OpenSpyser Release v1.0

OpenSpyser is a open source microprocessor based on OpenSPARC and DySER. OpenSpyser project includes the benchmarks, software simulation tools, hardware verification and simulation framework, and FPGA prototype. The framework is built to help evaluate an in-core accelerator, DySER.

The OpenSpyser project in developed in Vertical group, UW-Madison Computer Science Department. Here are the contributors:

- Chen-Han Ho
- Chris Frericks
- Jesse Benson
- Ryan Cofell
- Tony Nowatzki
- Venkatraman Govindaraju
- Zachary Marzec

Our advisor is professor Karu Sankaralingam [http://pages.cs.wisc.edu/~karu/wiki/]

OpenSpyser wiki pages:

- **Tutorial**: Evaluation with Gem5, VCS, and FPGA.
- **Manual**: Manual for the tools in the toolchain.
- **Release issues**: Known issues in the release.
- **FAQ**: Frequently asked questions and answers.

Tutorial

This tutorial will guide you to run your first gem5 simulation, VCS simulation and FPGA simulation.

For now, the tutorial is only for UW-Madison students.

Preparation

Things you need for a one-time setup:

- A Linux machine, we tested the framework on supported CSL machine which runs 64 bit RedHat 6 Linux.
- At least 10GB of free hard disk space
• Synopsys tools, on CSL machine this is installed in /s/, you can source /s/synopsys-2012_06_27/bin/synopsys_env.sh to use the tools.

• SW toolchain:
  • UW-Madison CSL RHEL 6 Machine installation instructions:
    • Add gperf and qmake in the path.

      ```bash
      export PATH=/unsup/gperf/amd64_rhel6/bin:/unsup/qt-4.6.3/bin:$PATH
      ```

    • The SW toolchain will not work in amd64_rhel5 environment in CS environment as it does not have updated tools (libtools, autoconf etc.,) which we rely on to build the cross compiler.

  • Fedora 17 installation instructions:
    • you need the following packages: (and any packages they depend on)
    • wget gcc gcc-c++ libtool gperf python python-devel zlib-devel scons bison flex qt4-devel texinfo patch ncurses-devel libmpc-devel
    • In default environment of fedora 17, qmake may not be in the default path. You may need to use qmake-qt4 instead of qmake to build dysched successfully.

  • Xilinx EDK 10.1
    • UW-Madison environment setup on vega:

      ```bash
      export PATH=$PATH:/opt/Xilinx/10.1/EDK/gnu/microblaze/lin64/bin/
      source /opt/Xilinx/10.1/ISE/settings64.sh
      source /opt/Xilinx/10.1/EDK/settings64.sh
      ```

  • Xilinx XUPV5-LX110T Evaluation Platform
    • JTAG interface to download and configure FPGA
    • Null modem cable
    • Internet

Build the simulation framework

The openspyser evaluation framework contains two pieces:

• SW toolchain files
  • Download/copy dyser.tar.bz2 from

      ```bash
      /p/vertical/projects/dyser/svn/sparc/
      ```

• HW toolchain files
  • Check out the HW framework from svn. This directory occupies 2.3 GB. Additional files created when simulating the verilog may exceed 10 GB in space.

      ```bash
      svn co file:///p/vertical/projects/dyser/svn/verilog/tags/pre-release-0.3 <check_out_dir>
      ```

The following instructions describe how to build the toolchain.

• Building SW toolchain:
  • Untar:

      ```bash
      tar xjf dyser.tar.bz2
      ```

  • Change directory to the dyser directory you just untarred

      ```bash
      cd dyser
      ```
Edit build-tools.sh line 4 to change the Install directory. The variable INSTALL_DIR basically should point to the top directory where you want the toolchain to be installed. Default is dyser-tools.

Execute the build-tools.sh

```
bash build-tools.sh
```

- toolchain (compiler, assembler etc.,) will be installed in `$(INSTALL_DIR)/toolchain`
- Simulator (gem5) will be in `$(INSTALL_DIR)/gem5`
- Tools (dysched, gen_dyser_config etc.,) will be in `$(INSTALL_DIR)/tools`

Building HW toolchain:
- You will find three directories in your `<checkout_dir>`: dyser-1.0, hardDySER, opensparc. The detailed description of the toolchain is in the Manual section. To set up the toolchain, navigate to:

  `<checkout_dir>/opensparc`

  and look for

  `OpenSPARCT1.bash`

- `<checkout_dir>/opensparc/OpenSPARCT1.bash`: this is the file you need to modify and source for your environment
  - Modify line 6(HW) and 7(SW) to point to your install directory
    
    ```
    # ***Modification required for new install***
    # Top of openparc portion
    export HW_ROOT="<path to your checkout dir>"
    export SW_ROOT="<path to your install dir>"
    ```

    The HW_ROOT should be your `<check_out_dir>`, and the SW_ROOT is the INSTALL_DIR you set in previous step.

