Build, Run, and Write SMIPS Programs

CS250 Tutorial 3 (Version 092509a)
September 25, 2009
Yunsup Lee

In this tutorial you will gain experience using the SMIPS toolchain to assemble and compile programs for the SMIPSv2 processor which you will implement in lab 2 and 3. You will also learn how to run the programs on the SMIPSv2 ISA simulator and use the test macros to write your own test programs.

The SMIPS toolchain is a standard GNU cross compiler toolchain ported for SMIPS. You will be using `smips-{gcc,as,ld}` to compile, assemble, and link your source files. Then you will run the compiled binary on the SMIPSv2 ISA simulator to figure out whether or not your binary runs as intended. The SMIPSv2 ISA simulator might report errors because of the SMIPS compiler generating instructions that are not defined in the SMIPSv2 ISA. You need to carefully write C code to avoid these instructions. Please refer *SMIPS Processor Specification* for more information about the ISA.

Unfortunately, your SMIPS Verilog test harness used in lab 2 and 3 cannot read SMIPS binaries directly, so you must use additional tools to convert the SMIPS binary into a usable format. You will also use the `smips-objdump` program which takes the SMIPS binary as input and produces a textual listing of the instructions and data contained in the binary. Then you will run the `objdump2vmh.pl` Perl script to convert this text objdump into a Verilog Memory Hex (VMH) file which the Verilog test harness can read into its magic memory.

Figure 1 shows how everything fits together.

Getting started

Before using the SMIPS toolchain you must run the course setup script with the following command.

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

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

```
% mkdir tut3
% cd tut3
% cp -R ~cs250/smips-tests/ .
% cp -R ~cs250/smips-bmarks/ .
```

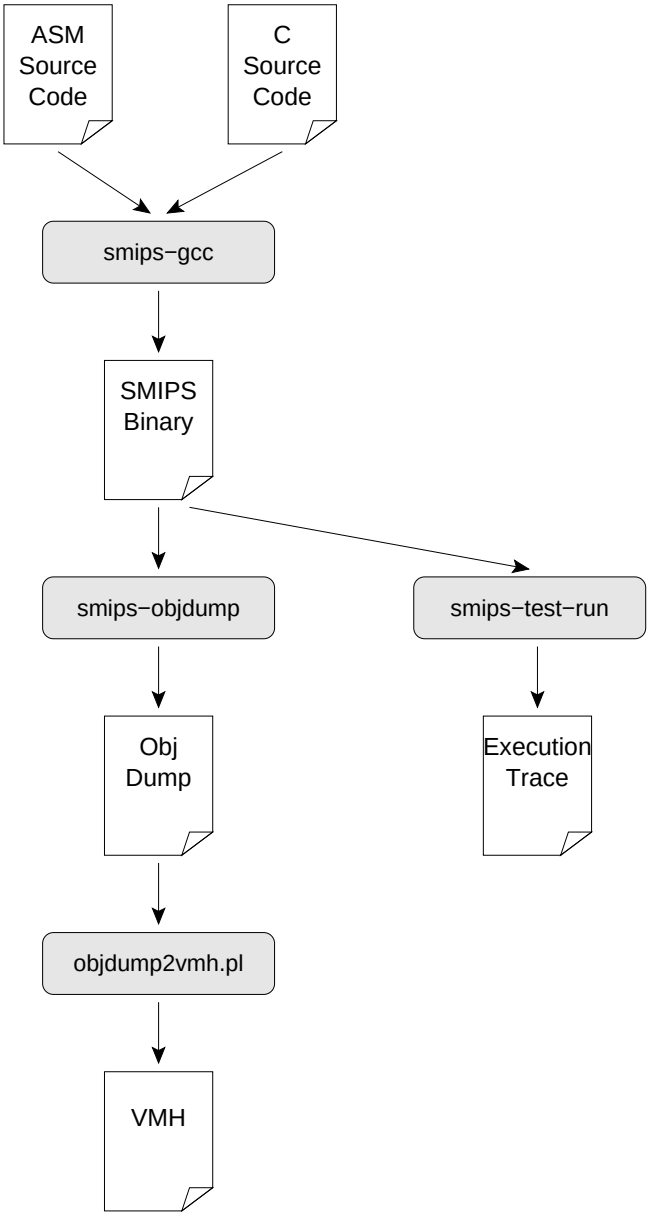

Figure 1: SMIPS Assembler and Compiler Toolchain

Building SMIPS Test Assembly Programs

You will begin by assembling the `smipsv1_simple.S` assembly test program. Take a look at the assembly in `smips-tests/smipsv1_simple.S` and notice that this test only has two instructions. You can use the following commands to generate a binary file, and a VMH file from the assembly file.

```
% cd smips-tests
% pwd
tut3/smips-tests
% smips-gcc -O2 -G 0 -nostdlib -nostartfiles smipsv1_simple.S -o smipsv1_simple.S.bin
% smips-objdump --disassemble-all --disassemble-zeroes \
  --section=.text --section=.data smipsv1_simple.S.bin > smipsv1_simple.S.dump
% objdump2vmh.pl smipsv1_simple.S.dump smipsv1_simple.S.vmh
```

Compare the original `smipsv1_simple.S` file to the generated `smipsv1_simple.S.dump`. 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 SMIPS test assembly programs. The following commands will clean the build directory and then build the desired `smipsv1_simple.S.vmh` file as well as all required intermediate files.

```
% rm -f smipsv1_simple.*
% make smipsv1_simple.S.vmh
```

Verify that the corresponding SMIPS binary and objdump file were generated.

