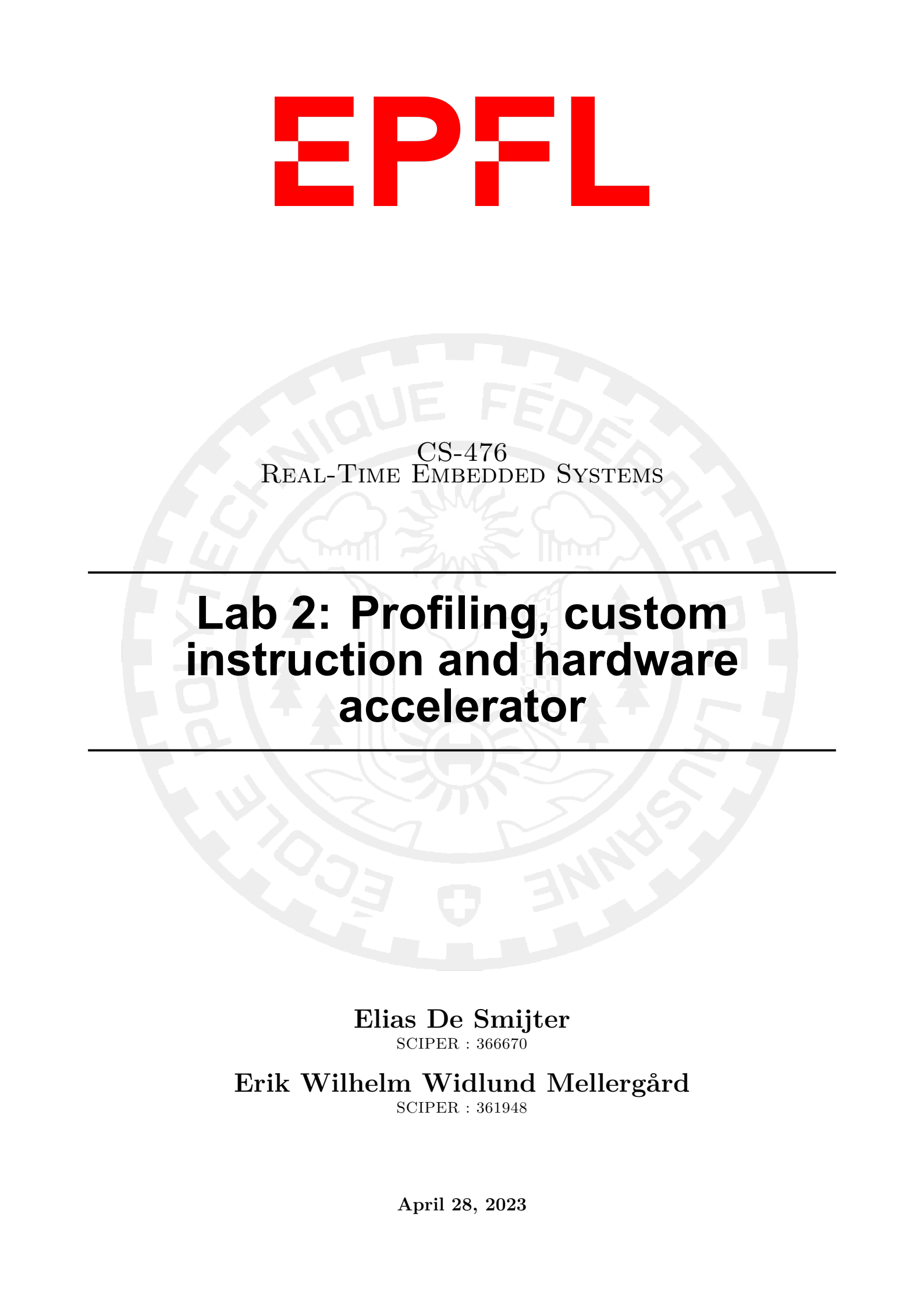

CS-476
Real-Time Embedded Systems

Lab 2: Profiling, custom instruction and hardware accelerator

Elias De Smijter
SCIPER : 366670

Erik Wilhelm Widlund Mellergård
SCIPER : 361948

April 28, 2023

Contents

Introduction

This is the report for the second lab of Real-time Embedded Systems (CS476). The purpose of this lab is to measure how implementing instructions in hardware can speed up the execution time of a program drastically. To see these results, one instruction (defined in section 1) was implemented in three ways: first purely in software, second with a custom hardware instruction that returns the result to the software and third with a hardware accelerator that directly writes the result to memory using a Direct Memory Access (DMA) module.

The profiling of the different implementations happened in hardware: a performance counter was used to measure how long everything took.

1. Instruction

The instruction that will be implemented three times is shown in figure 1. It consists of two parts: the 8 first bits are swapped with the 8 last bits and the 16 bits in the middle are flipped.

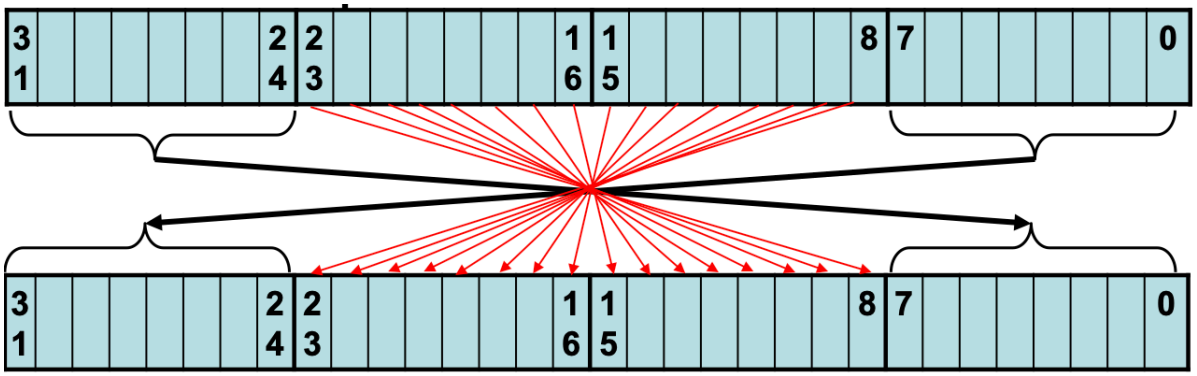

Figure 1: Instruction to implement

1.1 Implementation in pure software

The implementation of this instruction in pure software is done in C. The code follows the reasoning of figure 1 very closely:

```
1  uint32_t software_custom_inst(uint32_t input)
2  {
3    uint32_t output;
4    output = (input<<24) & 0xff000000;
5    output = output + ((input & 0xff000000)>>24);
6    uint32_t sought_bit;
7    int i;
8    for(i=8; i<24; i++)
9    {
10     sought_bit = (input<<i)&0x80000000;
11     output = output + (sought_bit>>(31 - i));
12   }
13   return output;
14 }
```

On line 4 and 5, the 8 highest and lowest bits get selected and swapped. For the flipping of the middle 16 bits, a for-loop is used which flips bit by bit. The implementation was tested in the main c script, see Figure 2 for the results and Annex 4.1 for the c code.

```
Software implementation test 1: Expected is 1, it gives 1
Software implementation test 2: Expected is 0, it gives 0
```

Figure 2: Software implementation test result

2. General schematic

The overall system, see Figure 3, implements the necessary components for the task. Descriptions of our own components; the custom instruction and the accelerator, follow below.

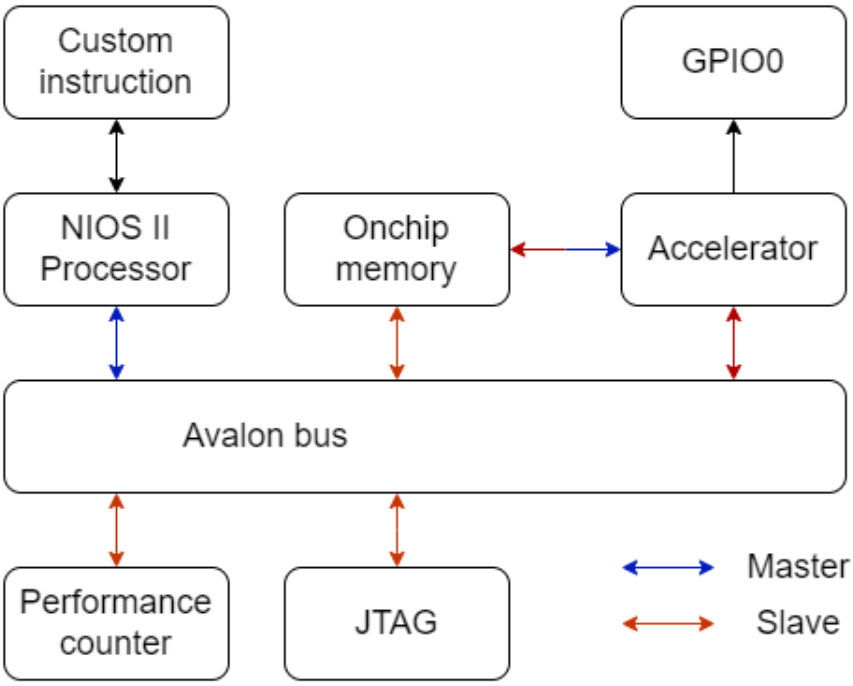

Figure 3: Overall system design

2.1 Custom instruction

This implementation of the operation is as a custom instruction for the NIOS II processor, which is accomplished with a very short .VHD file, see Annex 4.2, describing the operation. The file is used in Platform Designer to define what operation should be done by the processor when the custom instruction is called. The custom instruction was tested in the main c script, see Figure 4 for the results and Annex 4.1 for the c code.

```
Custom instruction implementation test 1: Expected is 1, it gives 1
Custom instruction implementation test 2: Expected is 0, it gives 0
```

Figure 4: Custom instruction implementation test result

2.2 Accelerator

The idea is to implement a Direct Memory Access (DMA) module that reads indata directly from memory, performs the operation on it, and then stores the outdata directly to memory. Despite a lot of time spent on it, we did not get the hardware accelerator working fully. We tried 4 different implementations, Accelerator v1 to v4, each at a different level of complexity, but had the same issue in all of them. They are all based on a Finite State Machine (FSM) that can be seen in Figure 5, which we attempted to implement in different ways.

