Department Of Engineering Science
Simon Frasaer University

ENSC 350 - Digital System Design

Spring 2018

Lab 6 : System Bus and Custom Instructions

Working week: Mar 26-30, 2018
Marking week: Apr 2-6, 2018

Overview

In this lab, you will gain experience in implementing a master for a simple bus interface and a set of bit manipulation instructions for a processor that implements the RISCV instruction set [1]. The processor will run a provided application that compares the performance of software implementations of different bit manipulation functions to your hardware implementation. The outputs of the tests will be written to a UART block that is connected to an Avalon bus as shown in Figure 1.

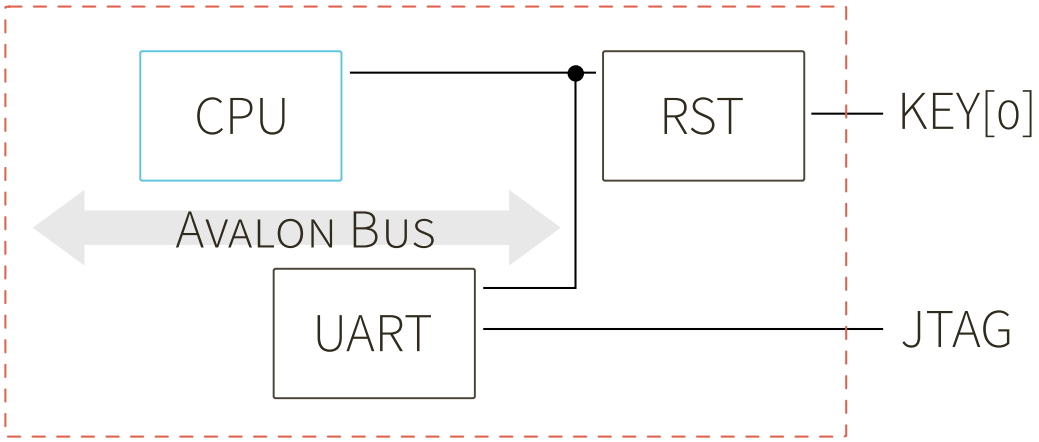

Figure 1 System Diagram. Dotted line indicates FPGA boundary

Background

There are many different ways of increasing the performance of a processor design. Most processor designs use pipelining to split the processor into different functional stages to improve clock frequency. A pipelined processor can have multiple instructions in-flight, but only one in each pipeline stage. Another approach, that is compatible with pipelining, is to have multiple execution units. A design with multiple execution units can execute more instructions in parallel and can better handle instructions with different latencies as each unit can be designed to the specifications for the instructions it processes. A simple example of which is shown in Figure 2. In this design, the processor has a single fetch unit decoupled from the issue and execute logic by a FIFO, and multiple execution units.

Figure 2 Processor Pipeline

The particular processor you will be working with has six different execution units: an Arithmetic Logic Unit (ALU), branch unit (Br), Control Status Registers unit (CSR), Multiply Unit (Mul), Divide Unit (Div) and Load Store unit (LS). In this lab you will be implementing a new bit operations unit as shown in Figure 3.

