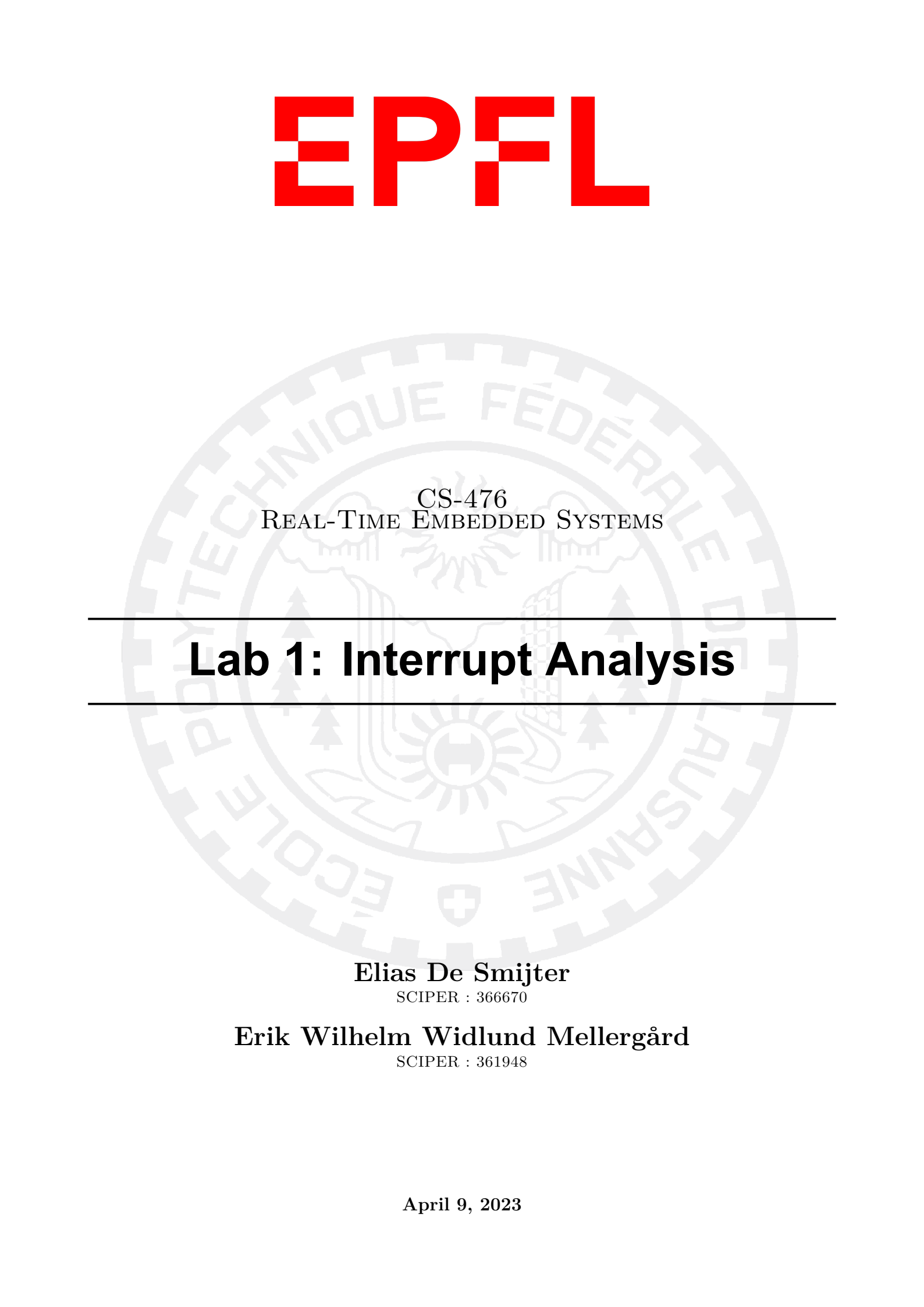

CS-476
Real-Time Embedded Systems

Lab 1: Interrupt Analysis

Elias De Smijter
SCIPER : 366670

Erik Wilhelm Widlund Mellergård
SCIPER : 361948

April 9, 2023

Contents

Introduction

This is the report for lab 1 of Real Time Embedded Systems (CS-476). The purpose of this lab is to measure response time, recovery time and latency for various hardware configurations, using various measurement methods. Regarding hardware, the times are measured for on-chip memory and SDRAM, and for different combinations of instruction and data caches enabled or disabled. In total, 6 different hardware configurations are tested. The measurement methods used are based on custom hardware modules; a counter and a parallel port. These tests are performed in a bare-metal environment. The measured times are then compared in order to evaluate both the hardware setups and the measurement methods.