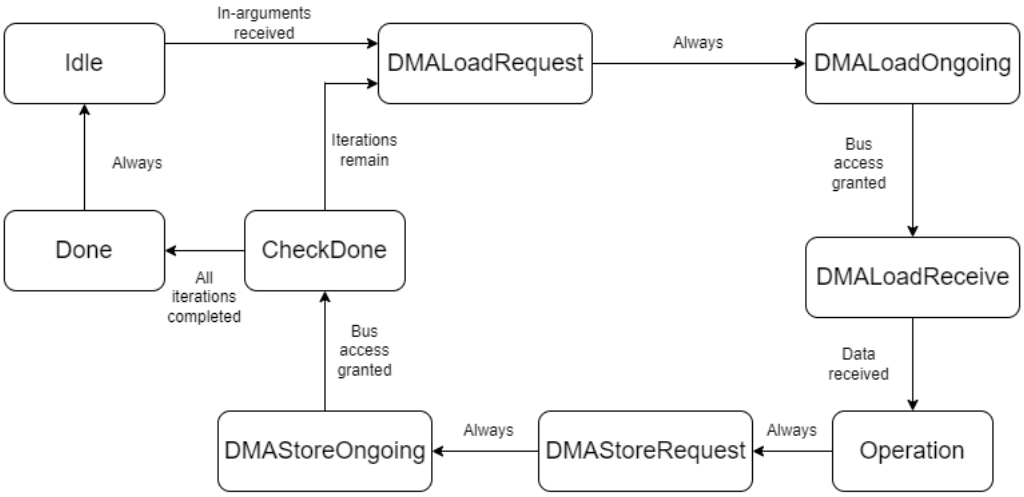

Figure 5: Finite State Machine

The code for each of the implementations can be seen in Annexes 4.3 to 4.6, and their differences are in short described here:

- v1: This implementation is the simplest, gathering the entire state machine and the Avalon Master interface in one single process. Only the Avalon Slave interface is implemented in separate processes.
- v2: The second implementation moves the counter update into its own process, but is otherwise unchanged from the v1 implementation.
- v3: The third implementation keeps all the essential working in one large main process, just like the v1 implementation, but uses a Variable, Variable_Next implementation where the main process updates all the Variable_Next signals. A separate update process handles the task of updating Variable <= Variable_Next. This implementation is intended to prevent the use of latches by the compiler.
- The fourth implementation uses the Variable, Variable_Next method just like the v3 implementation. Further, it also separates the state machine into two processes where one updates State_Next and the Avalon Master interface, while the other process updates the other internal signals.

The main problem seems to be the counter. It is meant to keep track of how many iterations of the operation have passed (thus counting the number of cycles in the state machine). In implementations v1 and v4, the counter increments in every clock cycle, causing the FSM to reach the Done state prematurely. This behaviour can be seen in Appendices 5.1 and 5.4.

In implementations v2 and v3, the counter instead has the value zero in all states except for the specific state where the counter should increment by 1, but the value jumps to much more than 1 immediately in this state. This causes these versions of the accelerator to never reach their target values at all, leading to infinitely continuing with the operation. Records of this behaviour is available in Appendices 5.2 and 5.3.

Beyond the counter problem, Figure 6 demonstrates that the Accelerator v4 implementation is indeed going through the states as it should. In the figure, channel 0 through channel 6 each represent the cycle of states for loading, performing the operation, storing the result, and finally the CheckDone state. Channel 7 represents the Done state: being done with the last iteration, and thus returning to the Idle state.

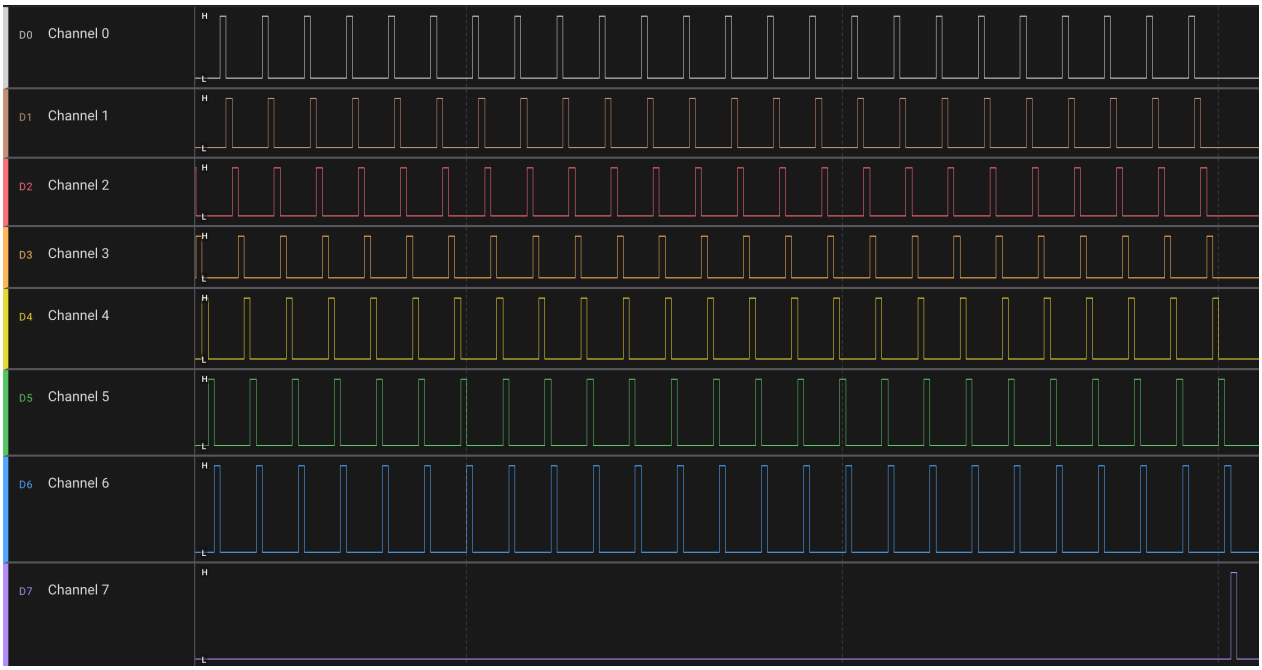

Figure 6: FSM flow, Accelerator V2

3. Profiling

The performance counter is used to measure how long the program spends in its various sub-routines. The result shows time spent in each sub-routine as a percentage of all time, in seconds, and in clock cycles. The idea was to perform 1 000 iterations for each method, and record the results of that. However, due to the counter problem with the accelerator described above in section 2.2, we were unable to get the accelerator to do exactly 1 000 iterations.

The problem led to the entire load->operation->store cycle, which takes about 8 clock cycles, resulting in about 8 increments of the counter in stead of the desired 1 increment. An exact number can't be found since some of the states may take an unknown amount of clock cycles: waiting for the Avalon bus during load and store states if it's busy at the first request. However, using the rough estimate of clock cycles per operation cycle, we were able to modify the C code such that 240 operation cycles were performed, and then compare that in profiling with 240 iterations each for the software implementation and custom instruction. Despite the counter problem, this enables a comparison of performance, the output of which can be seen in Figure 7.

```
Not busy
Writing arguments
Waiting
Done
--Performance Counter Report--
Total Time: 0.00446724 seconds  (223362 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|C software     | 76.6|    0.00342|         171099|          1|
+---------------+-----+-----------+---------------+-----------+
|Custom instr.  | 4.48|    0.00020|          10003|          1|
+---------------+-----+-----------+---------------+-----------+
|Accelerator    | 15.1|    0.00068|          33774|          1|
+---------------+-----+-----------+---------------+-----------+
```

Figure 7: Profiling, 240 iterations

Unsurprisingly, the software implementation is by far the slowest. When calling the custom instruction implemented in VHDL, the speedup is very significant: from 171 099 clock cycles to 10 003, corresponding to a 17-fold improvement. This is as expected because the entire instruction can be executed in one clock cycle by the custom instruction. In the software implementation on the contrary, each statement takes at least one clock cycle making the entire instruction take multiple clock cycles. The 16 bits to shift individually in the for loop are most likely the main cause of slowness for the software implementation: these bits are shifted sequentially in software, and concurrently in both the custom instruction and the accelerator implementations.