  - Modify line 23 for the scratch space of VCS object files. Make sure you can access the scratch space you assigned.
    
    ```
    # ***Modification required for new install***
    #Regression run-time scratchspace
    export DRMJOBSCRATCHSPACE=/scratch/vcsjobscratch
    ```

  - Run:
    ```
    source OpenSPARCT1.bash
    ```

Now we are ready to run benchmarks!

Run a benchmark

The Openspyser evaluation framework can evaluate the design on 3 different platforms:

- gem5
- Synopsys VCS
- Xilinx FPGA

We will begin the tutorial with a test benchmark, cumsum.
Run gem5 simulation

Navigate to

$DV_ROOT/verif/diag/c/hardDYSER/cumsum/splyser/

Run:

make run_perf

This will compile the cumsum splyser benchmark, and run it in gem5. The output of gem5 is in the m5out/ directory:

- m5out/gem5.log: gem5’s simulation log
- m5out/stats.txt: the report, system.switch_cpus.numCycles is the number of cycles that we are interested.
- m5out/trace.log: this trace file tells you what happened in DySER.

Here is the simulation result we have:

```
system.switch_cpus.numCycles 713288
```

Run VCS simulation

Navigate to

$DV_ROOT/regr_runs/hardDYSER/cumsum.splyser.1w/

Run:

./run.sh

This will build the VCS model from opensplyser with cumsum hardDYSER verilog, compile and link the cumsum benchmark, and run the VCS simulation. After simulation, many files will be created in the current directory. Here are some important files:

- sims.log : the sims simulation log file, sims is a tool provided in Sun OpenSPARC T1 project.
- sim.log : the simulation trace, which have to be processed for human to read.
- ./build : the directory that contains the objects and assembly files.

To interpret the sim.log, run:

```
procvlog sim.log > vlog.log
```

procvlog is a tool provided in OpenSPARC T1. It interprets sim.log and outputs a more readable log file.

Next, in vlog.log, grep two marker instructions we used to define the code region of interest. In gem5 they defines the start and stop of detail timing simulation, and in VCS simulation they are two dummy instructions.

```
grep "sethi %hi(0xff000000), %g0" vlog.log ;grep "sethi %hi(0xff00000), %g0" vlog.log
```

You’ll see the output like this
The first column is the VCS simulation time. The only thing left to extract a cycle count is to subtract two values and divide by 832 (clock period in VCS time). Here we can do a little bash scripting to extract final cycle count. Run:

```bash
x=`grep "sethi \%hi(0xff000000), \%g0" vlog.log | awk '{print $1}'`;
\n\ny=`grep "sethi \%hi(0x0ff0000), \%g0" vlog.log | awk '{print $1}'`;
\nx=$(x/832); y=$(y/832); expr $((y - x)/832))
```

Here is the result we have:

871032

Congratulations! You have completed VCS simulation!

**Run a benchmark on FPGA**

(Make sure you have read the [preparation](#).)

The Opensplicyser project is based on OpenSPARC, and the detail of OpenSPARC FPGA simulation can be found in `$DV_ROOT/doc/OpenSPARCT1_DVGuide.pdf`. Here we will give a simple example showing how to run a program on the FPGA. First, navigate to

`$DV_ROOT/design/sys/edk/`

We provide a `cumsum.bit` which contains a modified sparc core and a simple hardDySER core in `$DV_ROOT/design/sys/edk/tutorial`. To use this bitstream, first create an implementation directory:

`mkdir implementation`

**Copy the bitstream:**

`cp ./tutorial/cumsum.bit implementation/download.bit`

**Download bitstream:**

`impact -batch etc/download.cmd`

Next, we have to download firmware, os, prom using XMD:

`xmd -xmp system.xmp -opt etc/xmd_microblaze_0.opt`

**In XMD:**

```
XMD% dow ccx-firmware/executable.elf
XMD% dow -data os/Ubuntu/7.10-gutsy/proto/ramdisk.ubuntu-7.10-gutsy.gz 0x8af00000
XMD% dow -data os/proms/1cl1_0bp_prom.bin 0x8ff00000
```

Now, before we run the executable on MicroBlaze, we need to connect to the FPGA through the serial port to interact with OpenSplyser. Here we use minicom as our modem program. To use it, you need root access. Open a new terminal and type:

`sudo minicom`
The parameters of minicom connection should be set to 9600 8N1. In minicom, the parameter can be set by ctrl A-Z → P → c Now switch back to XMD, we can run the microblaze in XMD by:

```
XMD$ run
```

Switch back to minicom, you should be able to see microblaze is running. From now on the minicom will be your terminal to the OpenSlyser FPGA. After decompressing the images, we will see an ok prompt. Now, (in minicom) type:

```
boot
```

After boot up (this may take a while), login with username root and password root. We use dhclient to use dhcp to get internet to work:

```
root@tl-fpga-00:~# dhclient eth0
```

Next, we can create a binary for OpenSlyser to run. Navigate to:

```
$DV_ROOT/verif/diag/c/hardDYSER/cumsum/splyser/
```

Edit Makefile, change the -DFF flag in CFLAGS to -DFPGA

```
CFLAGS += -03 $(FLAGS) -DONEWIDE -DFPGA
```

And run:

```
Make
```

`cumsum.splyser` will be created.

To send executables to FPGA, we can use ftp. After putting `cumsum.splyser` to your ftp server, run

```
root@tl-fpga-00:~# ftp <your_ftp_server>
```

After login, download `cumsum.splyser`:

```
ftp> get cumsum.splyser
ftp> exit
```

(UW-Madison instructions: you can ftp to 128.105.102.19 with your account on vega and download the file)

Change the permission of the executable:

```
chmod u+x cumsum.splyser
```

Run the benchmark:

```
./cumsum.splyser
```

You should see something like:

```
root@tl-fpga-00:~# ./cumsum.splyser
tick elapsed = 21819516195438592
splyser check: 262140.000000
pass()
```
Congratulations! You have run a FPGA benchmark on OpenSlyser! Next, let’s try to build a benchmark from scratch!

**Build your own benchmark**

The OpenSlyser framework is used to evaluate the DySER architecture. To write a DySER program/benchmark, we use the following macros in C:

- **DySEND(reg,dy_port)** – send register to dyser (used with variables like int, float, etc)
- **DyLOAD(mem,dy_port)** – send memory to dyser (used with memory locations like int*, float*, etc)
- **DyRECV(dy_port,reg)** – receive from dyser to register
- **DySTORE(dy_port,mem)** – receive from memory to dyser

Details of the macros can be found in `$DV_ROOT/verif/diag/c/include/dyser-dlp-sparc.h`. Reading through the codes in `$DV_ROOT/verif/diag/c/hardDySER/<benchmark_name>` can help you understand DySER programming. Besides the c code, a DySER config should be created with the DySER program using the dysched GUI. With DySER program and DySER config file, we can evaluate the DySER architecture in the OpenSlyser framework.

Here, we’ll pretend that we have completed DySER programming and use the cumsum c code and the config file for the tutorial. First, navigate to:

```
$DV_ROOT/verif/diag/c/hardDySER
```

**Create a test directory:**

```
mkdir -p ./test/splyser
```

The splyser means this is a “dyserized” program. For each benchmark in the release, there is scalar and splyser version. Next, copy the c file and config file from cumsum/splyser

```
cp ./cumsum/splyser/cumsum.c ./test/splyser/test.c
cp ./cumsum/splyser/16input-cumsum-8W.hardDySER ./test/splyser/test.config
```

Now we have the c code and dyser config file. Navigate to the directory you created:

```
$DV_ROOT/verif/diag/c/hardDySER/test/splyser
```

**Before we compile the code, we need to process the config file for OpenSlyser:**

```
python $HW_ROOT/hardDySER/tools/genCore.py -f test.config -o test.v
```

This setp creates a port mapping and a hardDySER verilog. Now, in the directory you should see:

```
test.c test.config test.config.hardDySER.conf test.v
```

The 16input-cumsum-8W.hardDySER.hardDySER.conf is the file that has the original config and opensplyser port mapping, and the test.v is the hardDySER verilog module.

The port mapping maps the logical I/O ports in GUI to DySER physical I/O ports. (Here, in fact, the original config file has the port mapping. However, when you create your own config
from GUI, the generated file will not contain port mapping information) The hardDySER
verilog is a simplified DySER module that removes the unnecessary switches and functional
units.

Next, we want to create a Makefile to compile and run the program. We will use
$DV_ROOT/verif/diag/c/hardDYSER/config.mk for benchmark-invariant rules and variables.
To use config.mk, we have to define several variables in the Makefile:

    ROOTDIR ?= ../...
    BUILDDIR ?= .
    BUILDPATH ?= obj

The above variables set up the location

    CFLAGS += -O3 $(FLAGS) -DONEWIDE -DFF

The above is the CFLAGS you want to add, the -DFF is an flag for gem5 detail timing
simulation. More detail can be found in the SW manual.