Further, a uC/OS-II environment is used in combination with button-triggered interrupts, performing tests to measure semaphore, flag, mail and queue times.

1. General schematic

To perform all the measurements, different components were added together. The entire system can be seen in figure 1. The part contained within the dotted line is only used for section 3, where button-triggered interrupts are needed.

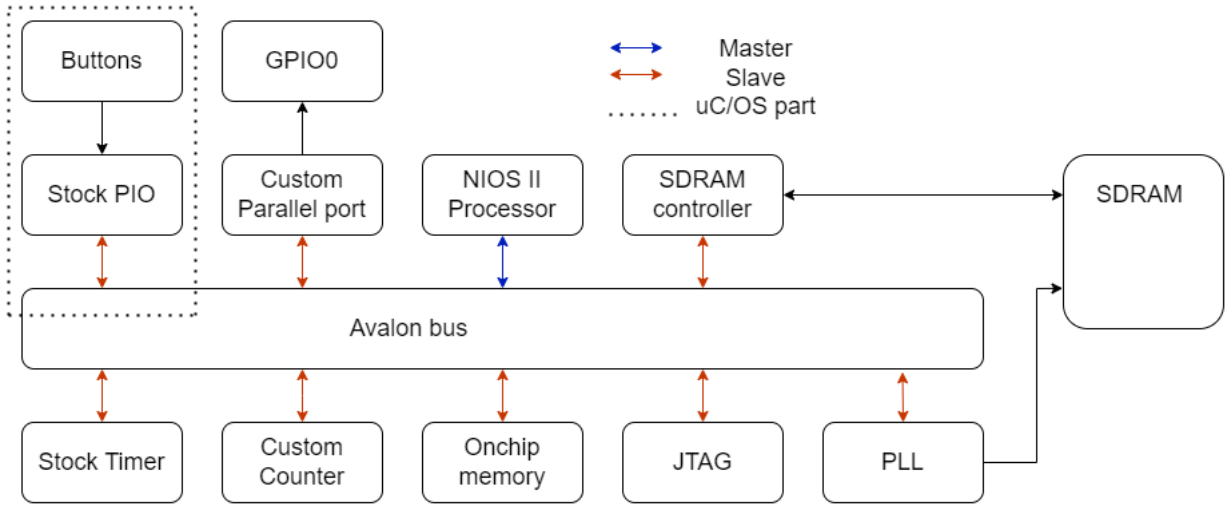

Figure 1: Overall system design

1.1 Custom counter

The custom counter is implemented in a rather straightforward way, with support for interrupt generation. A counter starts from zero, and increments on each clock cycle, until reaching its target value. An interrupt is generated when it reaches the target value, if interrupts are enabled for it. The VHDL code can be found in section 4.1. The register map of the counter can be seen in table 1, where some registers are used as command write addresses and some in the usual manner.

Address	Read	ReadData	Write	WriteData
000	Counter	iCounter	-	-
001	-	-	Reset command	DONTCARE
010	-	-	Start command	DONTCARE
011	-	-	Stop command	DONTCARE
100	IRQEn	iIRQEn	IRQEn	iIRQEn
101	Status	iEN, iEOT	ClrEOT	iClrEOT
110	Target	iTarget	Target	iTarget

Table 1: Custom counter register map

1.2 Custom parallel port

The custom parallel port is implemented in a straightforward manner with support for interrupt generation. It handles 8 ports, as outputs only, each represented by one bit in the Port values register. An interrupt request is sent when the signal is high on any specific pin for which interrupts are enabled. The VHDL code is in section 4.2.The register map of the parallel port can be seen in table 2.