It is unexpected that the hardware accelerator module takes a longer time (three times as long!) than the custom instruction. It should be shorter because of the DMA module, but probably due to the sub-optimal implementation this feature is not used optimally.

4. Annex 1: Source code

4.1 Main C program

```
#include <stdio.h>
#include "sys/alt_stdio.h"
#include "sys/alt_cache.h"
#include <altera_avalon_performance_counter.h>
#include "io.h"
#include "system.h"
#include <stdint.h>


// Declare a test number and it's expected result from the custom instruction
uint32_t test = 0x12AAAAEF;
uint32_t truth = 0xEF555512;

// Declare tables for testing 1000 long data
uint32_t in_1000[1000];
uint32_t out_1000[1000];

// Define macros for the accelerator
#define ACC_BUSY       0x00000002
#define ACC_DONE       0x00000001
#define ACC_InputStartAddr  0
#define  ACC_OutputStartAddr  4
#define ACC_Num         8
// Dumbest
//#define ACC_ADDR_BASE    DUMBESTACCELERATOR_0_BASE
// Dumb
//#define ACC_ADDR_BASE    DUMBACCELERATOR_0_BASE
// Base
//#define ACC_ADDR_BASE    ACCELERATOR_0_BASE
// V2
#define ACC_ADDR_BASE    ACCELERATOR_V2_0_BASE

void init_tables()
{
  int i;
  for(i=0; i<1000; i++)
  {
    in_1000[i] = test;
  }
  alt_dcache_flush_all();
}

uint32_t software_custom_inst(uint32_t input)
{
  uint32_t output;
  output = (input<<24) & 0xff000000;
  output = output + ((input & 0xff000000)>>24);
  uint32_t sought_bit;
```

```
  int i;
  for(i=8; i<24; i++)
  {
    sought_bit = (input<<i)&0x80000000;
    output = output + (sought_bit>>(31 - i));
  }
  return output;
}

void software_1000()
{
  // Start performance counter
  PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE, 1);

  int i;
  for(i=0; i<240; i++)
  {
    out_1000[i] = software_custom_inst(in_1000[i]);
  }

  // Stop performance counter
  PERF_END(PERFORMANCE_COUNTER_0_BASE, 1);
}

void custom_1000()
{
  // Start performance counter
  PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE, 2);

  int i;
  for(i=0; i<240; i++)
  {
    out_1000[i] = ALT_CI_CUSTOM_INSTRUCTION_0(in_1000[i]);
  }

  // Stop performance counter
  PERF_END(PERFORMANCE_COUNTER_0_BASE, 2);
}

void use_accelerator(uint32_t in_addr, uint32_t out_addr, uint32_t length)
{
  // Check if the device is already working
  uint32_t status = IORD_32DIRECT(ACC_ADDR_BASE, 0x0);

  if(status == ACC_BUSY)
  {
    alt_printf("Busy\n");
    return;
  }
  alt_printf("Not busy\n");

  // Start performance counter
  PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE, 3);
```

```
  alt_printf("Writing arguments\n");

  // Write the three arguments
  IOWR_32DIRECT(ACC_ADDR_BASE, ACC_InputStartAddr, in_addr);
  IOWR_32DIRECT(ACC_ADDR_BASE, ACC_OutputStartAddr, out_addr);
  IOWR_32DIRECT(ACC_ADDR_BASE, ACC_Num, length);

  alt_printf("Waiting\n");
  // Wait for it to be done
  while(IORD_32DIRECT(ACC_ADDR_BASE, 0x0) != ACC_DONE);

  alt_printf("Done\n");
  // Stop performance counter
  PERF_END(PERFORMANCE_COUNTER_0_BASE, 3);
}

int test_method_once(uint32_t input, uint32_t exp_output, int method)
{
  // Tests any implementation once, returning 1 if it calculates the correct answer
  // and 0 otherwise
  // method = 1 -> C software implementation
  // method = 2 -> Custom instruction implementation
  // method = 3 -> Hardware accelerator implementation
  uint32_t result;
  if(method == 1)
  {
    result = software_custom_inst(input);
  }
  else if(method == 2)
  {
    result = ALT_CI_CUSTOM_INSTRUCTION_0(input);
  }
  else if(method == 3)
  {
    use_accelerator((uint32_t)&input, (uint32_t)&out_1000, 0xff);
    alt_printf("It's not stuck!\n");
    result = out_1000[0];
  }
  return result == exp_output;
}

void test_implementations()
{
  // Test software implementation
  alt_printf("Software implementation test 1: Expected is 1, it gives %x \n", test_method_once
  alt_printf("Software implementation test 2: Expected is 0, it gives %x \n", test_method_once

  // Test custom instruction implementation
  alt_printf("Custom instruction implementation test 1: Expected is 1, it gives %x \n", test_m
  alt_printf("Custom instruction implementation test 2: Expected is 0, it gives %x \n", test_m

  // Test accelerator implementation
  alt_printf("Accelerator implementation test 1: Expected is 1, it gives %x \n", test_method_o
```

```
  alt_printf("Accelerator implementation test 2: Expected is 0, it gives %x \n", test_method_o
}

int main()
{
  // Setup stuff for 1000 words of indata
  init_tables();
  void* in_addr_1000 = (uint32_t)&in_1000[0];
  void* out_addr_1000 = (uint32_t)&out_1000[0];
  uint32_t len_1000 = 0x000003E8;
  uint32_t len_8000 = 0x00001F40;

  // Test the implementations
  test_implementations();

  // Start overall performance counter
  PERF_RESET(PERFORMANCE_COUNTER_0_BASE);
  PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);

  // Run software implementation on 1000 words of indata
  software_1000();

  // Run custom instruction implementation on 1000 words of indata
  custom_1000();

  // Run accelerator implementation on 1000 words of indata
  use_accelerator(in_addr_1000, out_addr_1000, 2*len_1000);

  // Stop overall performance counter and print results
  PERF_STOP_MEASURING(PERFORMANCE_COUNTER_0_BASE);
  perf_print_formatted_report(PERFORMANCE_COUNTER_0_BASE, alt_get_cpu_freq(), 3, "C software",
}
```

4.2 Custom instruction

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity CustomSwitcher is
  port(
    InData  :  in std_logic_vector(31 downto 0);
    OutData  :  out std_logic_vector(31 downto 0)
  );
end CustomSwitcher;

architecture design of CustomSwitcher is
  begin
    OutData(31 downto 24) <= InData(7 downto 0);
    OutData(7 downto 0) <= InData(31 downto 24);
    bitswap: for i in 8 to 23 generate
      OutData(31 - i) <= InData(i);
    end generate bitswap;
```