The `smipsv1_simple.S` test program was located locally in the `tut3/smips-tests` directory. There are globally installed SMIPS assembly test programs located in `~cs250/install/smips-tests` which you can use for lab 2 and 3 and projects. The following command will build all of the assembly tests.

```
% make
```

Running SMIPS Test Assembly Programs on the ISA Simulator

Now run your compiled SMIPS binary on the SMIPSv2 ISA simulator.

```
% pwd
tut3/smips-tests
% smipsv2-test-run smipsv1_simple.S.bin
CYC: 0 [pc=00001000] [inst=24020001] R[r 0=00000000] W[r2=00000001] addiu $v0,$zero,1
CYC: 1 [pc=00001004] [inst=4082a800] R[r 2=00000001] mtc0 $v0,$s5
*** PASSED ***
```

You can see the cycle count, pc, instruction, register accesses, and the disassembled instruction. The first register of the instruction `mtc0` 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.

```
% make run
  ...
  [ PASSED ] smipsv1_addiu.S.out
  [ PASSED ] smipsv1_bne.S.out
  [ PASSED ] smipsv1_simple.S.out
  [ PASSED ] smipsv1_lw.S.out
  [ PASSED ] smipsv1_sw.S.out
  [ PASSED ] smipsv2_addiu.S.out
  [ PASSED ] smipsv2_addu.S.out
  [ PASSED ] smipsv2_andi.S.out
  ...
  [ PASSED ] smipsv2_xor.S.out
```

Writing SMIPS 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_CASE(testnum,testreg,correctval,code...)` - This macro defines a test case. Runs the `code`, and loads `testnum` to register `$30`. 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 `$2`, executes `inst $4,$2,imm` and checks if the `result` and `$4` match.
- `TEST_IMM_SRC1_EQ_DEST(testnum,inst,result,val1,imm)` - Similar test to `TEST_IMM_OP`, though, executes `inst $2,$2,imm` and checks if the `result` and `$2` match.
- `TEST_IMM_DEST_BYPASS(testnum,nop_cycles,inst,result,val1,imm)` - Destination register bypass test for immediate instructions. Loads `val1` to `$2`, executes `inst $4,$2,imm` then reads `$4` 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 `$2`, 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 `$2`, `val2` to `$3`, executes `inst $4,$2,$3` and checks if the `result` and `$4` match.
- `TEST_RR_SRC1_EQ_DEST(testnum,inst,result,val1,val2)` - Similar test to `TEST_RR_OP`, though, executes `inst $2,$2,$3` and checks if the `result` and `$2` match.
- `TEST_RR_SRC2_EQ_DEST(testnum,inst,result,val1,val2)` - Similar test to `TEST_RR_OP`, though, executes `inst $3,$2,$3` and checks if the `result` and `$3` match.
- `TEST_RR_SRC12_EQ_DEST(testnum,inst,result,val1)` - Similar test to `TEST_RR_OP`, though, loads `val1` to `$2`, executes `inst $2,$2,$2` and checks if the `result` and `$2` match.
- `TEST_RR_DEST_BYPASS(testnum,nop_cycles,inst,result,val1,val2)` - Destination register bypass test for register register instructions. Loads `val1` to `$2`, `val2` to `$3`, executes `inst $4,$2,$3` then reads `$4` 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 `$2`, waits `src1_nops`, loads `val2` to `$3`, 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 `$3` before loading `val1` to `$2`.
- `TEST_LD_OP(testnum,inst,result,offset,base)` - Basic test for load instructions. Loads `base` to `$2`, executes `inst $4,offset($2)` and checks if the `result` and `$4` match.
- `TEST_ST_OP(testnum,load_inst,store_inst,result,offset,base)` - Basic test for store instructions. Loads `base` to `$2`, `result` to `$3`, executes `store_inst $3,offset($2)` and `load_inst $4,offset($2)` and checks if the `result` and `$4` match.
- `TEST_LD_DEST_BYPASS(testnum,nop_cycles,inst,result,offset,base)` - Destination register bypass test for load instructions. Loads `base` to `$2`, executes `inst $4,offset($2)`, then reads `$4` 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 `$2`, 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 `$2`, waits for `src1_nops`, loads `base` to `$3`, waits for `src2_nops`, executes the `store_instruction` and the `load_instruction`, then checks if the `result` and `$4` 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 `$3` before loading `result` to `$2`.
- `TEST_BR1_OP_TAKEN(testnum,inst,val1)` - Basic taken test for branch instructions with one input. Loads `val1` to `$2`, then executes `inst $2,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 `$2`, then executes `inst $2,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 `$2`, 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 `$2`, `val2` to `$3`, then executes `inst $2,$3,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 `$2`, `val2` to `$3`, then executes `inst $2,$3,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 `$2`, waits for `src1_nops`, loads `val2` to `$3`, 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 `$3` before loading `val1` to `$2`.
- `TEST_JR_SRC1_BYPASS(testnum,nop_cycles,inst)` - Loads an address to `$7`, 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 define what do to when success or fail. SMIPSv2 defines this macro using `mtc0`.