    TARGET    = $(BUILDDIR)/test.splyser
    SRC       = test.c

The above variables sets up the build target and source code

    DYSER_HEADER  = dyserconfig.h
    DYSER_SCHED   = test.config

The above tells the framework to generate a dyser header file from the dyser config. This
header file contains the configuration and will be compiled into the binary. Last, include the
config.mk:

    include $(ROOTDIR)/config.mk

Put all of the above in the Makefile, and run

    make run_perf

Congratulations! You completed the gem5 simulation of a new program!

Build VCS simulation

First, set up a test directory. We put all benchmarks in $DV_ROOT/regr_runs, so navigate to:

   $DV_ROOT/regr_runs
   mkdir test

To run VCS simulation, first we have to create a diagnose file for the toolchain to compile.
Navigate to:

   $DV_ROOT/regr_runs/test

Create test.splyser.s:

    #define USE_STACK      /* Turns on stack in template_mt.s */
    #define STACKSIZE 8192  /* Sets stack size */
    #include "c/template_mt.s"  /* Provides all vcs run environment */

    MIDAS_CC FILE=hardDYSER/cumsum/splyser/cumsum.c ARGS=-O3 \

The first 3 lines sets up the vcs simulation environment, and the MIDAS_CC is like compile rules in Makefile. More details can be found in OpenSPARC T1 Design and Verification guide, MIDAS appendix. The OpenSPARC T1 guide is in $DV_ROOT/doc

Next, we need to create a file list that contains all of the slyser verilog files. In the same directory, create test.flist:

```
+incdir+$DYS_ROOT/rtl
$DV_ROOT/design/sys/iop/SPARC_Changes/dyser/dyser_block.v
$DYS_ROOT/rtl/dyser_config.v
$DYS_ROOT/rtl/input_bridge.v
$DYS_ROOT/rtl/comp_logic.v
$DYS_ROOT/rtl/output_bridge.v
$DYS_ROOT/rtl/fu_stage.v
$DYS_ROOT/rtl/ff_stage.v
$DYS_ROOT/rtl/sw_stage.v
$DYS_ROOT/rtl/functional_unit.v
$DYS_ROOT/rtl/orl200_emultp2_32x32.v
$DYS_ROOT/rtl/switch_output.v
$DYS_ROOT/rtl/switch_lto2.v
$DYS_ROOT/rtl/broadcast_config.v
$DYS_ROOT/rtl/broadopcode.v
$DYS_ROOT/rtl/broadcast.v
$DYS_ROOT/rtl/fifol6.v
$DYS_ROOT/rtl/dyser.v
$DV_ROOT/verif/diag/c/hardDYSER/test/slyser/test.v
$DYS_ROOT/rtl/broadrom.v
$DYS_ROOT/fpu/except.v
$DYS_ROOT/fpu/fcmp.v
$DYS_ROOT/fpu/single_fpu.v
$DYS_ROOT/fpu/post_norm.v
$DYS_ROOT/fpu/pre_norm.v
$DYS_ROOT/fpu/pre_norm_fmul.v
$DYS_ROOT/fpu/primitives.v
```

These are the DySER verilog files that will be used in simulation. Notice that $DV_ROOT/verif/diag/c/hardDYSER/test/slyser/test.v is what we just created.

Last is to create the script to run the simulation, the example script is in $DV_ROOT/regr_runs/hardDYSER/vcs.config.sh. Create run.sh:

```
cp $DV_ROOT/regr_runs/hardDYSER/vcs.config.sh ./run.sh
```

Modify line 17, the variable “app_args”, to generate result in current directory:

```
app_args="-regress_id=$(app).$(version) -alias=$(app).$(version) -result_dir=$REGR_DIR/test"
```

There are many other arguments, and you can find detailed description of the SIMS tool in $DV_ROOT/doc/OpenSPARCCT1_DVGuide.pdf. Modify line 19, the app_args to point to the file list we just created:

```
-flist=$REGR_DIR/test/test.flist
```

Modify line 34, the run_args to set up the diagnose program path:
run_args="-vcs_run -asm_diag_path=$REGR_DIR/test"

Last, add the following lines at the top of run.sh to set up app and version variables:

app="test"
version="splyser"

This script runs the sims tool provided in OpenSPARC T1 project. Again, the manual of sims tool can be found OpenSPARC T1 design and verification guide. (or run sims -h)

Run your test benchmark:

./run.sh

Creating an OpenSplyser FPGA

To create a new OpenSplyser FPGA, first is to synthesize a sparc core with DySER. Navigate to:

$DV_ROOT/design/sys/iop/sparc/xst

Modify sparc.flist so that it contains all the splyser verilog files. You can check the files we listed in the previous section and modify the sparc.flist. However, to make this easier we provide a cumsum.flist for you:

cp cumsum.flist sparc.flist

To synthesis this entire FPGA netlist with this modified sparc core, run

rxil -device=XC5VLX110T sparc

The synthesized netlist will appear as

$DV_ROOT/design/sys/iop/sparc/xst/XC5VLX110T/sparc.ngc To use this file, navigate to:

$DV_ROOT/design/sys/edk

Copy the netlist to pcores/iop_fpga_v1_00_a/netlist/:

cp $DV_ROOT/design/sys/iop/sparc/xst/XC5VLX110T/sparc.ngc ./pcores/iop_fpga_v1_00_a/netlist/sparc.ngc

Now, we can use xps to load the edk project and synthesis the FPGA netlist(which includes sparc netlist and other modules). Run:

xps -nw

In Xilinx xps shell, run

XPS% xload xmp system.xmp

Create system.make by cleanup hardware files:

XPS% run hwclean

Now you can exit xps by:

XPS% exit

Loading the project will generate system.make, which will be used to generate the program, netlist, bitstream file and downloading bit file onto FPGA. Now, let's compile the microblaze
program:
make -f system.make program

Cleanup the hardware files:
make -f system.make hwclean

Build netlist and bitstream file:
make -f system.make netlist
make -f system.make bits

Download the bitstream:
make -f system.make download

Now you should have a FPGA which is configured as an OpenSlyser (with csumsum hardDySER). After following the instruction in Run a benchmark on FPGA, you should be able to boot Unbuntu and run the csumsum benchmark.

Now you have completed the tutorial! Congratulations!

Manual

SW toolchain
dysched

This is a front-end GUI that generates a configuration file that can be used later to create a hardDySER. To open the dysched GUI, run:

dysched <name of file>

The name of file is optional. Once it opens, you will be presented with a 4 x 4 DySER (4 Functional Units’s (FU) high, 4 FU’s wide). To change the size, change the numbers in the top left hand corner and click the plus logo to create a new DySER. The evaluation framework uses 4×4 or 5×5 DySER.

To create inputs to DySER, right click on the switches (diamonds) and choose “Add Top/Bottom Input” depending on where you want the input to be. Remember that switches can support two inputs. To route an input, double click on the input; your arrow should now turn the color of the input. Then, click on the desired location (either FU or switch). To stop routing, click on the whitespace outside of DySER (or right click). NOTE: All FU’s are 1 or 2 inputs (1 if using constant, 2 for using FU with 2 inputs). 3 inputs predecation FU is provided but not supported in opensplyser release.

To create a functional unit, right click on the large squares and choose the correct functional unit. You can clear the FU if you no longer need it by right clicking and choosing ‘Clear FU’. The ‘Edit Const’ command can be used to have a FU always add a constant to 1 inputs (i.e. an FADD that always adds +1).
To create an output, right click on the switch where the output should be and select 'Left/Right output' and the FU that has the output.

When you are finished, select 'Show Fu Inputs' at the top to be sure that the inputs are correctly hooked up (i.e. for FSub/FDiv, the order matters; 0 is the first input, 1 is the second). Also, select 'Show Port Numbers' to see what ports have been selected. Once satisfied, select “File→Save Config” and save your config. Place this file in the same folder as your source code.

**HW toolchain**

The Hardware toolchain contains: DySER release v1.0, HardDySER release, and Opensplyser xilinx EDK project (modified from Sun OpenSPARC T1 release).

**DySER release v1.0**

This is a release of full-function general DySER that can be configured in the runtime. More description will be added in the future release.

**HardDySER release**

This is a release of hardDySER that can be generated from script and dysched before synthesis. More description will be added in the future release.

**OpenSplyser release**

OpenSplyser project is based on Sun OpenSPARC T1 project. More description will be added in the future release.

**Release issues**

- Known issues:
  - Structural hazard between consecutive dyser load and dyser recv
  - Loads and dyser recv: the load in opensparc may flush the execution pipeline. dyser recv sends data to DySER at execution stage, and cannot be flushed. Currently we pad nop before dsend assembly in c code.
  - DySTORE does not work on FPGA: 3 benchmarks are tuned: fft (-DNO_DYSTORE), conv (TBD), stencil (TBD)

**FAQ**

projects/opensplyser-release.txt · Last modified: 2012/12/14 09:49 by aronasinanzoie