```
end design;
```

4.3 Accelerator v1

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity DumbestHwAccelerator is
  port(
    -- Main signals
    Clk             :      IN    STD_LOGIC;
    Reset_n         :      IN    STD_LOGIC;
    -- Avalon Master Interface
    AM_BE           :      OUT   STD_LOGIC_VECTOR(3 downto 0);
    AM_Addr         :      OUT   STD_LOGIC_VECTOR(31 downto 0);
    AM_Rd           :      OUT   STD_LOGIC;
    AM_Wr           :      OUT   STD_LOGIC;
    AM_WData        :      OUT   STD_LOGIC_VECTOR(31 downto 0);
    AM_WaitRq       :      IN    STD_LOGIC;
    AM_RDataValid   :      IN    STD_LOGIC;
    AM_RData        :      IN    STD_LOGIC_VECTOR(31 downto 0);
    -- Avalon slave interface
    AS_Addr         :      IN    STD_LOGIC_VECTOR(1 downto 0);
    AS_Wr           :      IN    STD_LOGIC;
    AS_WData        :      IN    STD_LOGIC_VECTOR(31 downto 0);
    As_Rd           :      IN    STD_LOGIC;
    AS_RData        :      OUT   STD_LOGIC_VECTOR(31 downto 0);


    -- Debug PIO output signal
    DEBUGSTATE      :      OUT   STD_LOGIC_VECTOR(7 downto 0);
    DEBUGCOUNT      :      OUT   STD_LOGIC_VECTOR(9 downto 0)
  );
end DumbestHwAccelerator;

architecture comp of DumbestHwAccelerator is
  -- FSM stuff
  TYPE State_type is (Idle, DMALoadRequest, DMALoadOngoing, DMALoadReceive, Operation, DMAStor
  SIGNAL  State     :    State_type;
  -- Internal registers
  SIGNAL  iReadAddr, iStoreAddr  :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iInAddr, iOutAddr      :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iNum        :  unsigned(9 downto 0); -- Range 0 to 1023
  SIGNAL  iCurrentCount      :  unsigned(9 downto 0); -- Range 0 to 1023
  SIGNAL  iInData     :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iOutData    :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iStart      :  STD_LOGIC_VECTOR(2 downto 0);
  SIGNAL  iStop       :  STD_LOGIC;
  SIGNAL   iWorking   :  STD_LOGIC;

begin
    -- Drives Avalon master interface, and basically everything except slave stuff
```

```
-- Drives DEBUG
FSMMAIN: process(Clk, Reset_n)
  begin
    if Reset_n = '0' then
      DEBUGSTATE <= (others => '1');
      State <= Idle;
      iInAddr <= (others => '0');
      iOutAddr <= (others => '0');
      iCurrentCount <= (others => '0');
      iInData <= (others => '0');
      iOutData <= (others => '0');
      iStop <= '0';
      AM_Wr <= '0';
      AM_Addr <= (others => '0');
      AM_BE <= (others => '1');
      AM_WData <= (others => '0');
      AM_Rd <= '0';
    elsif rising_edge(Clk) then
      case State is
        when Idle =>
          DEBUGSTATE <= "00000000";
          DEBUGCOUNT <= std_logic_vector(iCurrentCount);

          AM_Wr <= '0';
          AM_Addr <= (others => '0');
          AM_BE <= (others => '1');
          AM_WData <= (others => '0');
          AM_Rd <= '0';
          iCurrentCount <= (others => '0');
          iWorking <= '0';
          iStop <= '0';
          -- Start the process when all three arguments have been received
          if iStart = "111" then
            State <= DMALoadRequest;
            iInAddr <= iReadAddr;
            iOutAddr <= iStoreAddr;
          end if;
        when DMALoadRequest =>
          DEBUGSTATE <= "00000001";
          DEBUGCOUNT <= std_logic_vector(iCurrentCount);
          iWorking <= '1';
          -- Initialize request to read the next indata word from memory
          AM_Rd <= '1';
          AM_BE <= (others => '1');
          AM_Addr <= iInAddr;
          -- Move on if the read request was granted
          State <= DMALoadOngoing;
        when DMALoadOngoing =>
          DEBUGSTATE <= "00000010";
          DEBUGCOUNT <= std_logic_vector(iCurrentCount);
          iWorking <= '1';
          -- Continue requesting to read the next indata word from memory
          AM_Rd <= '1';
```

```
      AM_BE <= (others => '1');
      AM_Addr <= iInAddr;
      -- Move on if the read request was granted
      if AM_WaitRq = '0' then
        State <= DMALoadReceive;
        AM_Rd <= '0';
      end if;
    when DMALoadReceive =>
      DEBUGSTATE <= "00000100";
      DEBUGCOUNT <= std_logic_vector(iCurrentCount);
      iWorking <= '1';
      -- Wait for the read data to be received, then move on
      if AM_RDataValid = '1' then
        iInData <= AM_RData;
        State <= Operation;
      end if;
    when Operation =>
      DEBUGSTATE <= "00001000";
      DEBUGCOUNT <= std_logic_vector(iCurrentCount);
      iWorking <= '1';
      -- Perform the operation, then move on
      iOutData(31 downto 24) <= iInData(7 downto 0);
      iOutData(7 downto 0) <= iInData(31 downto 24);
      for i in 8 to 23 loop
        iOutData(31 - i) <= iInData(i);
      end loop;
      iCurrentCount <= iCurrentCount + 1;
      State <= DMAStoreRequest;
    when DMAStoreRequest =>
      DEBUGSTATE <= "00010000";
      DEBUGCOUNT <= std_logic_vector(iCurrentCount);
      iWorking <= '1';
      -- Request to store the outdata in memory
      AM_Wr <= '1';
      AM_Addr <= iOutAddr;
      AM_BE <= (others => '1');
      AM_WData <= iOutData;
      State <= DMAStoreOngoing;
    when DMAStoreOngoing =>
      DEBUGSTATE <= "00100000";
      DEBUGCOUNT <= std_logic_vector(iCurrentCount);
      iWorking <= '1';
      -- Continue requesting to store the outdata in memory
      AM_Wr <= '1';
      AM_Addr <= iOutAddr;
      AM_BE <= (others => '1');
      AM_WData <= iOutData;
      -- Move on when the request is granted
      if AM_WaitRq = '0' then
        State <= CheckDone;
        AM_Wr <= '0';
      end if;
    when CheckDone =>
```

```
            DEBUGSTATE <= "01000000";
            DEBUGCOUNT <= std_logic_vector(iCurrentCount);

            iInAddr <= std_logic_vector(unsigned(iInAddr) + 4);
            iOutAddr <= std_logic_vector(unsigned(iOutAddr) + 4);
            iWorking <= '1';
            -- Check if the process is done
            if iCurrentCount = iNum then
              -- Move to Done
              State <= Done;
              iStop <= '1';
            else
              -- Proceed with next word
              State <= DMALoadRequest;
            end if;
          when Done =>
            DEBUGSTATE <= "10000000";
            DEBUGCOUNT <= std_logic_vector(iCurrentCount);
            -- This state delays 1 clock cycle for the flags to be reset, before returning t
            -- Also keeps iStop high for one extra clock cycle
            State <= Idle;
            iStop <= '1';
            iWorking <= '0';
          when others => null;
        end case;
      end if;
  end process FSMMAIN;

  -- Avalon slave write
  -- Drives Avalon slave write interface, iStart, iNum
  AS_WriteProc: process(Clk, Reset_n) is
  begin
    if Reset_n = '0' then
      iStart <= (others => '0');
      iReadAddr <= (others => '0');
      iStoreAddr <= (others => '0');
      iNum <= (others => '0');
    elsif rising_edge(Clk) then
      -- Check for stopping condition
      if iStop = '1' then
        iStart <= (others => '0');
      end if;
      if AS_Wr = '1' then
        case AS_Addr is
          when "00" =>
            iReadAddr <= AS_WData;
            iStart(0) <= '1';
          when "01" =>
            iStoreAddr <= AS_WData;
            iStart(1) <= '1';
          when "10" =>
            iNum <= unsigned(AS_WData(9 downto 0));
            iStart(2) <= '1';
```

```
          when others => null;
        end case;
      end if;
    end if;
  end process AS_WriteProc;

  -- Avalon slave read
  -- Drives Avalon slave read interface
  AS_ReadProc: process(Clk, Reset_n) is
  begin
    if Reset_n = '0' then
      AS_RData <= (others => '0');
    elsif rising_edge(Clk) then
      AS_RData <= (others => '0');
      if(AS_Rd = '1') then
        AS_RData(0) <= iStop;
        AS_RData(1) <= iWorking;
      end if;
    end if;
  end process AS_ReadProc;

end comp;
```

4.4 Accelerator v2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity DumbHwAccelerator is
  port(
    -- Main signals
    Clk             :      IN    STD_LOGIC;
    Reset_n         :      IN    STD_LOGIC;
    -- Avalon Master Interface
    AM_BE           :      OUT   STD_LOGIC_VECTOR(3 downto 0);
    AM_Addr         :      OUT   STD_LOGIC_VECTOR(31 downto 0);
    AM_Rd           :      OUT   STD_LOGIC;
    AM_Wr           :      OUT   STD_LOGIC;
    AM_WData        :      OUT   STD_LOGIC_VECTOR(31 downto 0);
    AM_WaitRq       :      IN    STD_LOGIC;
    AM_RDataValid   :      IN    STD_LOGIC;
    AM_RData        :      IN    STD_LOGIC_VECTOR(31 downto 0);
    -- Avalon slave interface
    AS_Addr         :      IN    STD_LOGIC_VECTOR(1 downto 0);
    AS_Wr           :      IN    STD_LOGIC;
    AS_WData        :      IN    STD_LOGIC_VECTOR(31 downto 0);
    As_Rd           :      IN    STD_LOGIC;
    AS_RData        :      OUT   STD_LOGIC_VECTOR(31 downto 0);


    -- Debug PIO output signal
    DEBUGSTATE      :      OUT   STD_LOGIC_VECTOR(7 downto 0);
```

```
    DEBUGCOUNT        :      OUT  STD_LOGIC_VECTOR(9 downto 0)
  );
end DumbHwAccelerator;

architecture comp of DumbHwAccelerator is
  -- FSM stuff
  TYPE State_type is (Idle, DMALoadRequest, DMALoadOngoing, DMALoadReceive, Operation, DMAStor
  SIGNAL  State      :    State_type;
  -- Internal registers
  SIGNAL  iStartAddr, iStoreAddr  :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iNum        :  unsigned(9 downto 0); -- Range 0 to 1023
  SIGNAL  iCurrentCount      :  unsigned(9 downto 0); -- Range 0 to 1023
  SIGNAL  iInData      :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iOutData      :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iStart      :  STD_LOGIC_VECTOR(2 downto 0);
  SIGNAL  iStop        :  STD_LOGIC;
  SIGNAL   iWorking      :  STD_LOGIC;

begin
    -- Drives Avalon master interface, Next_State, iStop
    -- Drives DEBUG
    FSMMAIN: process(Clk, Reset_n)
      begin
        -- Don't stop by default
        iStop <= '0';
        -- Don't work by default
        iWorking <= '0';
        -- Don't use master interface by default
        AM_Wr <= '0';
        AM_Addr <= (others => '0');
        AM_BE <= (others => '1');
        AM_WData <= (others => '0');
        AM_Rd <= '0';
        if Reset_n = '0' then
          DEBUGSTATE <= (others => '1');
          iStop <= '0';
          AM_Wr <= '0';
          AM_Addr <= (others => '0');
          AM_BE <= (others => '1');
          AM_WData <= (others => '0');
          AM_Rd <= '0';
        elsif rising_edge(Clk) then
          case State is
            when Idle =>
              DEBUGSTATE <= "00000000";
              DEBUGCOUNT <= std_logic_vector(iCurrentCount);
              iWorking <= '0';
              -- Start the process when all three arguments have been received
              if iStart = "111" then
                State <= DMALoadRequest;
              end if;
            when DMALoadRequest =>
              DEBUGSTATE <= "00000001";
```