Address	Read	ReadData	Write	WriteData
000	PortValues	iPortValues	PortValues	iPortValues
001	-	-	Set bits	iPortValues
010	-	-	Clear bits	iPortValues
011	IRQEN	iIRQEN	IRQEN	iIRQEN
100	-	-	IRQCLEAR command	DONTCARE

Table 2: Custom counter register map

2. Interrupt timings

2.1 Different timings

When an interrupt occurs, the interrupt handler goes through several stages to handle the problem that induced the interrupt. For every stage, there is a corresponding delay.

First, the system goes to a general interrupt service routine (ISR) which handles all the interrupt requests (IRQ). The delay to go from the running program to this main ISR is called the latency (red arrow 1 on figure 2). In this main ISR, it is decided how the interrupt is best handled. If there is a specific ISR defined for the cause of the interrupt, this specific (custom) ISR will handle the interrupt. The delay from the IRQ to the start of this custom ISR is called the response time (red arrow 2 in figure 2). When the interrupt is handled, the running program gets the control back. The delay from the end of the custom ISR to the running program is the recovery time (red arrow 3 in figure 2).

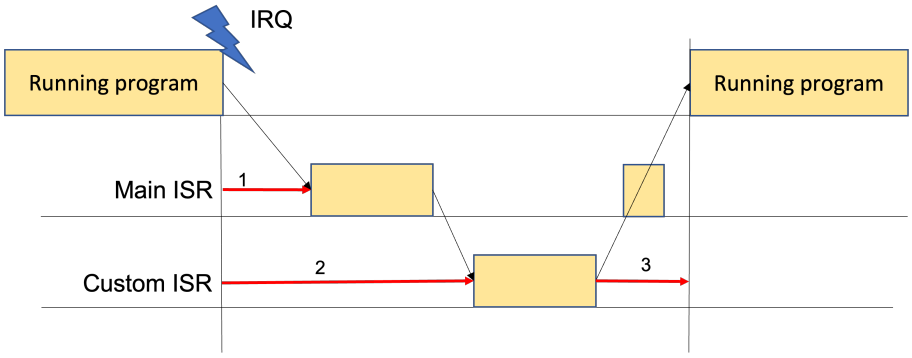

Figure 2: The different timings when handling an interrupt

2.2 Response time method

To determine the response time, two different methods were used. In the first method (the 'printing' method), a hardware-counter that can generate interrupts was used. Every time this counter reached zero, an IRQ was generated and the counter restarted from a predefined number. In the custom ISR, the first thing that happened was reading the value of the counter. This way the response time could be deduced.

The second method ('logic analyzer') used a custom parallel port (the source file for this is in section 4.2). When a bit was set high on this parallel port, an IRQ was generated. In the custom ISR, this bit was set low again. By connecting the parallel port to a logic analyzer, the response time could easily be measured as the duration of the pulse.

2.3 Recovery time method

For determining the recovery time, the procedure is very similar. In the 'printing' method, two different counters are used: one built-in and one custom (see the source code in section 4.1). The first counter

is used to regularly generate an interrupt. When this interrupt is being serviced in the custom ISR, the custom counter is started at the end of this ISR. Back in the running program, the value of the counter is read to determine the recovery time.

With the logic analyzer, reasoning is similar: a timer generates an interrupt, in the custom ISR a bit is set high and back in the running program the bit is set low again. The recovery time then corresponds to the pulse duration.

2.4 Latency method

To measure the latency, it is necessary to change the source code of the platform a little bit as some manipulation needs to happen in the main ISR. This ISR is located in *alt_irq_handler.c*. Since this latency is difficult to capture in software, it is only measured using the custom parallel port in this report. To measure the latency, an interrupt is generated by setting a bit on the parallel port high. In the main ISR, this bit is set low again. The latency corresponds to the pulse duration. The measurements could have been a bit more precise if this was implemented in the assembly code of the main ISR (instead of the c-code). This proved too much work however to make calls to the custom parallel port in assembly, and the implemented method was judged to be sufficient.

2.5 Timing results

In table 3, the results of the measurements as described above are put together for different set-ups (defined in table 4). The underlying measurements providing the data for this table can all be seen in section 5.1. For the 'printing' experiments, they are the average of 10 measurements, for the 'logic analyzer' they are the measured delay when it stabilized. To be able to compare the different measurement-methods, the 'logic analyzer'-column has the corresponding number of cycles in brackets. The system used a 50 MHz clock. The screenshots from the different measurements can be found in annex 2.

