Build, Run, and Write RISC-V Programs

CS250 Tutorial 3 (Version 091110b)
September 11, 2010
Yunsup Lee

In this tutorial you will gain experience using the RISC-V toolchain to assemble and compile programs for the RISC-V v2 processor which you will implement in lab 2 and 3. You will also learn how to run the programs on the RISC-V ISA simulator and use the test macros to write your own test programs.

The RISC-V toolchain is a standard GNU cross compiler toolchain ported for RISC-V. You will be using `riscv-{gcc,as,ld}` to compile, assemble, and link your source files. Then you will run the compiled binary on the RISC-V ISA simulator to figure out whether your binary runs as intended. The RISC-V ISA simulator might report errors because of the RISC-V compiler generating instructions that are not defined in the RISC-V ISA. You need to carefully write C code to avoid these instructions. Please refer *RISC-V Processor Specification* for more information about the ISA. You will use the same binary for the test harness used in lab 2 and 3. For debugging purposes, you can also compile benchmarks written in C natively, and run the binary on the host x86 machine.

Figure 1 shows how everything fits together.

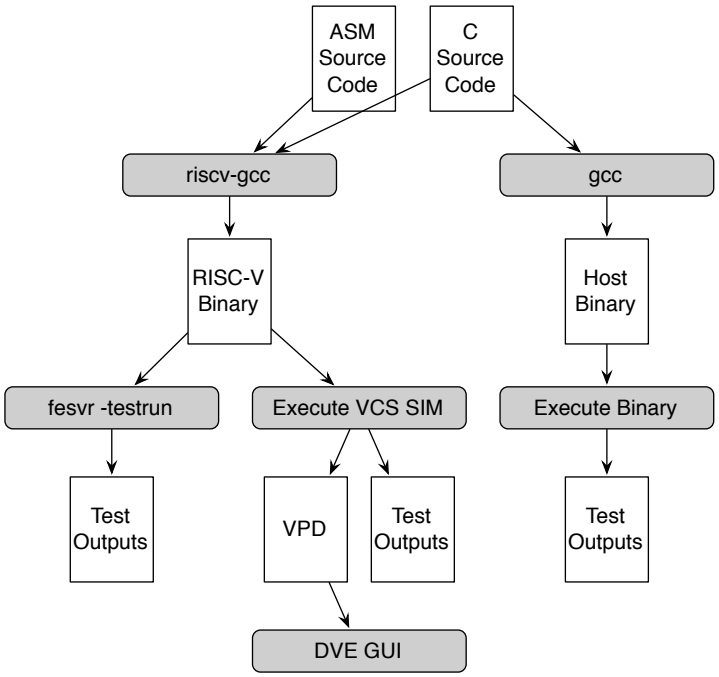

Figure 1: RISC-V Assembler and Compiler Toolchain

Getting started

You can follow along through the tutorial yourself by typing in the commands marked with a '%' symbol at the shell prompt. To cut and paste commands from this tutorial into your bash shell (and make sure bash ignores the '%' character) just use an alias to "undefine" the '%' character like this:

```
% alias %=""
```

All of the CS250 tutorials should be ran on an EECS Instructional machine. Please see the course website for more information on the computing resources available for CS250 students. Once you have logged into an EECS Instructional you will need to setup the CS250 toolflow with the following commands.

```
% source ~cs250/tools/cs250.bashrc
```

To begin this tutorial you will need to copy RISC-V test assembly source files and C benchmark source files from the course locker.

```
% mkdir tut3
% cd tut3
% TUTROOT=$PWD
% cp -R ~cs250/riscv-tests/ $TUTROOT
% cp -R ~cs250/riscv-bmarks/ $TUTROOT
```

Building RISC-V Test Assembly Programs

You will begin by assembling the `riscv-v1_simple.S` assembly test program. Take a look at the assembly in `riscv-tests/riscv-v1_simple.S` and notice that this test only has two instructions. You can use the following commands to generate a binary file, and an assembly dump file.

```
% cd $TUTROOT/riscv-tests
% riscv-gcc -O2 -G 0 -nostdlib -nostartfiles -T test.ld \
  riscv-v1_simple.S -o riscv-v1_simple
% riscv-objdump --disassemble-all --disassemble-zeroes \
  --section=.text --section=.data riscv-v1_simple > riscv-v1_simple.dump
```

Compare the original `riscv-v1_simple.S` file to the generated `riscv-v1_simple.dump` file. Using a combination of the assembly file and the objdump file you can get a good feel for what the test programs are supposed to do and what instructions are supposed to be executed.

