Zoocad Consulting AB

CHIPOTLE
C-Program Based Test Language Environment

Version 1.0

Revision History

Date	Version	Description	Author
2013-08-27	1.0	Initial document	Sven-Åke Andersson

Table of Contents

Writing C programs to verify a SoC RTL design

1. Introduction

CHIPOTLE is an ASIC/FPGA verification environment for SoC designs. It is also an extension to the C programming language that will enable us to write C-programs to verify SoC designs for both RTL and netlist implementations.

1.1 Abstract

Every SoC design implemented in an ASIC or FPGA probably has one or more embedded processors. These processors are running programs that are mostly written in C or C++ which controls the operation of the SoC. To create a RTL verification environment we have to came up with a method to write, compile, load and execute programs in a simulation environment. When having C programs running in the simulation setup we can then start extending these programs using the Chipotle test language. By adding the special test instruction defined in the Chipotle test language we can write programs to verify the complete SoC design. The Chipotle test language can easily be extended with special instructions to control input ports and check output ports in the ASIC/FPGA design. Test instruction can also be added to activate test generators and test recorders. These test instructions are monitored and converted to commands executed in the Verilog test bench that will enable the different tests functions.
This document describes how to setup a debug and verification environment built around the Chipotle C-program running in the processor(s).

1.2 References

References mentioned in this document can be found here:

Document	Description

2. Using C-programs for RTL verification

There are many reasons for using a C-program as the main test generator and test validator.

- More engineers know how to write C programs compared to writing VHDL, Verilog or SystemVerilog programs
- The C language is easy to understand and has a lot of built-in functions
- C compilers are very efficient and generate compact program code
- It is a great debugging environment
- The whole simulation setup can be controlled from one place
- By adding special instruction the SoC input can be controlled and outputs can be monitored
- The Chipotle test bench can be up and running long before a SystemVerilog test bench is ready to run
- Simulation time is saved because we don't have to compile and elaborate the RTL test bench for every simulation run
- Low-level software drivers can be tested and verified.
- Simplicity

There are also some disadvantages

- It can not fully replace an OVM SystemVerilog test bench
- Using the processor will probably slow down the simulation speed compared to a SystemVerilog bus function models (BFM)
- Measurement of functional coverage is not fully supported

To be able to use a C-program for RTL verification we have to add a number of functions not normally found in standard C-program. Here is a list of things that have been added to support RTL verification.

- Reading and writing hardware registers and memory locations
- Print text to the console or to a file (without an UART connected)
- Generate pseudo-random data
- Start and stop clock cycle measurements
- Increment pass and error counters
- Display address and data for read and write transfers on the processor data bus
- Start and stop hardware or software data generators
- Drive data on input ports
- Monitor and compare data on output ports
- Display simulation time
- Display statistics of cache read, write and misses
- Detect end of program
- Display memory content in DRAM, SRAM and data caches
- Measuring system clock period times
- Tracing program execution

Let's see how we can solve these tasks from within a C-program. In the following chapters we will go through this list and find a solution to all entries.

3. Housekeeping

First of all we defined some standards we are going to follow.

3.1 Text formatting

We will use the camelBack formatting in the c-programs. This means that all integers, variables and function names will have names like this:

generateRandomData
wordData32

Starting with lower case and the using upper case for following parts of the variable name.

All defines will be upper case only.

```
#define TESTCASE_NAME             "ClockTest"
#define MODULE_TESTED             "CLKGEN"
#define TESTCASE_ID               "1.1.1"
```

3.2 C-libraries used

The C library <stdint.h> is used to define integers with different bit widths. The following types of integers are defined in stdint:

- uint8_t
- uint16_t
- uint32_t
- uint64_t

If we want the processor to write or read bytes we must define the read and write data as uint8_t.