From the results, two things are clearly visible: the whole interrupt-process becomes faster when caches are enabled and it is faster with onchip memory. The reason for the difference in the memory type is not too difficult to find. The SDRAM is located further away from the processor, so it takes longer to perform a memory access. For the caches the reasoning is similar: caches are closer to the processor and are a faster memory type than the onchip memory. Overall, it can be seen that caches are more important than whether SDRAM or onchip memory is used, by noting that the results are practically the same for SDRAM and onchip when both instruction and data caches are used, across all three categories of measurements.

When analyzing table 3 in a vertical way, it can be seen that the largest gain is achieved when the instruction cache is enabled. This is because the address of the interrupt handlers is used often. The data cache also speeds up the process a little bit, but much less significantly. When analyzing it in a horizontal way, it is interesting to mention that the biggest delay when going into the custom ISR is the time spent in the main ISR. This is because here the entire state of the processor is saved in order to be able to resume the program once the interrupt is serviced. It can also be seen that the recovery time is only a little bit shorter than the response time. In the recovery time the saved state is loaded back into the processor, but there is no need to determine which custom ISR should be used (as we just returned from it). This last part is the difference between the response time and the recovery time.

It is also interesting to notice the difference between the measured timings using printing and the logic analyzer. For the recovery time, the difference between the 'printing'-measurement and 'logic analyzer'-measurement is very small. For the response time however, there is a big difference between the two methods.

Experiment	Response time		Recovery time		Latency
	printing	logic analyzer	printing	logic analyzer	logic analyzer
1	658 cycles	9.76μs (488)	316 cycles	6.4μs (320)	4.5μs (225)
2	284 cycles	4.16μs (208)	140 cycles	2.72μs (136)	1.76μs (88)
3	170 cycles	2.4μs (120)	79 cycles	1.44μs (72)	0.8μs (40)
4	932 cycles	13.28μs (664)	599 cycles	11.68μs (584)	4.32μs (216)
5	537 cycles	7.36μs (368)	350 cycles	6.72μs (336)	2.56μs (128)
6	174 cycles	2.4μs (120)	81 cycles	1.44μs (72)	0.8μs (40)

Table 3: Interrupt timings measurement results

Experiment	memory type	instruction cache enabled?	data cache enabled?
1	onchip memory	✗	✗
2	onchip memory	✓	✗
3	onchip memory	✓	✓
4	SDRAM	✗	✗
5	SDRAM	✓	✗
6	SDRAM	✓	✓

Table 4: Tested hardware configurations

3. Synchronization primitives timings

Using the uC/OS II operating system, which enables real time operation, some overheads are introduced by the different synchronization primitives. This section describes the measurement of some such overheads, specifically the Semaphore, Flags, Mailbox and Queue primitives. The hardware setup is mostly the same as described in section 1, with the addition of a stock PIO connected to the buttons on the board to trigger interrupts. Due to the memory requirements of uC/OS II, SDRAM is used along with both instruction and data caches.

For all overheads, the measurement implementation follows the same principle. A call to the function PEND is made in the mainloop, making the system wait for the meeting of condition(s) given in the function call. Then, interrupts triggered by the PIO buttons are used: in the ISRs they trigger, the conditions are fulfilled using the POST function. The overhead time is then the time it takes from just after the POST, to returning to the mainloop and executing the next line of code just after the PEND. To measure this, a timer is used. The timer value is recorded immeduately after POST in the ISR, and then recorded again just after the PEND in the mainloop. Finally, the overhead is calculated as the difference between the two recordings.

3.1 Semaphore measurement

One way to manage shared resources between tasks is to use semaphores, which may be accessed by either task. The introduced overhead time then consists of the time it takes for the OS to convert a semaphore from one task to another. To measure this, an assignment is made to only provide one available resource, forcing the conversion when accessed from a second task. In the mainloop, the PEND function is called, which in this case means that the system waits for the available resource. In the ISR, the POST function is used to make the resource available. Using the measurement method described in 3, the overhead of the semaphore primitive is measured.

3.2 Flag measurement

Two separate measurements are done with respect to flags, an OR and an AND version. Both operate on the idea of creating an event flag group, and then calling the PEND function in the mainloop, and waiting for the POST function to be called from inside an ISR. In this ISR, the POST function is used to set the corresponding flag bit(s). For the OR case, only one of the 4 available flags, corresponding to the 4 buttons, is needed. The AND case requires all 4 flags. The overhead is measured like previously.

