Tutorial #2
Verilog Simulation Toolflow

% vcs mips.v
% ./simv
% vcs -RPP &
A Hodgepodge Of Information

- CVS source management system
- Browsing a CVS repository with viewcvs
- Makefile verilog build system
- Writing SMIPS assembly
- Using the SMIPS disassembler
- Using trace output instead of waveforms
Concurrent Versions System

- A central repository contains all verilog code as well as information on who changed what and when
- Users **checkout** a copy of the verilog code, edit it, and then **commit** their modified version
- Users can see what has changed to help track down bugs and this allows multiple users to work on the same verilog code at the same time
- Our repository is at /mit/6.884/cvsroot, but you should never access the repository directly. Instead use CVS commands of the following form:
  
  ```
  % cvs <commandname>
  % cvs help
  ```
6.884 CVS Repository

There are three primary types of top-level directories in the repository

- Examples (everyone has access)
- Individual directories (only you have read/write)
- Project directories (everyone has access)

To checkout the examples and try them out use

% cvs checkout examples

To checkout your individual directory use

% cvs checkout <athena-username>
CVS Basics

Common CVS commands

- `cvs checkout pname`  Checkout a working copy
- `cvs update pname`    Update working dir vs. repos
- `cvs commit [filelist]`  Commit your changes
- `cvs add [filelist]`    Add new files/dirs to repos
- `cvs diff`             See how working copy differs

Set the `$CVSEDITOR` environment variable to change which editor is used when writing log messages
6.884 CVS Repository

Use the following commands to checkout the mips2stage test harness, test programs, etc and then put them into your own cvs directory

% cvs checkout 2005-spring/cbatten
% cd 2005-spring/cbatten
% cvs export -r HEAD examples/mips2stage
% mv examples/mips2stage .
% rm -rf examples
% find mips2stage | xargs cvs add
% <start to make your additions>
% cvs add [new files]
% cvs update
% cvs commit
Using CVS Tags

Symbolic tags are a way to mark all the files in your project at a certain point in the project development. You can then later checkout the whole project exactly as it existed previously with the tag.
CVS – Multiple Users

User A

F.1

User B

F.1

checkout

checkout
CVS - Multiple Users

User A  F.1

User B  F.2

commit
CVS – Multiple Users

User A

User B

F.2

F.3

commit

Conflict!
CVS - Multiple Users

Update

Merges F.2 changes and denotes conflicts with <<<< >>>>>
CVS – Multiple Users

commit

User A
F.4

User B
F.2

F.4
Use Viewcvs for Repository Browsing

Viewcvs is a convenient tool for browsing the cvs repository through a web interface. Use the start-viewcvs command to start the viewcvs web server then point your (local) browser to http://localhost:6884
Why not use Makefiles to start out with?
- Dependency tracking is less necessary
- Difficult to implement some operations

Why are we changing now?
- Makefiles are more familiar to many of you
- Dependency tracking will become more useful with the addition of test binary generation and Bluespec compilation
Using the Makefiles

Create a build directory, then use configure.pl to create a Makefile, and then use various make targets to create various generated products. Unlike mkasic.pl we will be placing much more generated product directly in the build directory.

```text
% cvs checkout examples
% cd examples/gcd
% mkdir build-gcd-rtl
% cd build-gcd-rtl
% ../configure.pl ../config/gcd_rtl.mk
% make simv
% ./simv
% vcs -RPP &
```

```text
gcd/
    Makefile.in
    configure.pl
verilog/
    gcd_behavioral.v
    gcd_rtl.v
config/
    gcd_behavioral.mk
    gcd_rtl.mk
tests/
    gcd-test.dat
```
Modifying the config/*.mk Files

Analagous to mkasic.pl .cfg files except these use standard make constructs

```
# Configuration makefile module

verilog_src_dir = verilog
verlog_toplevel = gcd_test
verilog_srcs = gcd_rtl.v

# vcs specific options
vcs_extra_options = -PP

# unit test options
utests_dir = tests
utests = gcd-test.dat
```
Using the Makefiles with mips2stage

% cvs checkout 2005-spring/cbatten/mips2stage
% cd spring-2005/cbatten/mips2stage
% mkdir build
% cd build
% ../configure.pl ../config/mips2stage.mk
% make simv
% make self_test.bin
% make self_test.vmh
% ./simv +exe=self_test.vmh
% make run-tests

You can just use make run-tests and the dependency tracking will cause the simulator and the tests to be built before running the tests.
Lab Checkoff Procedure

We will be using the following procedure to checkoff lab 1 so please make sure these steps work (on a clean build!)