3.3 Reading and writing hardware registers and memory locations

Here are a number of macros that will handle this task. We have to make sure we use the right routine for reading and writing different data sizes (8bit, 16bit and 32bit).

```
// Define read and write 32 bit word macro
#define writeData(addr,data)        (*(volatile uint32_t *) (addr) = (data))
#define readData(addr)              (*(volatile uint32_t*) (addr))

// Define read and write simulation control data
#define writeControl(addr,data)     (*(volatile uint32_t *) (addr) = (data))
#define readControl(addr)           (*(volatile uint32_t*) (addr))

// Define read and write char macro (= 8bits)
#define writeDataChar(addr,data)    (*(volatile char*) (addr) = (data))
#define readDataChar(addr)          (*(volatile char*) (addr))
```

```
// Define read and write byte macro
#define writeData8bits(addr,data)   (*(volatile uint8_t*) (addr) = (data))
#define readData8bits(addr)         (*(volatile uint8_t*) (addr))

// Define read and write 16 bit macro
#define writeData16bits(addr,data)  (*(volatile uint16_t*) (addr) = (data))
#define readData16bits(addr)        (*(volatile uint16_t*) (addr))

// Define read and write 32 bit macro
#define writeData32bits(addr,data)  (*(volatile uint32_t*) (addr) = (data))
#define readData32bits(addr)        (*(volatile uint32_t*) (addr))
```

Usage:

writeData(GPIO_BASE_ADDRESS+GPIO_CONFIG0 ,GPIO_DRIVE_LOW);

Observe:

```
writeControl used for all simulation control statements
writeData    used for all writes to real hardware
```

4. C program extensions

Here follows a description of all extensions added to the standard C language to support RTL verification.

4.1 Memory addresses used for simulation control

A number of memory addresses have been reserved for sending control data to the verilog test bench. These addresses can easily be changed to some other unused space in the memory map if needed.

```
#define ENABLE_TEST                 0xb000a400
#define TEST_SEQUENCE               0xb000a404
#define TEST_RESULT                 0xb000a408
#define EXPECTED_DATA               0xb000a40c
#define MASK_DATA                   0xb000a410
#define DISPLAY_READ_ADDRESS        0xb000a414
#define DISPLAY_WRITE_ADDRESS       0xb000a418
#define DISPLAY_READ_DATA           0xb000a41c
#define DISPLAY_WRITE_DATA          0xb000a420
#define SIM_CONTROL                 0xb000a424
#define CLOCK_CYCLE_LIMIT           0xb000a428
#define PRINT_TEXT_CHAR             0xb000a42c
#define DISPLAY_RESULT_HEX          0xb000a430
#define DISPLAY_RESULT_DEC          0xb000a434
#define WRITE_RESULT_HEX            0xb000a438
#define WRITE_RESULT_DEC            0xb000a43c
#define ENABLE_COUNTER              0xb000a440
```

```
#define CLEAR_COUNTER               0xb000a444
#define DEBUG_MODE                  0xb000a448
#define PRINT_TEST_HEADER           0xb000a44c
#define PRINT_HEADER_DEC            0xb000a450
#define DISPLAY_DATE                0xb000a454
#define DISPLAY_TIME                0xb000a458
#define SIMULATION_WAIT             0xb000a45c
#define DISPLAY_DCACHE_A            0xb000a460
#define DISPLAY_DCACHE_B            0xb000a464
#define DISPLAY_DCACHE_C            0xb000a468
#define DISPLAY_DCACHE_D            0xb000a46c
#define DISPLAY_ICACHE_A            0xb000a470
#define DISPLAY_ICACHE_B            0xb000a474
#define DISPLAY_ICACHE_C            0xb000a478
#define DISPLAY_ICACHE_D            0xb000a47c
#define DISPLAY_SRAM                0xb000a480
#define DISPLAY_SROM                0xb000a484
#define DISPLAY_DRAM                0xb000a488
#define DISPLAY_IRAM                0xb000a48c
#define SRAM_DISPLAY_START          0xb000a490
#define SROM_DISPLAY_START          0xb000a494
#define DRAM_DISPLAY_START          0xb000a498
#define IRAM_DISPLAY_START          0xb000a49c
#define INTERRUPT_ENABLE            0xb000a4a0
#define INTERRUPT_DISABLE           0xb000a4a4
#define TRACE_PC_ENABLE             0xb000a4a8
#define TRACE_PC_DISABLE            0xb000a4ac
#define TEST_COUNTERS               0xb000a4b0
#define READ_COUNTERS               0xb000a4b4
#define SELECT_COUNTER_SOURCE       0xb000a4b8
#define MEASURE_CLOCK_PERIOD        0xb000a4bc
#define REPORT_MEASURED_CLOCKS      0xb000a4c0
```