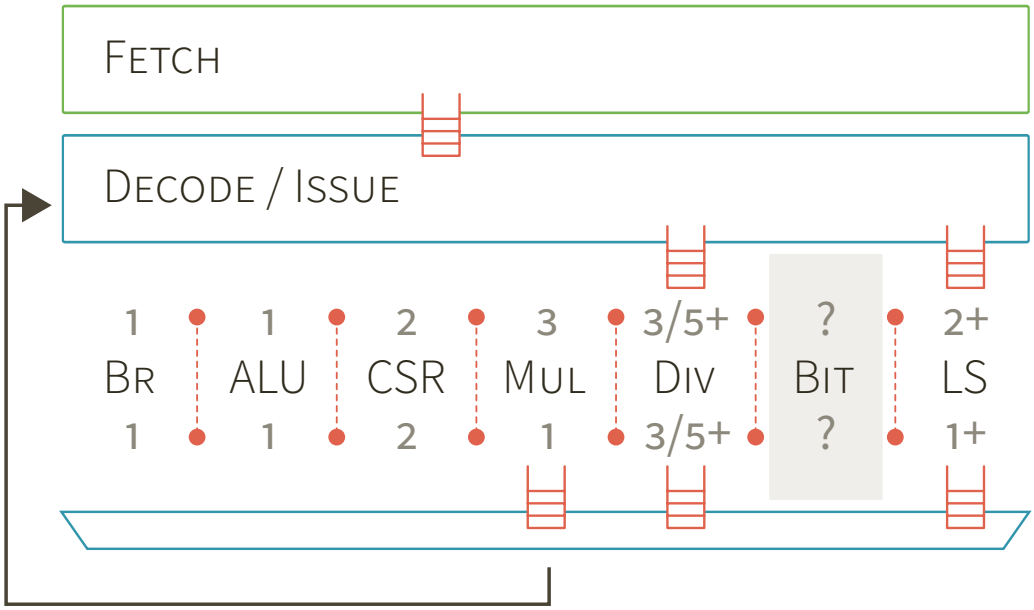

Figure 3 Latency details for execution units. The upper number indicates the latency to obtain the result of an instruction for each unit. The lower number indicates the sustained throughput rate in cycles. For the ALU, this means the ALU can complete one instruction per cycle with a one cycle latency. For the CSR unit, this means it can complete an instruction every other cycle and cannot start another instruction until the first one completes. For the DIV unit, latency is variable. A divide is typically 5 or more cycles and only one division operation can be in progress at once (although operations can be buffered in an input FIFO). In some cases, a divide will only take three cycles (happens when a subsequent DIV/REM instruction reuses the same operands as the immediately preceding instruction.

As the processor supports arbitrary latencies and throughputs for its units, when you design your bit operations unit it will be up to you to decide its performance characteristics.

Avalon Bus

The Avalon bus [2], is a bus standard designed by Altera for their FPGAs. Compared to other bus standards, such as ARM's AXI bus or IBM's PLB, the avalon bus is quite straightforward in operation. For this lab you will need to read the documentation referenced here in order to understand how the bus signals operate.

Bit Manipulation Instructions

The three bit manipulation instructions you will be implementing in this lab are: Count Leading Zeros (CLZ), Population Count (POPC) and byte swapping (BSWAP). All three of these instructions are common in commercial processors as their behaviours occur frequently in some common applications and they can be performed much more efficiently in hardware than software.

CLZ, and POPC both require iterating over bit indexes in software, in basic implementations[1], and thus can be quite slow. Both of these instructions can be implemented in a single cycle in hardware without difficulty. Byte swapping does not take many instructions to perform in software, but is a very common operation when interacting with different hardware (for example ethernet devices).

Count Leading Zeros (CLZ)

Count Leading Zeros counts the number of zero bits starting from the most significant bit (31) until a bit that is set to one is found. If no bits are set to one, the result is 32. The software implementation of this operation used in this lab is provided in Figure 4.

Figure 4 CLZ C code

```
u32 count_leading_zeros(u32 number) {
    for (int i = 31; i >=0; i--) {
        if (number >> i) return 31-i;
    }
    return 32;
}
```

[1] A good collection of more optimized software bit manipulation techniques: https://graphics.stanford.edu/~seander/bithacks.html

Population count (POPC)

Population count returns the number of bits that are set to one in the word. If no bits are set, the result would be zero, and if all the bits are set the result would be 32. The software implementation of this operation used in this lab is provided in Figure 5.

Figure 5 POPC C code

```
u32 popc (u32 number) {
    u32 result = 0;
    for (u32 i = 0; i < 32; i++) {
        result += (number & 0x1);
        number = number >> 1;
    }
    return result;
}
```

Swap Bytes (BSWAP)

Byte swapping converts a word from big-endian to little-endian or vice versa. The software implementation of this operation used in this lab is provided in Figure 6.

