI: schedule Analysis
  - The same as "-show-schedule" BSC flag
Select mkProc

Output of Schedule Analysis

- Rule order: execution order of rules and methods
  - From top to bottom
Alternative for Schedule Analysis

- Add `--show-schedule` to `project.bld`

```
[DEFAULT]
default-targets: all
bsc-compile-options: -aggressive-conditions -keep-fires show-schedule
bsc-link-options: -Xc++ -00 -keep-fires
```

- Generate `info_dir/*/sched`
- Contains same information as GUI outputs

Credits

- Considerable previous material adapted from last year’s tutorials by Sizhuo Zhang and Andy Wright