If the lab1-finale tag does not exist then we will just checkout the most recent version

```bash
% cvs checkout –r lab1-final \ 2005-spring/cbatten/mips2stage
% cd 2005-spring/cbatten/mips2stage
% mkdir build
% cd build
% ../configure.pl ../config/mips2stage.mk
% make simv
% make run-tests
% <run simv with some other tests>
```
Our assembler takes as input SMIPS assembly code with various test macros in the following format:

```
#include <smipstest.h>
TEST_SMIPS
TEST_CODEBEGIN
    addiu r2, r0, 1
    mtc0 r2, r21

loop: beq r0, r0, loop

EXIT
```

- `#include <smipstest.h>`: Includes assembler macros.
- `TEST_SMIPS`: Test assembler macros.
- `TEST_CODEBEGIN`: Begins the test code block.
- `addiu r2, r0, 1`, `mtc0 r2, r21`: Your test code.
- `loop: beq r0, r0, loop`: Test code.
- `EXIT`: Ends the test code block.
**Writing SMIPS Assembly**

You can find the assembly format for each instruction in the SMIPS processor spec next to the instruction tables.

Use `self_test.S` as an example.
Writing SMIPS Assembly

Our assembler accepts three types of register specifier formats

```
addiu r2, r0, 1
mtc0 r2, r21
loop: beq r0, r0, loop

addiu $2, $0, 1
mtc0 $2, $21
loop: beq $0, $0, loop

addiu t0, zero, 1
mtc0 t0, $21
loop: beq zero, zero, loop

} Traditional names for MIPS calling convention
```
Writing SMIPS Assembly (at)

The assembler reserves r1 for macro expansion and will complain if you try and use it without explicitly overriding the assembler.

```c
#include <smipstest.h>
TEST_SMIPS

TEST_CODEBEGIN

.set noat
.addiu r1, zero, 1
.mtc0 r1, r21

loop: .beq zero, zero, loop
.set at

TEST_CODEEND
```

Assembler directive which tells the assembler not to use r1 (at = assembler temporary)
Writing SMIPS Assembly (reorder)

By default the branch delay slot is **not visible**! The assembler handles filling the branch delay slot unless you explicitly direct it not to

```c
#include <smipstest.h>
TEST_SMIPS

TEST_CODEBEGIN

.set noreorder
addiu r2, zero, 1
mtc0 r2, r21

loop: beq zero, zero, loop
nop

.set reorder

TEST_CODEEND
```

Assembler directive which tells the assembler not to reorder instructions - programmer is responsible for filling in the delay slot
Use smips-objdump for Disassembly

Eventually the disassembled instructions will show up in the vmh file, but it is still useful to directly disassemble the binary

```
#include <smipstest.h>
TEST_SMIPS

TEST_CODEBEGIN
  addiu r2, zero, 1
  mtc0 r2, r21
loop: beq zero, zero, loop
TEST_CODEEND

% make simple_test.bin
% smips-objdump -D simple_test.bin
```
Examining Assembler Output

#include <smipstest.h>
TEST_SMIPS

TEST_CODEBEGIN
.set noat
addiu r1, zero, 1
mtc0 r1, r2

loop: beq zero, zero, loop
.set at
TEST_CODEEND

...
Examining Assembler Output

#include <smipstest.h>
TEST_SMIPS

TEST_CODEBEGIN
.set noat
addiu r1, zero, 1
mtc0 r1, r21
loop: beq zero, zero, loop
.set at
TEST_CODEEND

00001000 <__testresets>:
  1000: 40806800 mtc0 $zero,$13
  1004: 00000000 nop
  1008: 40805800 mtc0 $zero,$11
  100c: 3c1a0000 lui $k0,0x0
  1010: 8f5a1534 lw $k0,5428($k0)
  1014: 409a6000 mtc0 $k0,$12
  1018: 3c1a0000 lui $k0,0x0
  101c: 275a1400 addiu $k0,$k0,512
  1020: 03400008 jr $k0
  1024: 42000010 rfe

00001100 <__testexcep>:
  1100: 401a6800 mfc0 $k0,$13
  1104: 00000000 nop
  <snip>

