Testing the picoProcessor VHDL Models

Introduction

This document describes the test-bench framework that accompanies the VHDL models of the picoProcessor. The instruction set of the picoProcessor is described in The picoProcessor ISA. The picoProcessor VHDL Models document should be consulted for additional information.

Running A Test Program

There are three test-bench configurations provided with the picoProcessor models. The files

- *test_bench_behav-config.vhd*,
- *test_bench_unpipelined_single_cycle_rtl.vhd*, and
- *test_bench_unpipelined_multicycle_rtl.vhd*

provide test configurations for the behavioral, single-cycle and multi-cycle models, respectively. Each test configuration simply combines the appropriate processor model with the test-bench architecture (described below).

The program code that the test bench executes is loaded into instruction memory from a file named *test.ppe*. Additional test programs are provided in files named *test1A.ppe*, *test1B.ppe*, *test2.ppe*, etc. These files can be used to replace the contents of *test.ppe* and, in this way, load a different test program. Alternatively, the test configurations can be edited so that the test bench reads a file other then *test.ppe*.

Test Bench for the pP

The file *test.vhd* contains an empty entity declaration for a test bench. The file *test-bench.vhd* contains the test bench architecture body. The architecture declares signals to connect to the pP ports. It also declares a pP component with no generics and with ports that match those of the pP entity. The component is instantiated with the signals connected to the ports. The test bench includes four processes:

- `reset_gen`

  This process simply generates a reset pulse between 5 ns to 25 ns to start the test. (When the processor model is instantiated, it does not commence executing until reset.)

- `clock_gen`

  Generates a clock waveform with a 10 ns period and the first rising edge at time 10 ns.

- `int_gen`

  Generates a series of ten interrupt requests. The first occurs after 25 cycles, and subsequent requests occur 25 cycles after the prior request is acknowledged.

- `io_control`

  A simplistic I/O controller. It contains an array of byte values that are used as a “hard-wired” input stream. Any input request by the processor is satisfied by providing the next byte in this array, regardless of the port being read. Note that the controller satisfies a request in a single clock cycle, and assumes that I/O instructions may occur on back-to-back cycles.

Test Programs

The behavior of each of the test programs provided is described below. Note that *test1A.ppe* is the same as the default test program (*test.ppe*).

All the test programs have the same basic structure. The initial instruction jumps to the location in the code where the main program is found. The instruction at location 1 is the start of the interrupt handler, which always ends with a `RETI` instruction. The main program is located at some point after the interrupt handler.

Test 1A

The main code starts at location 16. The first action is to enable interrupts, then register r1 is initialized to 10. The code proceeds to loop, storing r1 into data memory location 0 and decrementing r1. The loop terminates when r1 reaches 0. The code then busy waits at location 21. (This is as close to halting as the picoProcessor allows.)

Test 1B

This has a similar behavior to Test 1A; however, in this case, the loop control variable is decremented by the interrupt handler. The first action of the main code is to initialize register r2 with the value 16. This will be used as the index into data memory where the loop control variable is stored. The memory at this location is initialized to the value 10, and interrupts are enabled. The main code then proceeds to enter a busy loop that repeatedly polls the value in memory indexed by r2. The loop continues until the value stored in memory becomes 0; the program then "halts." The action of the interrupt handling code is to decrement the main program's loop control variable. Effectively, the main program loops until ten interrupt requests have been received by the processor.

Test 2

This program is intended to test the `ADDC` and `SUBC` instructions. The second addition instruction results in the value 255 in register r2. The third addition overflows, so the result in r3 is zero, but the carry bit is set. Thus, the following subtraction effectively subtracts 2 from 0; the result in r4 is 254, and the carry bit is set to indicate that the subtraction underflowed. The second subtraction produces the result 125, and the final subtraction produces the result 0.

Test 3

This program tests some of the bit shifting and bit-wise logical instructions. The registers r1 through r4 are initialized with values that set a single bit. These values are used in a sequence of shifting, rotating and bit-wise combination operations that produce values in registers r5, r6 and r7. If the operations are implemented correctly, the final values in the registers will be as follows: r1 = 1, r2 = 4, r3 = 16, r4 = 64, r5 = 255, r6 = 170 and r7 = 0.

Test 4A

This program tests the output instruction. It sends a sequence of values to output port 15. Specifically, it generates the sequence 0, 1, 2, ..., 9.