You can use the makefile to automate the process of building RISC-V test assembly programs. The following commands will clean the build directory and then build the binary files.

```
% rm -f riscv-v1_simple riscv-v1_simple.dump
% make riscv-v1_simple.dump
```

Verify that the corresponding RISC-V binary and objdump file were generated.

The `riscv-v1_simple` test program is located locally in the `tut3/riscv-tests` directory. Globally installed RISC-V assembly test programs are located in `~cs250/install/riscv-tests` which you can use for lab 2 and 3 and projects. The following command will build all of the assembly tests.

```
% make
```

Running RISC-V Test Assembly Programs on the ISA Simulator

Now run your compiled RISC-V binary on the RISC-V ISA simulator.

```
% cd $TUTROOT/riscv-tests
% fesvr -testrun riscv-v1_simple
*** PASSED ***
```

In order to see more detailed trace of the run, you can use the interactive mode with a d option.

```
% cd $TUTROOT/riscv-tests
% fesvr -testrun -d riscv-v1_simple
:<enter>
core   0: 0x0000000000000000 (0xec100001) move  v0,v0
:<enter>
core   0: 0x0000000000000004 (0xfc185000) mtpcr v0,$cr16
:*** PASSED ***
% fesvr -testrun -d riscv-v1_simple
:while tohost 0 0
*** PASSED ***
% fesvr -testrun -d riscv-v1_simple
:<enter>
core   0: 0x0000000000000000 (0xec100001) move  v0,v0
:reg 0 0
0x0000000000000000
:mem 0
0xfc185000ec100001
:while tohost 0 0
:*** PASSED ***
```

You can see the cycle count, pc, instruction, register dump, memory dump, and the disassembled instruction. The first register of the instruction `mtpcr` tells you whether or not the test passed or not. Number 1 is used to indicate that the test passed, while the number bigger than 1 points you to the failed testcase number. You can also use the automated makefile to run through all the binaries.

```
% cd $TUTROOT/riscv-tests
% make run
  ...
  [ PASSED ] riscv-v1_addiu.out
  [ PASSED ] riscv-v1_bne.out
  [ PASSED ] riscv-v1_simple.out
  [ PASSED ] riscv-v1_lw.out
  [ PASSED ] riscv-v1_sw.out
  [ PASSED ] riscv-v2_addiu.out
  [ PASSED ] riscv-v2_addu.out
```

```
[ PASSED ] riscv-v2_andi.out
...
[ PASSED ] riscv-v2_xor.out
```

Writing RISC-V Test Assembly Programs

Take a look at `test_macro.h`. You can see helper macros which are used in various test assembly programs. Brief explanation of each macro follows.