3.3 Mailbox measurement

The intuitively named mailbox primitive allows for arrangement of multi-tasks. First, a message is created, and then PEND is used to wait for that message to be received. The POST call from the IRQ is in this case used to send the message. With the same measurement method, the mailbox primitive overhead is measured. Here, it intuitively describes the time it takes to ship the message.

3.4 Queue measurement

Very similar to the mailbox primitive, here a message queue is created, and the mainloop PEND function is used to wait for a specific message in the queue. Again, the ISR calls the POST function to send that message, and again the same measurement method is used.

3.5 Measurement results

Table 5 records the amount of clock cycles each primitive introduces in overhead. These results are gathered from the measurements made and recorded in section 5.2. They are all similar in value, with the semaphore being the fastest and the flags the slowest. While the differences are small, they should still be noted when choosing between synchronization methods in a real-time application.

Further, it can be noted that both flag primitives have the same overhead. This is as expected: the AND flag will behave just as the OR flag once all but one of the awaited flags are set. Both cases then have the same situation: expecting one single flag, and then reporting their condition fulfilled.

Measurement	Time (Clock cycles)
Semaphore	360
Flags (OR)	466
Flags (AND)	466
Mail	435
Queue	380

Table 5: uC/OS measurement results

4. Annex 1: source code of custom components

4.1 Counter

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

-- Register map:
-- 000 = Counter[31... 0]    Counter value (read access)
-- 001 = Rz        Reset the counter (write access)
-- 010 = Start        Start counting (write access)
-- 011 = Stop        Stop counting (write access)
-- 100 = Command[7... 0]    General command (RW access)
-- 101 = Status[7... 0]      General status (RW access)
-- 110 = Target[31... 0]    Count target value (RW access)

entity Counter is
  PORT(
    Clk       :  IN    std_logic;
    nReset    :  IN    std_logic;
    -- Avalon Slave interface signals
    Address   :  IN    std_logic_vector(2 downto 0);
    Read      :  IN    std_logic;
    ReadData  :  OUT   std_logic_vector(31 downto 0);
    Write     :  IN    std_logic;
    WriteData :  IN   std_logic_vector(31 downto 0);
    -- Interrupt Request signal
    IRQ       :  OUT   std_logic
  );
end Counter;

architecture comp of Counter is
  -- Counter
  signal   iCounter  :  unsigned(31 downto 0);
  -- Enable count
  signal  iEn    :  std_logic;
  -- Reset count
  signal  iRz    :  std_logic;
  -- Has reached end
  signal  iEOT   :  std_logic;
  -- Clear has reached end
  signal  iClrEOT  :  std_logic;
  -- Interrupts enabled
  signal  iIRQEn  :  std_logic;
  -- Count target value
  signal  iTarget  :  std_logic_vector(31 downto 0);

  begin
  -- Counter process
  -- Drives iCounter
  Counter_process:
```

```
process(Clk)
begin
  if rising_edge(Clk) then
    if iRz = '1' then
      -- Reset counter to zero
      iCounter <= (others => '0');
    elsif iEn = '1' then
      if iEOT = '0' then
        -- Increment counter
        iCounter <= iCounter + 1;
      end if;
    end if;
  end if;
end process Counter_process;

-- Read process
-- Drives no signals
Read_process:
process(Clk)
begin
  if rising_edge(Clk) then
    -- Default value

    -- TODO: should the default value be std_logic_vector(iCounter)
    -- to facilitate that
    -- "The counter value must be readable at all times -> transfer
     --   the counter value at the start of the read cycle"

    ReadData <= (others => '0');
    -- Read cycle
    if Read = '1' then
      case Address(2 downto 0) is
        when "000" =>
          -- Read the counter register value
          ReadData <= std_logic_vector(iCounter);
        when "100" =>
          -- Read the command register value (which is only
          -- the iIRQEn value)
          ReadData(0) <= iIRQEn;
        when "101" =>
          -- Read the status register value (iEOT and iEn)
          ReadData(0) <= iEOT;
          ReadData(1) <= iEn;
        when "110" =>
          -- Read the count target value
          ReadData <= std_logic_vector(iTarget);
        when others => null;
      end case;
    end if;
  end if;
end process Read_process;