Open `smips-test/smipsv2_addu.S` to see how the macros are used.

```
#-------------------------------------------------------------
# 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 );

TEST_PASSFAIL
```

Building SMIPS C Benchmark Programs

Go ahead and build the SMIPS binary and the corresponding VMH file for the quicksort benchmark.

```
% cd ../smips-bmarks/qsort
% pwd
tut3/smips-bmarks/qsort
% smips-gcc -O2 -G 0 -nostdlib -nostartfiles -DPREALLOCATE=1 -DHOST_DEBUG=0 \
  -c -I. qsort_main.c -o qsort_main.o
% smips-ld ~cs250/tools/smips-xcc/mips-elf/lib/crt1.o qsort_main.o \
  -o qsort.smips.bin
% smips-objdump --disassemble-all --disassemble-zeroes \
  --section=.text --section=.data qsort.smips.bin > qsort.smips.dump
% objdump2vmh.pl qsort.smips.dump qsort.smips.vmh
```

Search for symbol `sort` in `qsort.smips.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 SMIPS specific instructions embedded in the benchmark, for example, `mtc0` instruction would not run on an x86 machine. Take a close look at `qsort_main.c`. SMIPS specific stuff are already wrapped by `HOST_DEBUG`. You just need to define `HOST_DEBUG` to 1 when compiling.

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

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

```
% cd ..
% pwd
tut3/smips-bmarks
% make
```

Running SMIPS C Benchmark Programs on the ISA Simulator

Now run the compiled benchmarks on the SMIPSv2 ISA simulator. Go ahead an try the automated run as well.

```
% cd qsort
% pwd
tut3/smips-bmarks/qsort
% smipsv2-test-run qsort.smips.bin
...
CYC: 19278 [pc=00001104] [inst=ac470000] R[r 2=00001664] R[r 7=00000320] sw $a3,0($v0)
CYC: 19279 [pc=00001108] [inst=25080004] R[r 8=00001664] W[r 8=00001668] addiu $t0,$t0,4
CYC: 19280 [pc=0000110c] [inst=254a0004] R[r10=00001660] W[r10=00001664] addiu $t2,$t2,4
CYC: 19281 [pc=00001110] [inst=1060ffeb] R[r 3=00000001] R[r 0=00000000] beq $v1,$zero,-21
CYC: 19282 [pc=00001114] [inst=11e00065] R[r15=00000000] R[r 0=00000000] beq $t7,$zero,101
CYC: 19283 [pc=000012ac] [inst=8fb000d4] R[r29=0001ff18] W[r16=00c47a40] lw $s0,212($sp)
CYC: 19284 [pc=000012b0] [inst=27bd00d8] R[r29=0001ff18] W[r29=0001fff0] addiu $sp,$sp,216
CYC: 19285 [pc=000012b4] [inst=03e00008] R[r31=000012d4] jr $ra
CYC: 19286 [pc=000012d4] [inst=24020001] R[r 0=00000000] W[r 2=00000001] addiu $v0,$zero,1
CYC: 19287 [pc=000012d8] [inst=40825000] R[r 2=00000001] mtc0 $v0,$t2
*** PASSED ***
% cd ..
% make run-smips
...
[ PASSED ] median.smips.out
[ PASSED ] qsort.smips.out
[ PASSED ] towers.smips.out
[ PASSED ] vvadd.smips.out
```

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

```
% cd qsort
% pwd
tut3/smips-bmarks/qsort
% ./qsort.host.bin
..
979  979  981  985  985  989  989  997  997  998
*** PASSED ***
% cd ..
% make run-host
...
[ PASSED ] median.host.out
[ PASSED ] qsort.host.out
[ PASSED ] towers.host.out
[ PASSED ] vvadd.host.out
```

Writing SMIPS C Benchmark Programs

Writing benchmark programs for SMIPS 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 SMIPS 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 SMIPSv2 processor, some instructions generated by the compiler might not be in the SMIPSv2 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 SMIPS toolchain, copy the source files, build the binaries, run all tests, and report the results.

```
% source ~cs250/tools/cs250.bashrc
% mkdir tut3
% cd tut3
% cp -R ~cs250/smips-tests/ .
% cp -R ~cs250/smips-bmarks/ .
% cd smips-tests
% make run
% cd ../smips-bmarks
% make run-host
% make run-smips
```

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) - University of California at Berkeley