4.2 Simulation control commands

By writing specific data to the SIM_CONTROL address we can control actions in the simulation flow. Here are the actions supported so far.

```
Action                    Description
-------------------------------------------------------------------------
END_OF_TEST               Defines end of test. The simulation will stop.
CLOCK_CYCLES              Display current clock cycle count
START_MEASURE             Start a clock cycle measurement
STOP_MEASURE              Stop clock cycle measurement and display cycles
PROGRAM_COUNTER           Display current program counter
SIMULATION_TIME           Display current simulation time
INC_ERROR_COUNTER         Increment global error counter
INC_PASS_COUNTER          Increment global pass counter
NO_OPERATION              Generate a no operation
DISPLAY_STATISTICS        Set the maximum number of clock cycles to simulate
```

```
INC_TEST_COUNTER          Increment global test counter
REGRESSION_PRINT_ENABLE   Enable regression test print out
REGRESSION_PRINT_DISABLE  Disable regression print out
```

Example:

writeControl(SIM_CONTROL,END_OF_TEST);

4.3 Print text to the console or to a file

Here is a function that will send one byte at a time to the processor data bus. The characters will be picked up by the verilog test bench and printed using the $write or $fwrite commands.

```
void printTextString (char* volatile text)
 {
   uint16_t        bufferLen;
    uint8_t         i;

    bufferLen = strlen(text);
    for (i = 0; i < bufferLen; i++)
    writeControl(PRINT_TEXT_CHAR ,text[i]);
  }
```

Usage:

```
printTextString  ("\nTest description\n\n");
```

4.4 Start and stop clock cycle measurements

Measuring the time it takes to execute a certain part of the c-progran can give important information about the algorithms we are running. Here are two instruction that will start and stop clock cycle measurements.

```
writeControl(SIM_CONTROL, START_MEASURE);
writeControl(SIM_CONTROL, STOP_MEASURE);
```

Here is a print out from the program:

```
Start clock cycle measurement        :      25025
Stop clock cycle measurement         :      31033
Measured number of clock cycles      :       6008
```

Use this command to print out the current simulation time.

```
writeControl(SIM_CONTROL, SIMULATION_TIME);
```

4.5 Increment pass and error counters

To keep track of the number of passing and failing tests there are three counters built in to the verilog test bench. To increment these counters the following statements are used:

```
writeControl(SIM_CONTROL, INC_TEST_COUNTER);
writeControl(SIM_CONTROL, INC_PASS_COUNTER);
writeControl(SIM_CONTROL, INC_FAIL_COUNTER);
```

4.6 Display statistics from the program execution

During program execution a lot of statistics will be collected. Use this statement to enable the print out of the statistics.

```
writeControl(SIM_CONTROL, DISPLAY_STATISTICS);
```

This is what the print out at the end of the simulation looks like:

```
Number of clock cycles               :      39108
Number of instr cache writes         :        272
Number of instr cache reads          :      23590
Number of data cache writes          :        308
Number of data cache reads           :       1341
Number of data RAM reads             :          7
Number of data RAM writes            :          0
Number of system RAM reads           :       4100
Number of system RAM writes          :       1475
Number of data cache misses          :         79
Number of instruction fetch stalls   :       5680
Number of data fetch stalls          :        389
```

4.7 Detect end of program

The verilog test bench must detect the end of program execution and stop the simulation. If we add this statement as the last instruction in our C-program the test bench will detect end of test.

```
writeControl(SIM_CONTROL, END_OF_TEST);
```

If the program hangs and does not reach end of test we can stop on a maximum cycle count. The cycle count is set in the verilog test bench and can be changed from the C-program using the following statement.

```
writeControl(CLOCK_CYCLE_LIMIT, <clock cycles>);
```

4.8 Print data

A number of statements support printing to the console or file of data that are sent on the processor address and data buses

writeControl(DISPLAY_READ_ADDRESS,<address>);
writeControl(DISPLAY_READ_DATA,<data>);

Printout example:

```
Read address                        :   80000c04
Read data                           :   000000ab
```

writeControl(DISPLAY_WRITE_ADDRESS,<address>);
writeControl(DISPLAY_WRITE_DATA,<data>);

Example:

```
Write address                       :   80000c00
Write data                          :   0000000f
```

The following statements will only print the result with or without a carriage return.