```
    DEBUGCOUNT <= std_logic_vector(iCurrentCount);
    iWorking <= '1';
    -- Initialize request to read the next indata word from memory
    AM_Rd <= '1';
    AM_BE <= (others => '1');
    AM_Addr <= std_logic_vector(unsigned(iStartAddr) + iCurrentCount*4);
    -- Move on if the read request was granted
    State <= DMALoadOngoing;
  when DMALoadOngoing =>
    DEBUGSTATE <= "00000010";
    DEBUGCOUNT <= std_logic_vector(iCurrentCount);
    iWorking <= '1';
    -- Continue requesting to read the next indata word from memory
    AM_Rd <= '1';
    AM_BE <= (others => '1');
    AM_Addr <= std_logic_vector(unsigned(iStartAddr) + iCurrentCount*4);
    -- Move on if the read request was granted
    if AM_WaitRq = '0' then
      State <= DMALoadReceive;
    end if;
  when DMALoadReceive =>
    DEBUGSTATE <= "00000100";
    iWorking <= '1';
    -- Wait for the read data to be received, then move on
    if AM_RDataValid = '1' then
      iInData <= AM_RData;
      State <= Operation;
    end if;
  when Operation =>
    DEBUGSTATE <= "00001000";
    DEBUGCOUNT <= std_logic_vector(iCurrentCount);
    iWorking <= '1';
    -- Perform the operation, then move on
    iOutData(31 downto 24) <= iInData(7 downto 0);
    iOutData(7 downto 0) <= iInData(31 downto 24);
    for i in 8 to 23 loop
      iOutData(31 - i) <= iInData(i);
    end loop;
    State <= DMAStoreRequest;
  when DMAStoreRequest =>
    DEBUGSTATE <= "00010000";
    DEBUGCOUNT <= std_logic_vector(iCurrentCount);
    iWorking <= '1';
    -- Request to store the outdata in memory
    AM_Wr <= '1';
    AM_Addr <= std_logic_vector(unsigned(iStoreAddr) + iCurrentCount*4);
    AM_BE <= (others => '1');
    AM_WData <= iOutData;
    State <= DMAStoreOngoing;
  when DMAStoreOngoing =>
    DEBUGSTATE <= "00100000";
    DEBUGCOUNT <= std_logic_vector(iCurrentCount);
    iWorking <= '1';
```

```
            -- Continue requesting to store the outdata in memory
            AM_Wr <= '1';
            AM_Addr <= std_logic_vector(unsigned(iStoreAddr) + iCurrentCount*4);
            AM_BE <= (others => '1');
            AM_WData <= iOutData;
            -- Move on when the request is granted
            if AM_WaitRq = '0' then
              --iCurrentCount <= iCurrentCount + 1;
              State <= CheckDone;
            end if;
          when CheckDone =>
            DEBUGSTATE <= "01000000";
            DEBUGCOUNT <= std_logic_vector(iCurrentCount);
            iWorking <= '1';
            -- Check if the process is done
            if iCurrentCount = (iNum - 1) then
              -- Move to Done
              State <= Done;
              iStop <= '1';
            else
              -- Proceed with next word
              State <= DMALoadRequest;
            end if;
          when Done =>
            DEBUGSTATE <= "10000000";
            DEBUGCOUNT <= std_logic_vector(iCurrentCount);
            -- This state delays 1 clock cycle for the flags to be reset, before returning t
            -- Also keeps iStop high for one extra clock cycle
            State <= Idle;
            iStop <= '1';
            iWorking <= '0';
          when others => null;
        end case;
      end if;
  end process FSMMAIN;

  -- Counter update process
  CountProc: process(State, Reset_n) is
  begin
    if Reset_n = '0' then
      iCurrentCount <= (others => '0');
    else
      case State is
        when Idle =>
          iCurrentCount <= (others => '0');
        when CheckDone =>
          iCurrentCount <= iCurrentCount + 1;
        when others =>
          iCurrentCount <= iCurrentCount;
      end case;
    end if;
  end process CountProc;
```

```
    -- Avalon slave write
    -- Drives Avalon slave write interface, iStart
    AS_WriteProc: process(Clk, Reset_n, iStop) is
    begin
      if Reset_n = '0' then
        iStart <= (others => '0');
      elsif rising_edge(Clk) then
        -- Check for stopping condition
        if iStop = '1' then
          iStart <= (others => '0');
        end if;
        if AS_Wr = '1' then
          case AS_Addr is
            when "00" =>
              iStartAddr <= AS_WData;
              iStart(0) <= '1';
            when "01" =>
              iStoreAddr <= AS_WData;
              iStart(1) <= '1';
            when "10" =>
              iNum <= unsigned(AS_WData(9 downto 0));
              iStart(2) <= '1';
            when others => null;
          end case;
        end if;
      end if;
    end process AS_WriteProc;

    -- Avalon slave read
    -- Drives Avalon slave read interface
    AS_ReadProc: process(Clk, Reset_n) is
    begin
      if Reset_n = '0' then
        AS_RData <= (others => '0');
      elsif rising_edge(Clk) then
        AS_RData <= (others => '0');
        if(AS_Rd = '1') then
          AS_RData(0) <= iStop;
          AS_RData(1) <= iWorking;
        end if;
      end if;
    end process AS_ReadProc;