- `TEST_RISCV_BEGIN` - This macro defines things that need to be included at the beginning of the test.
- `TEST_RISCV_END` - This macro defines things that need to be included at the end of the test.
- `TEST_CASE(testnum,testreg,correctval,code...)`- This macro defines a test case. Runs the `code`, and loads `testnum` to register `$x28`. Then checks if the value of `testreg` is `correctval`. If not, the program will jump to `fail` which is defined in `TEST_PASSFAIL`.
- `TEST_INSERT_NOPS_[0-10]`- This macro defines `nops`. The number in macro the indicates the number of `nops` to be inserted.
- `TEST_IMM_OP(testnum,inst,result,val1,imm)`- Basic test for immediate instructions. Loads `val1` to `$x1`, executes `inst $x3,$x1,imm` and checks if the `result` and `$x3` match.
- `TEST_IMM_SRC1_EQ_DEST(testnum,inst,result,val1,imm)`- Similar test to `TEST_IMM_OP`, though, executes `inst $x1,$x1,imm` and checks if the `result` and `$x1` match.
- `TEST_IMM_DEST_BYPASS(testnum,nop_cycles,inst,result,val1,imm)`- Destination register bypass test for immediate instructions. Loads `val1` to `$x1`, executes `inst $x3,$x1,imm` then reads `$x3` from the next instruction which is separated by `nop_cycles`.
- `TEST_IMM_SRC1_BYPASS(testnum,nop_cycles,inst,result,val1,imm)`- Source register bypass test for immediate instructions. Loads `val1` to `$x1`, waits for `nop_cycles`, then executes the instruction, and checks.
- `TEST_RR_OP(testnum,inst,result,val1,val2)`- Basic test for register register instructions. Loads `val1` to `$x1`, `val2` to `$x2`, executes `inst $x3,$x1,$x2` and checks if the `result` and `$x3` match.
- `TEST_RR_SRC1_EQ_DEST(testnum,inst,result,val1,val2)`- Similar test to `TEST_RR_OP`, though, executes `inst $x1,$x1,$x2` and checks if the `result` and `$x1` match.
- `TEST_RR_SRC2_EQ_DEST(testnum,inst,result,val1,val2)`- Similar test to `TEST_RR_OP`, though, executes `inst $x2,$x1,$x2` and checks if the `result` and `$x2` match.
- `TEST_RR_SRC12_EQ_DEST(testnum,inst,result,val1)`- Similar test to `TEST_RR_OP`, though, loads `val1` to `$x1`, executes `inst $x1,$x1,$x1` and checks if the `result` and `$x1` match.
- `TEST_RR_DEST_BYPASS(testnum,nop_cycles,inst,result,val1,val2)`- Destination register bypass test for register register instructions. Loads `val1` to `$x1`, `val2` to `$x2`, executes `inst $x3,$x1,$x2` then reads `$x3` from the next instruction which is separated by `nop_cycles`.
- `TEST_RR_SRC12_BYPASS(testnum,src1_nops,src2_nops,inst,result,val1,val2)`- Source register bypass test for register register instructions. Loads `val1` to `$x1`, waits `src1_nops`, loads `val2` to `$x2`, waits `src2_nops`, then executes instruction, and checks.
- `TEST_RR_SRC21_BYPASS(testnum,src_nops,src2_nops,inst,result,val1,val2)`- Similar to `TEST_RR_SRC12_BYPASS`, though, loads `val2` to `$x2` before loading `val1` to `$x1`.