Figure 6 BSWAP C code

```
u32 byte_swap (u32 number) {
    u32 result;

    result  = 0x000000FF & (number >> 24);
    result |= 0x0000FF00 & (number >> 8);
    result |= 0x00FF0000 & (number << 8);
    result |= 0xFF000000 & (number << 24);

    return result;
}
```

Task 1: Getting Started

1.1 Provided Files

Files for this project are split into two directories **sw** and **hw**. The **hw** directory has all the files needed to build the processor and stubs for your Avalon bus im-

plementation and bit operations unit implementation. The **sw** directory contains the C code the processor will run along with the disassembled binary (so you can see the assembly instructions) and two RAM initialization files for the processor's instruction RAM **debug.raminit** and **demo.raminit**. The debug version has additional **printf** calls to help you verify the output of your bit operations.

1.2 Build the system

Your first task will be to put the system together and find the maximum operating frequency. Make sure to add all the files in the base **hw** directory including the .sdc file and, additionally, the **.qip** file in the auto_gen directory. Next you will need to modify the line:

```
localparam RAM_FILE = "";
```

in **lab6.sv** to point to the *absolute* path to the **demo.raminit** in the **sw** directory. After you have done that, modify the **CLOCK_50** listing in the **.sdc** file and change the period to **10ns** (an unachievable operating frequency). Rebuild the system and **screenshot** the maximum operating frequency and what the critical path is in the design. You will use this information later to determine if your bus implementation and bit operations unit have impacted the critical path of the processor. Once you have recorded this information, change the period back to **20ns**, remove the placeholder code for the avalon_master and bitops_unit and start Task 2.

Task 2: Implement the Avalon Bus (3 marks total)

In this task, you will be implementing an Avalon bus master (avalon_master.vhd) that integrates into the processor's load store unit. A block diagram showing the connectivity between the bus master, load store unit and Avalon bus is shown in Figure 7. You will need to refer to the official documentation for the Avalon bus [2] to understand how the bus signals behave. Specifically you will want to look at *Section 3.5.3 Read and Write Transfers with Fixed Wait-States*. As the only peripheral in your system is the JTAG UART, your bus behaviour only needs to support the UARTs bus behaviour, which in this case has a fixed latency of zero cycles.