end comp;
```

4.5 Accelerator v3

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity HwAccelerator is
  port(
```

```
    -- Main signals
    Clk             :      IN    STD_LOGIC;
    Reset_n         :      IN    STD_LOGIC;
    -- Avalon Master Interface
    AM_BE           :      OUT   STD_LOGIC_VECTOR(3 downto 0);
    AM_Addr         :      OUT   STD_LOGIC_VECTOR(31 downto 0);
    AM_Rd           :      OUT   STD_LOGIC;
    AM_Wr           :      OUT   STD_LOGIC;
    AM_WData        :      OUT   STD_LOGIC_VECTOR(31 downto 0);
    AM_WaitRq       :      IN    STD_LOGIC;
    AM_RDataValid   :      IN    STD_LOGIC;
    AM_RData        :      IN    STD_LOGIC_VECTOR(31 downto 0);
    -- Avalon slave interface
    AS_Addr         :      IN    STD_LOGIC_VECTOR(1 downto 0);
    AS_Wr           :      IN    STD_LOGIC;
    AS_WData        :      IN    STD_LOGIC_VECTOR(31 downto 0);
    As_Rd           :      IN    STD_LOGIC;
    AS_RData        :      OUT   STD_LOGIC_VECTOR(31 downto 0);


    -- Debug PIO output signal
    DEBUGSTATE      :      OUT   STD_LOGIC_VECTOR(7 downto 0);
    DEBUGCOUNT      :      OUT   STD_LOGIC_VECTOR(9 downto 0)
  );
end HwAccelerator;

architecture comp of HwAccelerator is
  -- FSM stuff
  TYPE State_type is (Idle, DMALoadRequest, DMALoadOngoing, DMALoadReceive, Operation, DMAStor
  SIGNAL  State, Next_State      :    State_type;
  -- Internal registers
  SIGNAL  iStartAddr, iStoreAddr  :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iNum        :  unsigned(9 downto 0); -- Range 0 to 1023
  SIGNAL  iCount, iCountNext      :  unsigned(9 downto 0); -- Range 0 to 1023
  SIGNAL  iInData     :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iOutData    :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iStart      :  STD_LOGIC_VECTOR(2 downto 0);
  SIGNAL  iStop       :  STD_LOGIC;
  SIGNAL   iWorking   :  STD_LOGIC;

begin
    -- Drives Avalon master interface, Next_State, iCountNext, iStop
    -- Drives DEBUG
    FSMMAIN: process(State, Reset_n, iStart, AM_WaitRq, AM_RDataValid)
      begin
        -- Stay in the current state unless the continue condition is met
        Next_State <= State;
        -- Keep the current count as default
        --iCountNext <= iCount;
        iCount <= iCount;
        -- Don't stop by default
        iStop <= '0';
        -- Don't work by default
```

```
iWorking <= '0';
-- Don't use master interface by default
AM_Wr <= '0';
AM_Addr <= (others => '0');
AM_BE <= (others => '1');
AM_WData <= (others => '0');
AM_Rd <= '0';
if Reset_n = '0' then
  DEBUGSTATE <= (others => '1');
  Next_State <= Idle;
  --iCountNext <= (others => '0');
  iCount <= (others => '0');
  iStop <= '0';
  AM_Wr <= '0';
  AM_Addr <= (others => '0');
  AM_BE <= (others => '1');
  AM_WData <= (others => '0');
  AM_Rd <= '0';
else
  case State is
    when Idle =>
      DEBUGSTATE <= "00000000";
      iWorking <= '0';
      -- Start the process when all three arguments have been received
      if iStart = "111" then
        Next_State <= DMALoadRequest;
      end if;
    when DMALoadRequest =>
      DEBUGSTATE <= "00000001";
      iWorking <= '1';
      -- Initialize request to read the next indata word from memory
      AM_Rd <= '1';
      AM_BE <= (others => '1');
      AM_Addr <= std_logic_vector(unsigned(iStartAddr) + iCount*4);
      -- Move on if the read request was granted
      Next_State <= DMALoadOngoing;
    when DMALoadOngoing =>
      DEBUGSTATE <= "00000010";
      iWorking <= '1';
      -- Continue requesting to read the next indata word from memory
      AM_Rd <= '1';
      AM_BE <= (others => '1');
      AM_Addr <= std_logic_vector(unsigned(iStartAddr) + iCount*4);
      -- Move on if the read request was granted
      if AM_WaitRq = '0' then
        Next_State <= DMALoadReceive;
      end if;
    when DMALoadReceive =>
      DEBUGSTATE <= "00000100";
      iWorking <= '1';
      -- Wait for the read data to be received, then move on
      if AM_RDataValid = '1' then
        iInData <= AM_RData;
```

```
      Next_State <= Operation;
    end if;
  when Operation =>
    DEBUGSTATE <= "00001000";
    iWorking <= '1';
    -- Perform the operation, then move on
    iOutData(31 downto 24) <= iInData(7 downto 0);
    iOutData(7 downto 0) <= iInData(31 downto 24);
    for i in 8 to 23 loop
      iOutData(31 - i) <= iInData(i);
    end loop;
    Next_State <= DMAStoreRequest;
  when DMAStoreRequest =>
    DEBUGSTATE <= "00010000";
    iWorking <= '1';
    -- Request to store the outdata in memory
    AM_Wr <= '1';
    AM_Addr <= std_logic_vector(unsigned(iStoreAddr) + iCount*4);
    AM_BE <= (others => '1');
    AM_WData <= iOutData;
    Next_State <= DMAStoreOngoing;
  when DMAStoreOngoing =>
    DEBUGSTATE <= "00100000";
    iWorking <= '1';
    -- Continue requesting to store the outdata in memory
    AM_Wr <= '1';
    AM_Addr <= std_logic_vector(unsigned(iStoreAddr) + iCount*4);
    AM_BE <= (others => '1');
    AM_WData <= iOutData;
    -- Move on when the request is granted
    if AM_WaitRq = '0' then
      --iCountNext <= iCount + 1;
      iCount <= iCount + 1;
      Next_State <= CheckDone;
    end if;
  when CheckDone =>
    DEBUGSTATE <= "01000000";
    iWorking <= '1';
    -- Check if the process is done
    if iCount = iNum then
      -- Move to Done
      --iCountNext <= (others => '0');
      iCount <= (others => '0');
      Next_State <= Done;
      iStop <= '1';
    else
      -- Proceed with next word
      Next_State <= DMALoadRequest;
    end if;
  when Done =>
    DEBUGSTATE <= "10000000";
    -- This state delays 1 clock cycle for the flags to be reset, before returning t
    -- Also keeps iStop high for one extra clock cycle
```

```
          Next_State <= Idle;
          iStop <= '1';
          iWorking <= '0';
        when others => null;
      end case;
    end if;
end process FSMMAIN;


-- State machine update process
-- Drives State, iCount
FSMUpdate: process(Clk, Reset_n) is
  Begin
    if Reset_n = '0' then
      DEBUGCOUNT <= "1111111111";
      State <= Idle;
      --iCount <= (others => '0');
    elsif rising_edge(Clk) then
      DEBUGCOUNT <= std_logic_vector(iCount);
      State <= Next_State;
      --iCount <= iCountNext;
    end if;
end process FSMUpdate;


-- Avalon slave write
-- Drives Avalon slave write interface, iStart
AS_WriteProc: process(Clk, Reset_n, iStop) is
begin
  if Reset_n = '0' then
    iStart <= (others => '0');
  elsif rising_edge(Clk) then
    -- Check for stopping condition
    if iStop = '1' then
      iStart <= (others => '0');
    end if;
    if AS_Wr = '1' then
      case AS_Addr is
        when "00" =>
          iStartAddr <= AS_WData;
          iStart(0) <= '1';
        when "01" =>
          iStoreAddr <= AS_WData;
          iStart(1) <= '1';
        when "10" =>
          iNum <= unsigned(AS_WData(9 downto 0));
          iStart(2) <= '1';
        when others => null;
      end case;
    end if;
  end if;
end process AS_WriteProc;
```

```
    -- Avalon slave read
    -- Drives Avalon slave read interface
    AS_ReadProc: process(Clk, Reset_n) is
    begin
      if Reset_n = '0' then
        AS_RData <= (others => '0');
      elsif rising_edge(Clk) then
        AS_RData <= (others => '0');
        if(AS_Rd = '1') then
          AS_RData(0) <= iStop;
          AS_RData(1) <= iWorking;
        end if;
      end if;
    end process AS_ReadProc;

end comp;
```

4.6 Accelerator v4

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity HwAccelerator2 is
  port(
    -- Main signals
    Clk             :       IN    STD_LOGIC;
    Reset_n         :       IN    STD_LOGIC;
    -- Avalon Master Interface
    AM_BE           :       OUT   STD_LOGIC_VECTOR(3 downto 0);
    AM_Addr         :       OUT   STD_LOGIC_VECTOR(31 downto 0);
    AM_Rd           :       OUT   STD_LOGIC;
    AM_Wr           :       OUT   STD_LOGIC;
    AM_WData        :       OUT   STD_LOGIC_VECTOR(31 downto 0);
    AM_WaitRq       :       IN    STD_LOGIC;
    AM_RDataValid   :       IN    STD_LOGIC;
    AM_RData        :       IN    STD_LOGIC_VECTOR(31 downto 0);
    -- Avalon slave interface
    AS_Addr         :       IN    STD_LOGIC_VECTOR(1 downto 0);
    AS_Wr           :       IN    STD_LOGIC;
    AS_WData        :       IN    STD_LOGIC_VECTOR(31 downto 0);
    As_Rd           :       IN    STD_LOGIC;
    AS_RData        :       OUT   STD_LOGIC_VECTOR(31 downto 0);


    -- Debug PIO output signal
    DEBUGSTATE      :       OUT   STD_LOGIC_VECTOR(7 downto 0);
    DEBUGCOUNT      :       OUT   STD_LOGIC_VECTOR(15 downto 0)
  );
end HwAccelerator2;

architecture comp of HwAccelerator2 is
  -- FSM stuff
```

```
  TYPE State_type is (Idle, DMALoadRequest, DMALoadOngoing, DMALoadReceive, Operation, DMAStor
  SIGNAL  State, State_Next      :    State_type;
  -- Internal registers
  SIGNAL  iStartAddr, iStoreAddr, iStartAddr_Next, iStoreAddr_Next  :  STD_LOGIC_VECTOR(31 dow
  SIGNAL  iNum, iNum_Next        :  unsigned(15 downto 0); -- Range 0 to 1023
  SIGNAL  iCount, iCount_Next    :  unsigned(15 downto 0); -- Range 0 to 1023
  SIGNAL  iInData, iInData_Next      :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iOutData, iOutData_Next      :  STD_LOGIC_VECTOR(31 downto 0);
  SIGNAL  iStart, iStart_Next    :  STD_LOGIC_VECTOR(2 downto 0);
  SIGNAL  iStop, iStop_Next      :  STD_LOGIC;
  SIGNAL   iWorking, iWorking_Next      :  STD_LOGIC;

begin
    Next_Process: process(Clk, Reset_n)
    begin
    if Reset_n = '0' then
      DEBUGCOUNT <= "1111111111111111";
      State <= Idle;
      iStartAddr <= (others => '0');
      iNum <= (others => '0');
      iInData <= (others => '0');
      iOutData <= (others => '0');
      iStart <= (others => '0');
      iCount <= (others => '0');
      iStop <= '0';
      iWorking <= '0';
    elsif rising_edge(Clk) then
      DEBUGCOUNT <= std_logic_vector(iCount);

      State <= State_Next;
      iStartAddr <= iStartAddr_Next;
      iStoreAddr <= iStoreAddr_Next;
      iNum <= iNum_Next;
      iCount <= iCount_Next;
      iInData <= iInData_Next;
      iOutData <= iOutData_Next;
      iStart <= iStart_Next;
      iStop <= iStop_Next;
      iWorking <= iWorking_Next;
    end if;
    end process Next_Process;


    -- internal signal process
    Signal_Proc: process(State, Reset_n)
      begin
        iCount_Next <= iCount;
        iStop_Next <= iStop;
        iWorking_Next <= iWorking;
        iInData_Next <= iInData;
        iOutData_Next <= iOutData;
        if Reset_n = '0' then
          iCount_Next <= (others => '0');
```