Test 4B

This program exercises both the input and output instructions. Its behavior is to read data from input port 0, buffer the data in memory and echo the data to output port 15. Data is read by the interrupt handler; i.e., for each interrupt received, a single byte is read from port 0. The value is placed at the end of the buffer, which starts at data address 1 and extends to the ends of data memory. The memory at address 0 is used to store the offset of the end of the buffer. The main program uses register r2 to count the number of bytes that it has consumed from the buffer. The code repeatedly compares the value in r2 with the value stored in memory at address 0. If there are unconsumed values in the buffer, indicated by a difference between r2 and the value stored in memory, those values are consumed.

Test 5

This program demonstrates a (rather inefficient) way of using the add/subtract with carry instructions to perform multiplication. It also demonstrates subroutines using the `JSB` and `RETI` instructions. The program contains subroutine code at instruction locations 2 through 14. The subroutine takes two 8-bit values from registers r4 and r5, multiplies the values to obtain a 16-bit result and stores the result in memory. The result is stored in big-endian byte order at an offset specified in register r6. Note that the subroutine assumes that register r7 is spare and modifies the value of r5. However, the value of r4 is not altered. The main program code uses the subroutine to generate a series of 16-bit values in memory. It uses registers r3 and r4 to store two working numbers. It executes a loop that multiplies the two numbers, stores their product in memory and then increments each working number. The main program uses r2 to track the memory index where the next result is stored and r1 to count the number of results that have been produced. After generating 15 results, the main program halts.

Test Program Output

Simulating the pP design using the default test program (*test1A.ppe*) yields the following trace:

```
#    0: JUMP 16
#   16: ENAI
#   17: ADD  R1, R0, 10
#   18: STM  R1, 0
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   18: STM  R1, 0
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   18: STM  R1, 0
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   18: STM  R1, 0
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   18: STM  R1, 0
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   18: STM  R1, 0
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   18: STM  R1, 0
# Interrupt acknowledged at PC =   19
#    1: RETI
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   18: STM  R1, 0
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   18: STM  R1, 0
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   18: STM  R1, 0
#   19: SUB  R1, R1, 1
#   20: BNZ  -3
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
#   21: JUMP 21
# Interrupt acknowledged at PC =   21
#    1: RETI
#   21: JUMP 21
     ...
```

The single- and multi-cycle models will execute with different timings, so the interleaving of interrupt requests, interrupt handler instructions and main program instructions will differ between the models. However the final outcome of a program will be the same.

If a program executes I/O instructions, trace lines similar to the following are produced:

```
#    1: INP  R7, 0
# IO: port read; address = 0, data = 00001100
```

This is the form of output produced by the multi-cycle model. With the single-cycle model, the trace line generated by an instruction is emitted at the end of a cycle. As a consequence, the "`IO: port read ...`" trace line actually appears before the "`INP ...`" trace line.

Discussion

It should be noted that the test programs are far from comprehensive. Collectively, the programs do not exercise all instructions (although a least one from each class in the ISA is used). They are more useful for debugging the VHDL code of the model, rather than for exploring and verifying the behavior of the specific processor design models. Further, the test bench in which the processor models are being investigated is quite simplistic.

To improve the comprehensiveness of the test programs, an important consideration is the investigation of boundary conditions. For example, the test program *test2.ppe* aims to verify that the carry bit is used properly by executing various `ADDC` and `SUBC` instructions that subtract one from zero, add one to 255, etc. This is the only code in the suite that explicitly aims to verify the operation of a particular instruction under circumstances where subtle errors are likely to be revealed. The test programs could be expanded to cover other boundary conditions.

Another way in which the test suite could be improved is to investigate more exhaustively the interleaving of main program code and interrupt handling code. The code in *test4B.ppe* demonstrates an approach to achieving this. Under the multi-cycle execution model, the interrupt handler code executes in one less instruction than the rate at which interrupts are being generated. As a result, the main program code is only able to execute a single instruction between the handling of each interrupt request. This facilitates verification of the desired behavior for interrupts (i.e., that they are interleaved atomically between instruction execution).

Finally, the test bench configuration used to drive the execution of the processor models could be improved in many ways. Notably, the I/O controller could be more realistic. For example, the controller is driven by the same clock signal as the processor. If it were altered to use a different clock signal the behavior of an asynchronous I/O controller could be modeled.