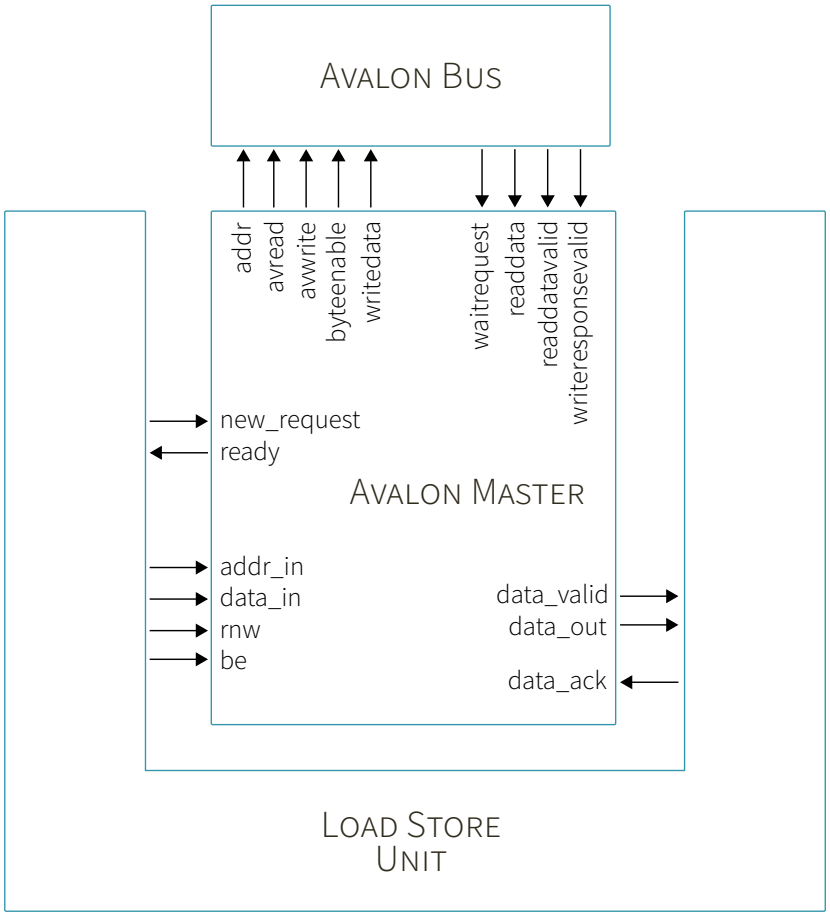

Figure 7 Connectivity of the Avalon Master

2.0.1 Load Store Unit signal details

The Load Store unit uses simple handshaking signals to send requests to the Avalon master and to transfer data. Details of how the signals behave is provided below:

Please note, all resets are active high in this design

new_request is asserted when the Load Store unit has determined that a request is to be sent to the bus AND your ready output is high.

ready Should be high when you can handle a new request on the next cycle. The default value out of reset for this signal can be high.

addr_in is valid when new_request is one. (Should be sampled with new_request)

data_in is valid when new_request is one. (Should be sampled with new_request)

rnw Read-not-write signal, is valid when new_request is one. (Should be sampled with new_request)

be Byte enable signal, is valid when new_request is one. (Should be sampled with new_request)

data_valid Should be one when your data out is currently valid and has not been acknowledged yet by the Load Store unit. Note, writes will never set data_valid as they do not return results.

data_out Your data output, must be valid when data_valid is high and remain unchanged until data_ack has been asserted.

data_ack Will be asserted in response to data_valid. May occur on the same cycle that data_valid goes high or on a subsequent cycle.

To assist in understanding the handshaking behaviour between the Load/Store unit and your avalon bus sub unit we have provided the following code to generate a waveform showing an example read, and an example write transaction. Paste the code into the following online editor `http://wavedrom.com/editor.html` to see the waveform.

FIGURE 8 Wavedrom code for example Load/Store unit handshaking

```
{signal: [
  {name: 'clk', wave: 'p...|.....|..'},
  {name: 'new_request', wave: '0.10|...10|..'},
  {name: 'ready', wave: '1..0|.1..0|.1'},
  {name: 'addr_in', wave: 'x.=x|xxx=x|..'},
  {name: 'data_in', wave: 'x..x|xxx=x|..'},
  {name: 'rnw', wave: 'x.1x|...0x|..'},
  {name: 'be', wave: 'x..x|xxx=x|..'},
  {},
  {name: 'data_valid', wave: '0...|1.0..|..'},
  {name: 'data_out', wave: 'x..x|=.x..|..'},
  {name: 'data_ack', wave: '0...|.10..|..'},
]}
```

2.1 BLOCK DIAGRAM (1 MARK) **INDIVIDUAL**

Before you implement your master interface you will need to draw a block diagram of the logic. You may use high level constructs such as registers with enables, comparitors, adders, muxes etc. All wires in the diagram should be labelled and their bit widths indicated. For this simple bus implementation no state machine is required. Your state is based around your handshaking logic involving the ready/new_request and data_valid/data_ack signal pairs. To receive a full mark,

the diagram should show your initial design (regardless of whether it is correct or not) and should be fully labelled. This should be done individually and will be collected at the beginning of your demo. (Hand drawn is perfectly acceptable).

2.2 Implementation (2 marks)

Once you have completed your block diagram, translate your design to VHDL, (there should be a one-to-one mapping from your diagram to the HDL). To test your design, download it to the board, then open the UART terminal:

```
C:\altera\15.0\quartus\bin64\nios2-terminal
```

and *hold* **KEY0** to take the system out of reset. Note that the UART terminal can not be open when programming the board. If you have implemented the circuit properly you will see output similar to the example output shown at the end of this document. Only choose the software option at this point as you do not have hardware versions of the bitops at this point in the lab. If you do not see any output, or you have garbled output, you may want to try using SignalTap and add all this signals shown in Figure 7.