```
        iStop_Next <= '0';
      else
        case State is
          when Idle =>
            iWorking_Next <= '0';
          when DMALoadRequest =>
            iWorking_Next <= '1';

          when DMALoadOngoing =>
            iWorking_Next <= '1';
          when DMALoadReceive =>
            iWorking_Next <= '1';
            if AM_RDataValid = '1' then
              iInData_Next <= AM_RData;
            end if;
          when Operation =>
            iWorking_Next <= '1';
            -- Perform the operation, then move on
            iOutData_Next(31 downto 24) <= iInData(7 downto 0);
            iOutData_Next(7 downto 0) <= iInData(31 downto 24);
            for i in 8 to 23 loop
              iOutData_Next(31 - i) <= iInData(i);
            end loop;
          when DMAStoreRequest =>
            iWorking_Next <= '1';
          when DMAStoreOngoing =>
            iWorking_Next <= '1';
          when CheckDone =>
            iWorking_Next <= '1';
            iCount_Next <= iCount + 1;
            if iCount = (iNum - 1) then
              -- Move to Done
              iCount_Next <= (others => '0');
              iStop_Next <= '1';
            end if;
          when Done =>
            iWorking_Next <= '0';
            iStop_Next <= '1';
          when others => null;
        end case;
      end if;
    end process Signal_Proc;

  -- Drives Avalon master interface, State_Next
  -- Drives DEBUG
  FSMMAIN: process(State, Reset_n)
    begin
      -- Keep current values unless a change is called for
      State_Next <= State;
      -- Don't use master interface by default
      AM_Wr <= '0';
      AM_Addr <= (others => '0');
      AM_BE <= (others => '1');
```

```
AM_WData <= (others => '0');
AM_Rd <= '0';
if Reset_n = '0' then
  DEBUGSTATE <= (others => '1');
  State_Next <= Idle;
  AM_Wr <= '0';
  AM_Addr <= (others => '0');
  AM_BE <= (others => '1');
  AM_WData <= (others => '0');
  AM_Rd <= '0';
else
  case State is
    when Idle =>
      DEBUGSTATE <= "00000000";
      -- Start the process when all three arguments have been received
      if iStart = "111" then
        State_Next <= DMALoadRequest;
      end if;
    when DMALoadRequest =>
      DEBUGSTATE <= "00000001";
      -- Initialize request to read the next indata word from memory
      AM_Rd <= '1';
      AM_BE <= (others => '1');
      AM_Addr <= std_logic_vector(unsigned(iStartAddr) + iCount*4);
      -- Move on if the read request was granted
      State_Next <= DMALoadOngoing;
    when DMALoadOngoing =>
      DEBUGSTATE <= "00000010";
      -- Continue requesting to read the next indata word from memory
      AM_Rd <= '1';
      AM_BE <= (others => '1');
      AM_Addr <= std_logic_vector(unsigned(iStartAddr) + iCount*4);
      -- Move on if the read request was granted
      if AM_WaitRq = '0' then
        State_Next <= DMALoadReceive;
      end if;
    when DMALoadReceive =>
      DEBUGSTATE <= "00000100";
      -- Wait for the read data to be received, then move on
      if AM_RDataValid = '1' then
        State_Next <= Operation;
      end if;
    when Operation =>
      DEBUGSTATE <= "00001000";
      State_Next <= DMAStoreRequest;
    when DMAStoreRequest =>
      DEBUGSTATE <= "00010000";
      -- Request to store the outdata in memory
      AM_Wr <= '1';
      AM_Addr <= std_logic_vector(unsigned(iStoreAddr) + iCount*4);
      AM_BE <= (others => '1');
      AM_WData <= iOutData;
      State_Next <= DMAStoreOngoing;
```

```
          when DMAStoreOngoing =>
            DEBUGSTATE <= "00100000";
            -- Continue requesting to store the outdata in memory
            AM_Wr <= '1';
            AM_Addr <= std_logic_vector(unsigned(iStoreAddr) + iCount*4);
            AM_BE <= (others => '1');
            AM_WData <= iOutData;
            -- Move on when the request is granted
            if AM_WaitRq = '0' then
              State_Next <= CheckDone;
            end if;
          when CheckDone =>
            DEBUGSTATE <= "01000000";
            -- Check if the process is done
            if iCount = (iNum - 1) then
              -- Move to Done
              State_Next <= Done;
            else
              -- Proceed with next word
              State_Next <= DMALoadRequest;
            end if;
          when Done =>
            DEBUGSTATE <= "10000000";
            -- This state delays 1 clock cycle for the flags to be reset, before returning t
            -- Also keeps iStop high for one extra clock cycle
            State_Next <= Idle;
          when others => null;
        end case;
      end if;
  end process FSMMAIN;


  -- Avalon slave write
  -- Drives Avalon slave write interface, iStart_Next, iStartAddr_Next, iStoreAddr_Next, iNu
  AS_WriteProc: process(Clk, Reset_n, iStop, iStartAddr, iStoreAddr, iNum, iStart) is
  begin
    if Reset_n = '0' then
      iStart_Next <= (others => '0');
    elsif rising_edge(Clk) then
      --iStartAddr_Next <= iStartAddr;
      --iStoreAddr_Next <= iStoreAddr;
      --iNum_Next <= iNum;
      --iStart_Next <= iStart;
      -- Check for stopping condition
      if iWorking = '1' then
        iStart_Next <= (others => '0');
      end if;
      if AS_Wr = '1' then
        case AS_Addr is
          when "00" =>
            iStartAddr_Next <= AS_WData;
            iStart_Next(0) <= '1';
          when "01" =>
```

```
            iStoreAddr_Next <= AS_WData;
            iStart_Next(1) <= '1';
          when "10" =>
            iNum_Next <= unsigned(AS_WData(15 downto 0));
            iStart_Next(2) <= '1';
          when others => null;
        end case;
      end if;
    end if;
  end process AS_WriteProc;

  -- Avalon slave read
  -- Drives Avalon slave read interface
  AS_ReadProc: process(Clk, Reset_n) is
  begin
    if Reset_n = '0' then
      AS_RData <= (others => '0');
    elsif rising_edge(Clk) then
      AS_RData <= (others => '0');
      if(AS_Rd = '1') then
        AS_RData(0) <= iStop;
        AS_RData(1) <= iWorking;
      end if;
    end if;
  end process AS_ReadProc;