00001400 <__testcode>:
  1400: 24010001 li $at,1
  1404: 4081a800 mtc0 $at,$21
  <loop>:
  1408: b 1408 <loop>
  <snip>
  1410: 3c080000 lui $t0,0x0
  1420: 8d081530 lw $t0,5424($t0)
  1424: 3c01dead lui $at,0xdead
  1428: 3421beef ori $at,$at,0xbeef
  142c: 11010003 beq $t0,$at,143c <loop+34>
  ...  
  1438: 0000000d break
  143c: 24080001 li $t0,1
  1440: 4088a800 mtc0 $t0,$21
  1444: b 1444 <loop+3c>
Trace Output Instead of Waveforms

It is sometimes very useful to use $display calls from the test harness to create cycle-by-cycle trace output instead of pouring through waveforms

```verilog
integer cycle = 0;
always @(posedge clk)
begin
  #2;
  $display("CYC:%2d [pc=%x] [ireg=%x] [rd1=%x] [rd2=%x] [wd=%x] tohost=%d",
           cycle, mips.fetch_unit.pc, mips.exec_unit.ir,
           mips.exec_unit.rd1, mips.exec_unit.rd2, mips.exec_unit.wd, tohost);
  cycle = cycle + 1;
end
```

| CYC: 0 [pc=00001000] [ireg=xxxxxxxx] [rd1=xxxxxxxx] [rd2=xxxxxxxx] [wd=00001004] tohost= 0 |
| CYC: 1 [pc=00001004] [ireg=08000500] [rd1=00000000] [rd2=00000000] [wd=00001008] tohost= 0 |
| CYC: 2 [pc=00001400] [ireg=00000000] [rd1=00000000] [rd2=00000000] [wd=00000000] tohost= 0 |
| CYC: 3 [pc=00001404] [ireg=24010001] [rd1=00000000] [rd2=xxxxxxxx] [wd=00000001] tohost= 0 |
| CYC: 4 [pc=00001408] [ireg=4081a800] [rd1=xxxxxxxx] [rd2=00000001] [wd=0000140c] tohost= 0 |
| CYC: 5 [pc=0000140c] [ireg=1000ffff] [rd1=00000000] [rd2=00000000] [wd=00001410] tohost= 1 |
| CYC: 6 [pc=00001408] [ireg=00000000] [rd1=00000000] [rd2=00000000] [wd=00000000] tohost= 1 |

6.884 – Spring 2005 02/16/05
Trace Output Instead of Waveforms

It is sometimes very useful to use $display calls from the test harness to create cycle-by-cycle trace output instead of pouring through waveforms

```
#include <smipstest.h>
TEST_SMIPS
   TEST_CODEBEGIN
      .set noat
      addiu r1, zero, 1
      mtc0 r1, r21
   loop: beq zero, zero, loop
      .set at
   TEST_CODEEND
```

<table>
<thead>
<tr>
<th>CYC:</th>
<th>pc</th>
<th>ireg</th>
<th>rd1</th>
<th>rd2</th>
<th>wd</th>
<th>tohost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00001000</td>
<td>xxxxxxxx</td>
<td>xxxxxxxx</td>
<td>xxxxxxxx</td>
<td>00001004</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>00001004</td>
<td>08000500</td>
<td>00000000</td>
<td>00000000</td>
<td>00001008</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>00001400</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>00001404</td>
<td>24010001</td>
<td>00000000</td>
<td>xxxxxxxx</td>
<td>00000001</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>00001408</td>
<td>4081a800</td>
<td>xxxxxxxx</td>
<td>00000001</td>
<td>0000140c</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0000140c</td>
<td>1000ffff</td>
<td>00000000</td>
<td>00000000</td>
<td>00001410</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>00001408</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>1</td>
</tr>
</tbody>
</table>
Final Notes

Lab Assignment 1
- Don't worry about cvs/make for now since I will be finishing setting this up this afternoon
- Please write at least one other small test (it took me a long time to get the assembly toolchain working!)
- You must verify that the checkoff procedure works
- Your verilog will be checked out automatically Friday at 1pm

Lab Assignment 2
- Synthesize your two stage mips processor
- Assigned on Friday and due the following Friday, Feb 25
- I will work on a synthesis tutorial over the weekend and email it out on Monday