Task 3: Implement the Bitop Instructions (6 marks total)

With the bus interface implemented you are now ready to implement the bit manipulation instructions. All your changes will be contained within the **bitops_unit.vhd** file, a block diagram representation of which is shown in Figure 9. All other changes needed to support the new instructions (to the decoder, writeback support etc.) have already been made for you. As a hint, it should be fully possible to implement all bitop operations in a single cycle.

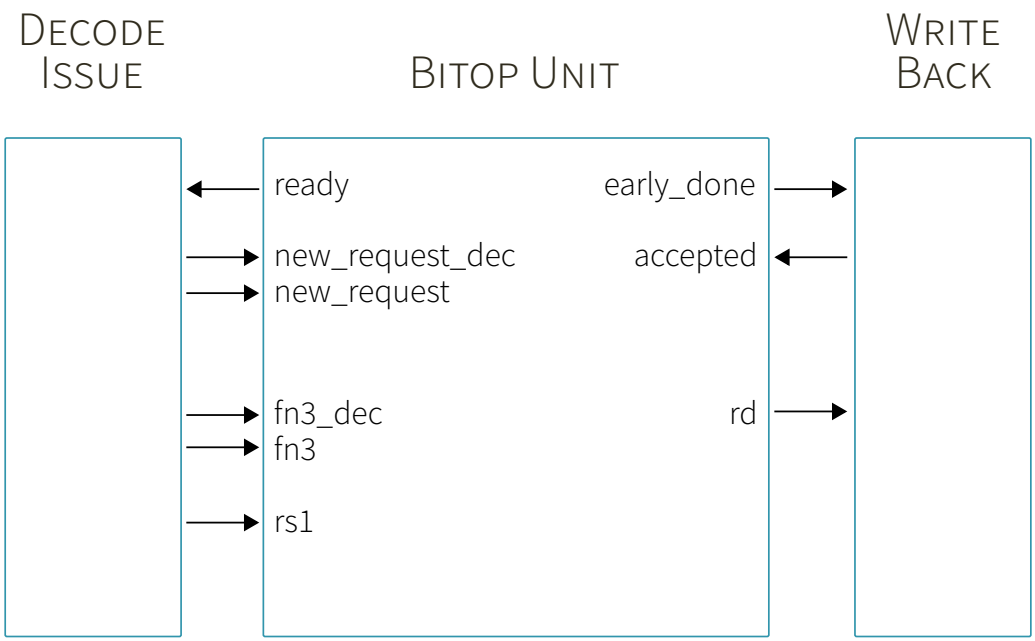

Figure 9 Bitop Unit Connectivity

3.0.1 Bitop Unit signal details

Like the Avalon bus master, the Bitop unit uses handshaking between the decode/issue logic and the writeback logic. For some signals there are decode stage variants (suffixed with **dec**). The registered versions of these signals do not have the **dec** suffix. The reason both of these signals have been provided is for frequency and throughput optimization purposes. The ready and early_done signals tie into the control logic that determines if an instruction can be issued in a particular cycle. As such, it is important that these signals have as little delay as possible. Using the dec signals we can have some of the logic for ready/early_done registered instead of being purely combinational, thus cutting down the amount of logic we add to the issue logic. You can choose to use the dec signals or not in your implementation. Your design can be implemented either way. If you do not use them, however, you might be more likely to see your logic showing up in the critical path of the processor, and/or you will not be able to match the reference timing results and earn a potential bonus mark.

new_request_dec is asserted when the decode/issue logic has determined that a request is to be sent to the bus AND ready is high.

new_request is a one cycle delayed version of new_request_dec.

ready Should be high when you can handle a new request on the next cycle. Should default to high as a result of reset.

fn3_dec is valid when new_request_dec is one (only valid while new_request_dec is asserted).

fn3 The sub-operation for the unit. Is valid when new_request is one and is held constant until the next assertion of new_request.

rs1 the input data (32 bits). Is valid when new_request is one and is held constant until the next assertion of new_request.

early_done Should be asserted one cycle before your output is ready (does not matter if delayed, but will limit your throughput). Remains asserted until accepted is asserted.

rd Your data output.

accepted Asserted by the writeback logic at least one cycle after early_done being asserted. Your data output is sampled on the cycle in which accepted is asserted.