end comp;
```

5. Annex 2: Logic analyzer measurements

In the following figures, the channels correspond to the following:

Channel number	Information
5	DMAStoreOngoing
6	CheckDone
7	Done
8	Counter, bit 0
9	Counter, bit 1
10	Counter, bit 2

5.1 Accelerator v1

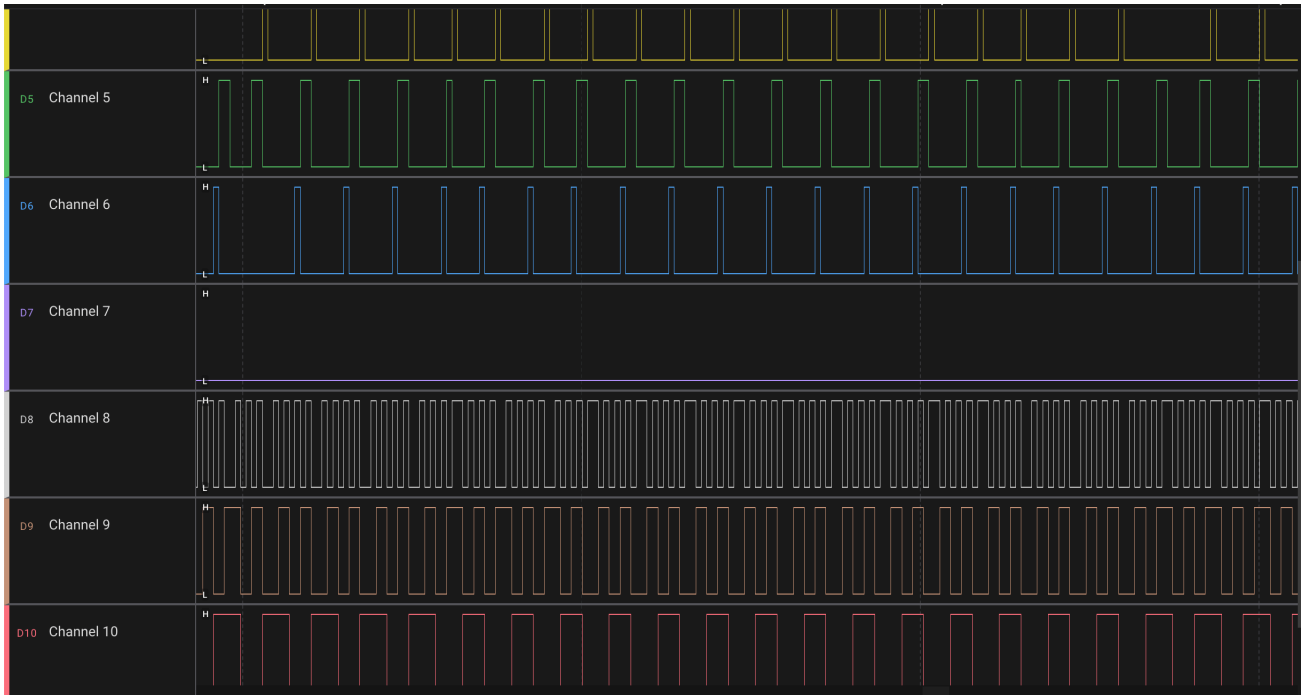

Figure 8: Accelerator v1

5.2 Accelerator v2

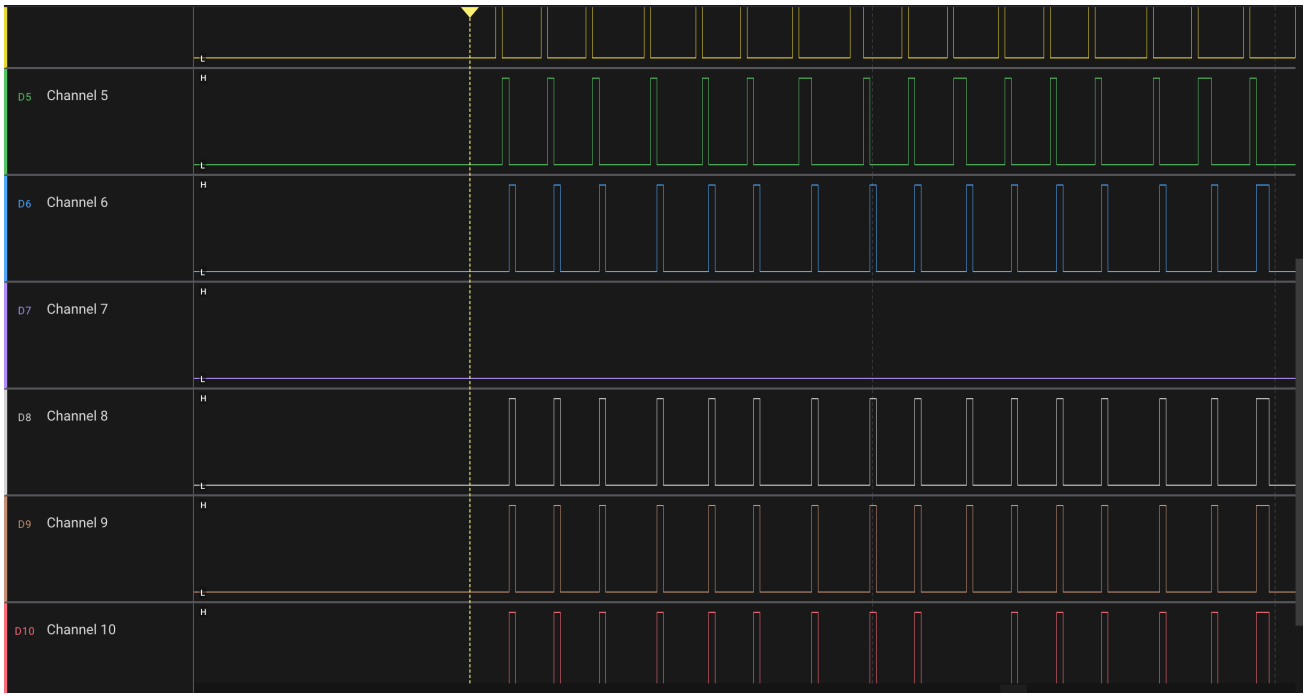

Figure 9: Accelerator v2

5.3 Accelerator v3

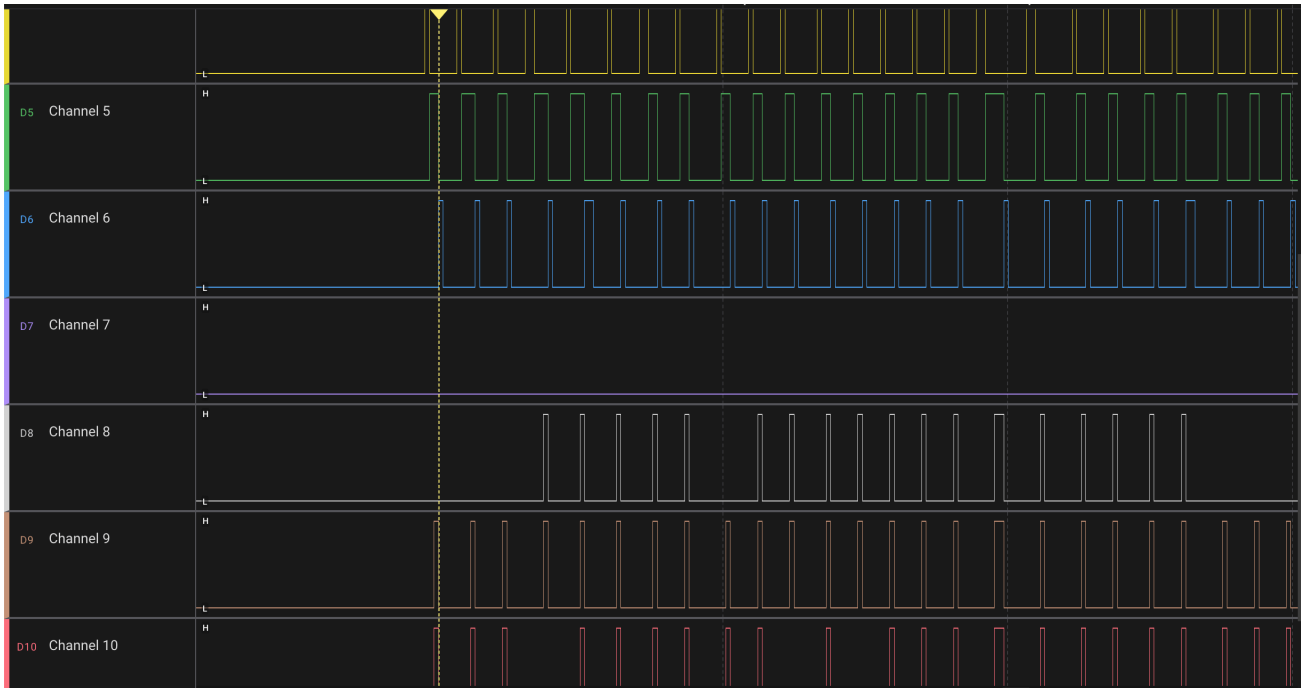

Figure 10: Accelerator v3

5.4 Accelerator v4

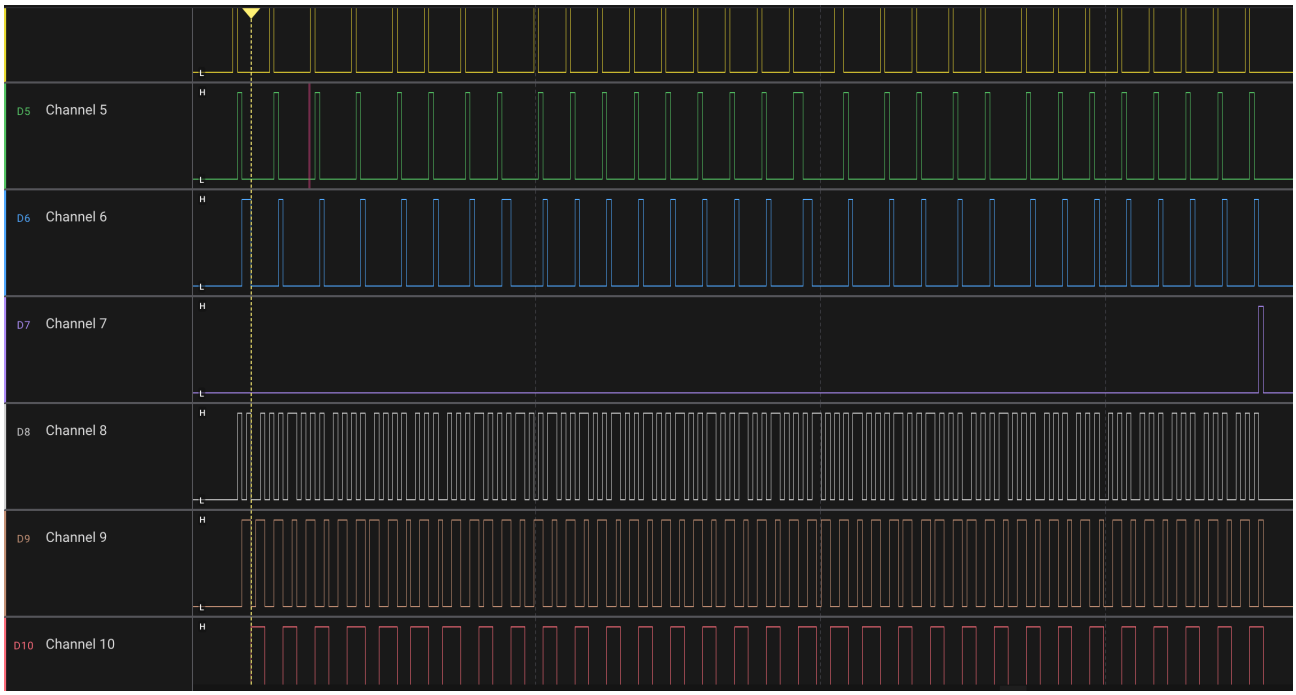

Figure 11: Accelerator v4