-- Write process (also handles asynchronous reset)
```

```
-- Drives iEn, iRz, iIRQEn and iClrEOT
Write_process:
process(Clk, nReset)
begin
  -- Asynchronous reset: counting disabled, counter is reset, interrupts disabled
  if nReset = '0' then
    iEn <= '0';
    iRz <= '1';
    iIRQEn <= '0';
    iClrEOT <= '1';
  elsif rising_edge(Clk) then
    -- Default values: don't reset or clear
    iRz <= '0';
    iClrEOT <= '0';
    -- Write cycle
    if Write = '1' then
      case Address(2 downto 0) is
        -- Commands: Writing to them just means to perform the
        -- command associated with the register, so WriteData
        -- is not considered.
        when "001" =>
          -- Rz: reset counter command
          iRz <= '1';
          iClrEOT <= '1';
        when "010" =>
          -- Start: enable counting command
          iEn <= '1';
        when "011" =>
          -- Stop: disable counting command
          iEn <= '0';

        -- Registers: WriteData is considered
        when "100" =>
          -- Command register
          iIRQEn <= WriteData(0);
        when "101" =>
          -- Status register
          iClrEOT <= WriteData(0);
        when "110" =>
          -- Target value register
          iTarget <= WriteData;
          iClrEOT <= '1';
        when others => null;
      end case;
    end if;
  end if;
end process Write_process;

-- Interrupt process. End Of Time (iEOT) is activated when iCounter
-- is at the value in iTarget
-- Drives iEOT
Interrupt_process:
process(Clk)
```

```
  begin
    if rising_edge(Clk) then
      if std_logic_vector(iCounter) = iTarget then
        -- Flag End Of Time when counter = target
        iEOT <= '1';
      end if;
      if iClrEOT = '1' then
        -- Cleared by writing 1 to status(0)
        iEOT <= '0';
      end if;
    end if;
  end process Interrupt_process;

  -- IRQ activation. No process required since it's only one signal
  -- conditionally activated
  IRQ <= '1' when (iEOT = '1' and iIRQEn = '1' and iEN = '1') else '0';

end comp;
```

4.2 Parallel port

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

-- Register map
-- 000 = Port values (RW)
-- 001 = set bit access (W)
-- 010 = clear bit access (W)
-- 011 = IRQ enable (RW)
-- 100 = IRQ clear (W)

entity ParallelPort is
  port(
    Clk       :  IN    std_logic;
    nReset    :  IN    std_logic;
    -- Avalon Slave interface signals
    Address   :  IN    std_logic_vector(2 downto 0);
    Read      :  IN    std_logic;
    ReadData  :  OUT   std_logic_vector(31 downto 0);
    Write     :  IN    std_logic;
    WriteData :  IN    std_logic_vector(31 downto 0);
    -- Port output
    Pout      :  OUT   std_logic_vector(31 downto 0);
    -- Interrupt Request signal
    IRQ       :  OUT   std_logic
  );
end ParallelPort;

architecture comp of ParallelPort is
  signal  iPortValues  :  std_logic_vector(31 downto 0);
  signal  iIRQEN       :  std_logic_vector(31 downto 0);
```

```
signal  iIRQCLEAR  :  std_logic;

begin

-- Read process
-- Any read request will return the port values, regardless of address
-- Drives no signals
Read_process:
process(Clk)
begin
  if rising_edge(Clk) then
    if Read = '1' then
      case Address(2 downto 0)  is
        when "000" =>
          ReadData <= iPortValues;
        when "011" =>
          ReadData <= iIRQEN;
        when others => null;
      end case;
    end if;
  end if;
end process Read_process;

-- Write process (also handles asynchronous reset)
-- Writes, sets and clears bits in the Port values register
-- Drives iPortValues
Write_process:
process(Clk, nReset)
begin
  if nReset = '0' then
    iPortValues <= (others => '0');
  elsif rising_edge(Clk) then
    -- Write cycle
    iIRQCLEAR <= '0';
    if Write = '1' then
      case Address(2 downto 0)  is
        when "000" =>
          iPortValues <= WriteData;
        when "001" =>
          -- Set bits: the corresponding bits are set to 1,
          -- the others are left untouched
          iPortValues <= (iPortValues OR WriteData);
        when "010" =>
          -- Clear bits: the corresponding bits are cleared,
          -- the others are left untouched
          iPortValues <= (iPortValues AND NOT WriteData);
        when "011" =>
          -- Toggle interrupt enabled or disabled
          iIRQEN <= WriteData;
        when "100" =>
          iIRQCLEAR <= '1';
        when others => null;
      end case;
```

```
      end if;
    end if;
  end process Write_process;

  Interrupt_process:
  process(Clk, iPortValues, iIRQCLEAR)
  begin
    if rising_edge(Clk) then
      if( or_reduce(iPortValues AND iIRQEN) = '1') then
        IRQ <= '1';
      end if;
      if(iIRQCLEAR = '1') then
        IRQ <= '0';
      end if;
    end if;
  end process Interrupt_process;


  -- Set port values
  Pout <= iPortValues;

end comp;
```