3.0.2 FN3 DETAILS

"000" CLZ

"001" POPC

"010" SWAPB

others SQRT

3.1 Timing Diagram (1 mark) **Individual**

Before you implement your design you should take the time to make sure you fully understand the relationship between all the control signals. As such, you are required to create a timing diagram showing at least three different instructions being issued to the bitop unit and being completed, each with different timing characteristics. (i.e. different delays for the new request and accepted signals). This should be done individually and will be collected at the beginning of your demo. (Hand drawn is perfectly acceptable).

As an aid to get you started, use the following code below with the `http://wavedrom.com/editor.html` to see some example unit transactions. Shown are three back-to-back instructions being issued with no delays caused by the unit.

For the third instruction there is a one cycle delay in the acknowledgement of the result.

FIGURE 10 Wavedrom code for example bitops waveforms

```
{signal: [
  {name: 'clk', wave: 'p........'},
  {name: 'new_request_dec', wave: '0.1..0...'},
  {name: 'new_request', wave: '0..1..0..'},
  {name: 'ready', wave: '1....01..'},
  {name: 'early_done', wave: '0.1...0..'},
  {name: 'accepted', wave: '0..1.010.'},
  {},
  {name: 'fn3_dec', wave: 'x.===x...', data: ['ins1', 'ins2', 'ins3']},
  {name: 'fn3', wave: 'x..===x..', data: ['ins1', 'ins2', 'ins3']},
  {name: 'rs1', wave: 'x..===x..', data: ['ins1', 'ins2', 'ins3']},
  {},
  {name: 'rd', wave: 'x..===.x.', data: ['ins1', 'ins2', 'ins3']},
]}
```

3.2 IMPLEMENTATION (4 MARKS)

With a solid understanding of all the control signals, begin your implementation in **bitops_unit.vhd**. Some suggestions on how to get started: start by simply confirming that you have implemented the control signals correctly by tying **rd** to a constant value and confirming that that the hw versions of the functions complete. You may find the **debug.raminit** file helpful in verifying the output of your unit as it prints out the result for each computation. The **demo.raminit** should be used to check that your design is generating the same results as software and to see the speedup obtained. The software test does not exhaustively test all behaviours of the control signals or all possible inputs. As such, you should be able to explain in your demo why you think your design is fully correct.

3.3 TIMING REPORT (1 MARK)

Once you have completed your bitop implementation (or gotten as far as you have been able to, i.e. you can get this mark even if you do not have a working design). Repeat the steps from Task1 and find the maximum operating frequency of the system. **Screenshot** both what the value is and what the current critical path is. Present this information during your demo along with the data you collected from Task1.

Task 4: Challenge Task (1 mark)

If you are looking for a further challenge you can consider implementing an integer square root instruction for your bitop unit. Unlike the bitop instructions, a square root operation *cannot* be completed in a single cycle. As such, in addition to creating the logic for the square root operation you will need to modify the handshaking logic too.

Square Root (SQRT)

Figure 11 contains the C code used in this lab for the integer square root. It uses Newton's method and the CLZ function. This example would not translate well to hardware due to the use of the divide operation. You will likely want to find an algorithm that uses neither division nor multiplication.