- `TEST_LD_OP(testnum,inst,result,offset,base)` - Basic test for load instructions. Loads `base` to `$x1`, executes `inst $x3,offset($x1)` and checks if the `result` and `$x3` match.
- `TEST_ST_OP(testnum,load_inst,store_inst,result,offset,base)` - Basic test for store instructions. Loads `base` to `$x1`, `result` to `$x2`, executes `store_inst $x2,offset($x1)` and `load_inst $x3,offset($x1)` and checks if the `result` and `$x3` match.
- `TEST_LD_DEST_BYPASS(testnum,nop_cycles,inst,result,offset,base)` - Destination register bypass test for load instructions. Loads `base` to `$x1`, executes `inst $x3,offset($x1)`, then reads `$x3` from the next instruction which is separated by `nop_cycles`.
- `TEST_LD_SRC1_BYPASS(testnum,nop_cycles,inst,result,offset,base)` - Source register bypass test for load instructions. Loads `base` to `$x1`, waits `nop_cycles`, then executes instruction, and checks.
- `TEST_ST_SRC12_BYPASS(testnum,src1_nops,src2_nops,load_inst,store_inst,results,offset,base)` - Source register bypass test for store instructions. Loads `result` to `$x1`, waits for `src1_nops`, loads `base` to `$x2`, waits for `src2_nops`, executes the `store_instruction` and the `load_instruction`, then checks if the `result` and `$x3` match,
- `TEST_ST_SRC21_BYPASS(testnum,src1_nops,src2_nops,load_inst,store_inst,results,offset,base)` - Similar to `TEST_ST_SRC12_BYPASS`, though, loads `base` to `$x2` before loading `result` to `$x1`.
- `TEST_BR1_OP_TAKEN(testnum,inst,val1)` - Basic taken test for branch instructions with one input. Loads `val1` to `$x1`, then executes `inst $x1,pass`. If branch is not-taken the program will jump to `fail` which is defined in `TEST_PASSFAIL`.
- `TEST_BR1_OP_NOTTAKEN(testnum,inst,val1)` - Basic not-taken test for branch instructions with one input. Loads `val1` to `$x1`, then executes `inst $x1,fail`. If branch is taken the program will jump to `fail` which is defined in `TEST_PASSFAIL`.
- `TEST_BR1_SRC1_BYPASS(testnum,nop_cycles,inst,val1)` - Source register bypass test for branch instructions with one input. Loads `val1` to `$x1`, waits for `nop_cycles`, then executes branch instruction.
- `TEST_BR2_OP_TAKEN(testnum,inst,val1,val2)` - Basic taken test for branch instruction with two inputs. Loads `val1` to `$x1`, `val2` to `$x2`, then executes `inst $x1,$x2,pass`. If branch is not-taken the program will jump to `fail` which is defined in `TEST_PASSFAIL`.
- `TEST_BR2_OP_NOTTAKEN(testnum,inst,val1,val2)` - Basic not-taken test for branch instruction with two inputs. Loads `val1` to `$x1`, `val2` to `$x2`, then executes `inst $x1,$x2,fail`. If branch is taken the program will jump to `fail` which is defined in `TEST_PASSFAIL`.
- `TEST_BR2_SRC12_BYPASS(testnum,src1_nops,src2_nops,inst,val1,val2)` - Source register bypass test for branch instruction with two inputs. Loads `val1` to `$x1`, waits for `src1_nops`, loads `val2` to `$x2`, waits for `src2_nops`, executes branch instruction.
- `TEST_BR2_SRC21_BYPASS(testnum,src1_nops,src2_nops,inst,val1,val2)` - This macro is similar to `TEST_BR2_SRC12_BYPASS`, though, loads `val2` to `$x2` before loading `val1` to `$x1`.
- `TEST_JR_SRC1_BYPASS(testnum,nop_cycles,inst)` - Loads an address to `$x6`, waits for `nop_cycles`, then executes jump register instruction.
- `TEST_JALR_SRC1_BYPASS(testnum,nop_cycles,inst)` - Similar to `TEST_JR_SRC1_BYPASS`, though, executes jump and link register instruction.
- `TEST_PASSFAIL` - This macro defines what do to when success or fail. RISC-V v2 defines this macro using `mtpcr`.
- `SET_STATS(enable)` - This macro stimulates the test harness to turn logging on/off.