4.9 Test read data

To test some read data use the following setup.

```
// Read GPIO_IN data
   readData32 = readData(GPIO_BASE_ADDRESS+GPIO_READ_DATA);

// Measure GPIO outputs
   writeControl(EXPECTED_DATA,0x5468);
   writeControl(TEST_RESULT,readData32);
```

4.10 Display memory contents

There are a number of control commands that can be used to display the memory content of system memories.

To display data in DRAM use the following commands:

```
// Set start address to display
  writeControl(DRAM_DISPLAY_START,DRAM_ADDRESS_START);
// Display DRAM memory content (1kB)
  writeControl(DISPLAY_DRAM,DRAM_ADDRESS_START+1024);
```

4.11 Display processor program counter

By adding this instruction in the c-program the processor program counter value will be displayed.

writeControl(SIM_CONTROL,PROGRAM_COUNTER);

4.12 Tracing program execution

We can trace the Xtensa program execution by adding the following instructions in the beginning of the c-program:

writeControl(TRACE_PC_ENABLE,<start clock cycle>);
writeControl(TRACE_PC_DISABLE,<stop clock cycle>);

<start clock cycle> specifies the system clock cycle when the trace should start
<stop clock cycle> specifies the system clock cycle when the trace should stop

Here is an example of a print out from the trace

```
Clock cycle :     120000   PC : 48000d90  Instr : d2ddaa01d80020c0
Clock cycle :     120023   PC : 48000d98  Instr : aa1b0020c0000dd2
Clock cycle :     120026   PC : 48000d80  Instr : 7ac28b6bdf74a0a0
Clock cycle :     120027   PC : 48000d88  Instr : 0580007ac28b6bdf
Clock cycle :     120028   PC : 48000d90  Instr : d2ddaa01d80020c0
Clock cycle :     120091   PC : 48000d98  Instr : aa1b0020c0000dd2
Clock cycle :     120094   PC : 48000d80  Instr : 7ac28b6bdf74a0a0
Clock cycle :     120095   PC : 48000d88  Instr : 0580007ac28b6bdf
Clock cycle :     120096   PC : 48000d90  Instr : d2ddaa01d80020c0
```

This information will also be printed to the file: sim/result/debug.res

5. Controlling the hardware

We can also control some hardware functions from the c-program. Here are some examples. You can easily add more hardware support.

5.1 Generate interrupts

We can force the interrupt input of a processor from the c-program using the following commands:

```
writeControl(INTERRUPT_ENABLE, <one bit per interrupt input);
writeControl(INTERRUPT_DISBLE, <one bit per interrupt input);

Enable interrupt for inputs 0 and 3
writeControl(INTERRUPT_ENABLE, 0x9);

Disable interrupt for inputs 0 and 3
writeControl(INTERRUPT_DISABLE, 0x9);
```

For more information see testprogram: sw/program/Interrupt_test

5.2 Testing GPIO

A number of instructions for testing the GPIO module has been added to the Chipotle test language.

Instruction	**Description**
writeControl(DRIVE_GPIO_IN,<data>);	Drive all GPIO inputs with <data>
writeControl(CHECK_GPIO_OUT,<expect_data>);	Test the GPIO port for <data>
writeControl(GPIO_DRIVE_HIGH,<data_high>);	Drive only the “1” bits high. Leave the “0” bits unaffected.
writeControl(GPIO_DRIVE_LOW,<data_low>);	Drive only the “1” bits low. Leave the “0” bits unaffected.
writeControl(GPIO_DRIVER_OFF,<data_off>);	Drive only the “1” bits with “z=tri-state”. Leave the “0” bits unaffected.
writeControl(GPIO_OUT_MASK,<mask_data>);	Mask the “1” bits when testing output data

6. Functions

The following C functions can be used in our c-programs.