Figure 11 SQRT example C code

```
u32 square_root (u32 number) {
  u32 guess;
  u32 n_over_quess;

  if(number > 1) {
    guess = 1 << (16 - count_leading_zeros(number - 1)/2);
    do {
      n_over_quess = number/guess;
      guess = (guess + n_over_quess) / 2;
    } while (n_over_quess < guess);
    return guess;
  }
  else {
    return number;
  }
}
```

Performance Goal (1 Bonus mark)

At the end of this document is a reference output for the bitops and square root results. For the bitops unit, all bitops can be completed in one cycle and the unit can start one request per cycle. The square root takes multiple cycles and has variable latency depending on the input. If you can achieve runtime results that meet or exceed these results (for every operation) you will receive one bonus mark.

Best Implementation (1 Bonus mark)

A common metric for comparing systems is runtime performance per LUT. For this lab, if you meet the performance goal, also have ready your system resource usage and operating frequency, (and runtime if you achieve better than the reference results). The group with the best result will receive an additional bonus mark. Good luck!

Demo Procedure

For this lab submit just your **bitops_unit.vhd** and **avalon_master.vhd** files. Have your files submitted before your demo time, with both partners submission pages open showing that the files have been submitted. Additionally, have your block diagram and timing diagrams ready for collection and your screenshots open on your computer showing your timing results and critical path.

References

[1] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-118, May 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[2] Altera Corporation, *Avalon Interface Specifications*, 2015. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf

Reference output

Your output should have the same numbers for all the *result* values, but will likely have different time values, due to implementation differences and UART behaviour. UART behaviour only affects the times between tests and does not impact the *user time* metric. *User time* is affected only by your bitops latency and initiation interval.

Note, using the hardware reset after running the code once does not guarantee that you will get the same output. The hardware state will be reset, but the software may not run correctly as the Block RAMs are not reset outside of reprogramming the board. As such, if you want to run the program twice, reprogram the board between tests.

```
Software Only test, input :0
Hardware Only test, input :1
Software and hardware test, input :2

Please input choice for Count Leading Zeros test:
Response was 2:
Starting Count Leading Zeros sw test...
Begin time: 84663317
End time: 135720912
User time: 51057595
Test time in microsecond: 1021151
result: 8444619
Begin inst: 36284482
End inst: 79938694
User inst: 43654212

Starting Count Leading Zeros hw test...
Begin time: 140973759
End time: 147143104
User time: 6169345
Test time in microsecond: 123386
result: 8444619
Begin inst: 82193662
End inst: 88363011
User inst: 6169349

Please input choice for Population Count test:
Response was 2:
Starting population count sw test...
Begin time: 272355711
End time: 484585477
User time: 212229766
Test time in microsecond: 4244595
result: 15945120
Begin inst: 142029343
End inst: 346855745
User inst: 204826402
```

```
Starting population count hw test...
Begin time: 488821411
End time: 494990756
User time: 6169345
Test time in microsecond: 123386
result: 15945120
Begin inst: 348675302
End inst: 354844651
User inst: 6169349

Please input choice for Swap Bytes test:
Response was 2:
Starting swap bytes sw test...
Begin time: 591590706
End time: 606396816
User time: 14806110
Test time in microsecond: 296122
result: 1160025042
Begin inst: 396248585
End inst: 411054698
User inst: 14806113

Starting swap bytes hw test...
Begin time: 612845042
End time: 619014387
User time: 6169345
Test time in microsecond: 123386
result: 1160025042
Begin inst: 413822308
End inst: 419991657
User inst: 6169349

Please input choice for square root test:
Response was 2:
Starting square root sw test...
Begin time: 703400698
End time: 840799411
User time: 137398713
Test time in microsecond: 2747974
result: -1572468426
Begin inst: 456161272
End inst: 535089800
User inst: 78928528

Starting square root sw with hw clz test...
Begin time: 844740477
End time: 924914402
User time: 80173925
Test time in microsecond: 1603478
result: -1572468426
Begin inst: 536782966
End inst: 565890182
User inst: 29107216

Starting square root hw test...
Begin time: 930968168
End time: 956120806
User time: 25152638
Test time in microsecond: 503052
result: -1572468426
Begin inst: 568488769
End inst: 574658113
User inst: 6169344
```