Open `$TUTROOT/riscv-test/riscv-v2_addu.S` to see how the macros are used.

```
#include "test_macros.h"

  TEST_RISCV_BEGIN
  SET_STATS(1)

  #-------------------------------------------------------------
  # Arithmetic tests
  #-------------------------------------------------------------

  TEST_RR_OP( 2,  addu, 0x00000000, 0x00000000, 0x00000000 );
  TEST_RR_OP( 3,  addu, 0x00000002, 0x00000001, 0x00000001 );
  TEST_RR_OP( 4,  addu, 0x0000000a, 0x00000003, 0x00000007 );

  TEST_RR_OP( 5,  addu, 0xffff8000, 0x00000000, 0xffff8000 );
  TEST_RR_OP( 6,  addu, 0x80000000, 0x80000000, 0x00000000 );
  TEST_RR_OP( 7,  addu, 0x7fff8000, 0x80000000, 0xffff8000 );

  TEST_RR_OP( 8,  addu, 0x00007fff, 0x00000000, 0x00007fff );
  TEST_RR_OP( 9,  addu, 0x7fffffff, 0x7fffffff, 0x00000000 );
  TEST_RR_OP( 10, addu, 0x80007ffe, 0x7fffffff, 0x00007fff );

  TEST_RR_OP( 11, addu, 0x80007fff, 0x80000000, 0x00007fff );
  TEST_RR_OP( 12, addu, 0x7fff7fff, 0x7fffffff, 0xffff8000 );

  TEST_RR_OP( 13, addu, 0xffffffff, 0x00000000, 0xffffffff );
  TEST_RR_OP( 14, addu, 0x00000000, 0xffffffff, 0x00000001 );
  TEST_RR_OP( 15, addu, 0xfffffffe, 0xffffffff, 0xffffffff );

  #-------------------------------------------------------------
  # Source/Destination tests
  #-------------------------------------------------------------

  TEST_RR_SRC1_EQ_DEST( 16, addu, 24, 13, 11 );
  TEST_RR_SRC2_EQ_DEST( 17, addu, 25, 14, 11 );
  TEST_RR_SRC12_EQ_DEST( 18, addu, 26, 13 );

  #-------------------------------------------------------------
  # Bypassing tests
  #-------------------------------------------------------------

  TEST_RR_DEST_BYPASS( 19, 0, addu, 24, 13, 11 );
  TEST_RR_DEST_BYPASS( 20, 1, addu, 25, 14, 11 );
  TEST_RR_DEST_BYPASS( 21, 2, addu, 26, 15, 11 );

  TEST_RR_SRC12_BYPASS( 22, 0, 0, addu, 24, 13, 11 );
  TEST_RR_SRC12_BYPASS( 23, 0, 1, addu, 25, 14, 11 );
  TEST_RR_SRC12_BYPASS( 24, 0, 2, addu, 26, 15, 11 );
  TEST_RR_SRC12_BYPASS( 25, 1, 0, addu, 24, 13, 11 );
```

```
TEST_RR_SRC12_BYPASS( 26, 1, 1, addu, 25, 14, 11 );
TEST_RR_SRC12_BYPASS( 27, 2, 0, addu, 26, 15, 11 );

TEST_RR_SRC21_BYPASS( 28, 0, 0, addu, 24, 13, 11 );
TEST_RR_SRC21_BYPASS( 29, 0, 1, addu, 25, 14, 11 );
TEST_RR_SRC21_BYPASS( 30, 0, 2, addu, 26, 15, 11 );
TEST_RR_SRC21_BYPASS( 31, 1, 0, addu, 24, 13, 11 );
TEST_RR_SRC21_BYPASS( 32, 1, 1, addu, 25, 14, 11 );
TEST_RR_SRC21_BYPASS( 33, 2, 0, addu, 26, 15, 11 );

SET_STATS(0)
TEST_PASSFAIL
TEST_RISCV_END
```

Building RISC-V C Benchmark Programs

Go ahead and build the RISC-V binary and the corresponding object dump file for the quicksort benchmark.

```
% cd $TUTROOT/riscv-bmarks
% riscv-gcc -mabi=32 -O2 -G 0 -nostdlib -nostartfiles -DPREALLOCATE=1 -DHOST_DEBUG=0 \
  -c -I./qsort qsort/qsort_main.c -o qsort_main.o
% riscv-gcc -mabi=32 -T ./stuff/test.ld  qsort_main.o  -o qsort.riscv
% riscv-objdump --disassemble-all --disassemble-zeroes \
  --section=.text --section=.data qsort.riscv > qsort.riscv.dump
```

Search for symbol `sort` in `qsort.riscv.dump`. You can see how the compiler transformed the C `sort` function into instructions.