6.1 Generate a wait

The following function is used to generate a well defined wait statement.

```
// This function will generate a wait of "wait" clock cycles

void waitCycles (int wait)

 {
   int       i;

   for (i=0; i < wait; i++)
      asm volatile ("nop");
 }
```

6.2 Generate random data

Here are four functions that can used to generate pseudo-random data.

- The first routine will generate data in the full 32 bit range.
- The second routine will generate constraint random data with values in the range max_value <= rand value >= min_value. All values that don't fall within this range will be skipped.
- The third routine will generate constraint random data and convert them to fit in the range max_value <= rand value >= min_value.

```
// This function will generate 32 bit words of pseudo-random data

uint32_t generateRandomData32 (int *seed)

{
    uint32_t      value;

    value = rand_r(seed);
    return value;
}

// This function will generate 16 bit words of pseudo-random data

uint16_t generateRandomData16 (uint16_t *seed)
```

```
{
   int16_t      value;

   value = rand_r(seed);
   return value;
}


// This function will generate 32 bit words of pseudo-random data
// In the range max_value - min_value

uint32_t generateRandomAddr (uint32_t min_value, uint32_t max_value,
                        uint32_t mask_value, uint32_t *seed)

{
  uint32_t      value  = 0;

  while (value <= min_value || value >= max_value)
     value = rand_r(seed) & mask_value;

  return value;
}

// Generate random data in the range max_value - min_value faster than the function above

uint32_t generateRandomFast ( uint32_t min_value, uint32_t max_value,
                         uint32_t mask_value, uint32_t *seed)

{
 uint32_t      value;

 value = ((rand_r(seed) % (max_value - min_value)) + min_value) & mask_value;

 return value;
}
```

6.3 Print binary

The C language has no binary print out format like for example Verilog. Here are three functions that can be sued to print in binary format from our C program. They can be found in the header file: Function.h

```
// Print byte in binary format

void
print8bits(uint8_t byte)
{
	uint8_t		i;
	for (i = 8; i > 0; i--)
		if (byte & (1 << i-1))
			writeDataChar(PRINT_TEXT_CHAR ,(int)'1');
		else
			writeDataChar(PRINT_TEXT_CHAR ,(int)'0');

	writeDataChar(PRINT_TEXT_CHAR ,'\n');
}

// Print 16 bit word in binary format

void
print16bits(uint16_t word)
{
	uint16_t		i;
	for (i = 16; i > 0; i--)
		if (word & (1 << i-1))
			writeDataChar(PRINT_TEXT_CHAR ,(int)'1');
		else
			writeDataChar(PRINT_TEXT_CHAR ,(int)'0');

	writeDataChar(PRINT_TEXT_CHAR ,'\n');

}

// Print 32 bit word in binary format

void
print32bits(uint32_t word)
{
	uint32_t		i;
	for (i = 32; i > 0; i--)
		if (word & (1 << i-1))
			writeDataChar(PRINT_TEXT_CHAR ,(int)'1');
		else
			writeDataChar(PRINT_TEXT_CHAR ,(int)'0');

	writeDataChar(PRINT_TEXT_CHAR ,'\n');

}
```

7. Test generators and test recorders

It is not possible to test everything in a SoC design from a standard C program. Many hardware functions are dependent on external hardware generators to generate specific input data (SPI, USB and I2C). The same thing for output data that must be recorded and analyzed. If we incorporate generator and recorder modules in our Verilog test bench we can extend the testing possibilities.

7.1 Counter

The following macro instruction have been added to support clearing, enabling and reading the counter registers.

```
// Disable all counters
  writeControl(ENABLE_COUNTER,0x0);

// Enable all counters
  writeControl(ENABLE_COUNTER,0xffff);

// Clear all counters (set one bit for every counter to clear)
  writeControl(CLEAR_COUNTER,0xffff);

// Read counter 0
  dataWord32 = readData32bits(READ_COUNTER0);

// Read counter 15
  dataWord32 = readData32bits(READ_COUNTER15);
```