For debugging purposes, you might want to compile your code natively. There is no reason why you can't do that because the benchmark is written in C. However, there are some RISC-V specific instructions embedded in the benchmark, for example, `mtpcr` instruction would not run on an x86 machine. Take a close look at `qsort_main.c`. RISC-V specific things are already wrapped by `HOST_DEBUG`. You just need to define `HOST_DEBUG` to 1 when compiling.

```
% gcc -DPREALLOCATE=0 -DHOST_DEBUG=1 ./qsort/qsort_main.c -o qsort.host
```

You can use the makefile to automate build process for RISC-V binaries. There are globally installed RISC-V C benchmarks located in `~cs250/install/riscv-bmarks` which are already compiled for lab 2 and 3.

```
% cd $TUTROOT/riscv-bmarks
% make
```

Running RISC-V C Benchmark Programs on the ISA Simulator

Now run the compiled benchmarks on the RISC-V ISA simulator. Go ahead an try the automated run as well.

```
% cd $TUTROOT/riscv-bmarks
% fesvr -testrun qsort.riscv
*** PASSED ***
% fesvr -testrun -d qsort.riscv
:<enter>
core   0: 0x0000000000000000 (0xe3d00020) lui sp,0x20
:<enter>
core   0: 0x0000000000000004 (0xc80001c8) jal 0x0000000000000390
:<enter>
core   0: 0x0000000000000390 (0xe2400000) lui a1,0x0
:<enter>
core   0: 0x0000000000000394 (0xedde8fe8) addiw sp,sp,-24
:<enter>
core   0: 0x0000000000000398 (0xec4204c0) addiw a1,a1,1216
:<enter>
core   0: 0x000000000000039c (0xe83040fa) li  a0,0xfa
:<enter>
core   0: 0x00000000000003a0 (0xf3fea014) sw  ra,20(sp)
:<enter>
core   0: 0x00000000000003a4 (0xc80000b4) jal 0x0000000000000168
:<enter>
core   0: 0x0000000000000168 (0xedde8f38) addiw sp,sp,-200
:while tohost 0 0
:*** PASSED ***
% cd $TUTROOT/riscv-bmarks
% make run-riscv
...
[ PASSED ] median.riscv.out
[ PASSED ] qsort.riscv.out
[ PASSED ] towers.riscv.out
[ PASSED ] vvadd.riscv.out
```

You can also run the benchmark which is compiled natively. The native run is automated as well.

```
% cd $TUTROOT/riscv-bmarks
% ./qsort.host
..
979  979  981  985  985  989  989  997  997  998
*** PASSED ***
% cd $TUTROOT/riscv-bmarks
% make run-host
...
[ PASSED ] median.host.out
[ PASSED ] qsort.host.out
[ PASSED ] towers.host.out
[ PASSED ] vvadd.host.out
```

Writing RISC-V C Benchmark Programs

Writing benchmark programs for RISC-V is similar with writing plain C programs. However when you are coding, keep in mind that you also want to test the code natively. Try to guard your RISC-V specific parts with a macro named `HOST_DEBUG`. Another thing to keep in mind is that since you are running the compiled code on the RISC-V v2 processor, some instructions generated by the compiler might not be in the RISC-V v2 ISA. Keep your memory accesses aligned by 4 bytes, and avoid arithmetic that is not defined in the ISA. Try to write your own function that emulates the functionality. For example, write a multiply function which only uses adds and shifts. Then call the multiply function whenever you need to do a multiplication. Before you test your program on the processor, always try to verify the compiled binary against the ISA simulator first!

Review

The following sequence of command will setup the RISC-V toolchain, copy the source files, build the binaries, run all tests, and report the results.

```
% source ~cs250/tools/cs250.bashrc
% mkdir tut3
% cd tut3
% TUTROOT=$PWD
% cp -R ~cs250/riscv-tests/ $TUTROOT
% cp -R ~cs250/riscv-bmarks/ $TUTROOT
% cd $TUTROOT/riscv-tests
% make run
% cd $TUTROOT/riscv-bmarks
% make run-host
% make run-riscv
```

Acknowledgements

Many people have contributed to versions of this tutorial over the years. The tutorial was originally developed for CS250 VLSI Systems Design course at University of California at Berkeley by Yunsup Lee. Contributors include: Krste Asanović, Christopher Batten, John Lazzaro, and John Wawrzynek. Versions of this tutorial have been used in the following courses:

- CS250 VLSI Systems Design (2009-2010) - University of California at Berkeley
- CSE291 Manycore System Design (2009) - University of California at San Diego