5. Annex 2: screenshots of measurements

5.1 Bare-metal measurement results

Onchip memory, no instruction cache, no data cache

```
Testing response time using stock timer and
 counting the average over 10 iterations...
Value at iteration 1 is 0x2a7 clock cycles
Value at iteration 2 is 0x2a1 clock cycles
Value at iteration 3 is 0x2a1 clock cycles
Value at iteration 4 is 0x2a2 clock cycles
Value at iteration 5 is 0x2a7 clock cycles
Value at iteration 6 is 0x2a6 clock cycles
Value at iteration 7 is 0x2a5 clock cycles
Value at iteration 8 is 0x2a6 clock cycles
Value at iteration 9 is 0x2a6 clock cycles
Value at iteration a is 0x2a5 clock cycles
Average value for response time is 676 clock cycles

Testing recovery time using custom counter and
 counting the average over 10 iterations...
Value at iteration 1 is 0x3eee clock cycles
Value at iteration 2 is 0x140 clock cycles
Value at iteration 3 is 0x132 clock cycles
Value at iteration 4 is 0x147 clock cycles
Value at iteration 5 is 0x138 clock cycles
Value at iteration 6 is 0x128 clock cycles
Value at iteration 7 is 0x138 clock cycles
Value at iteration 8 is 0x12d clock cycles
Value at iteration 9 is 0x147 clock cycles
Value at iteration a is 0x132 clock cycles
Average value for recovery time is 1891 clock cycles
```

Figure 3: Print-based response and recovery time

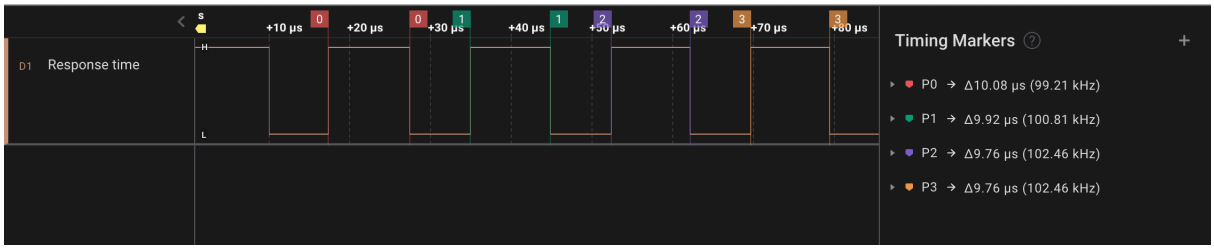

Figure 4: PIO response time

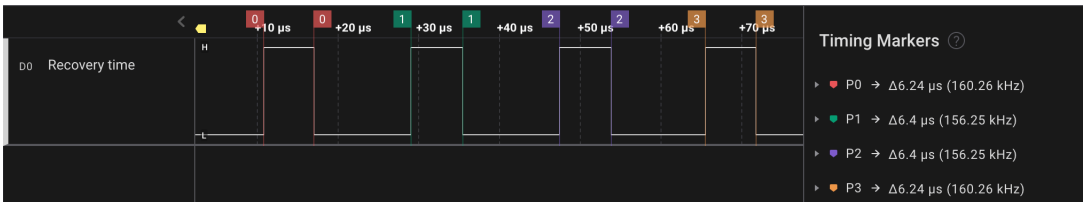

Figure 5: PIO recovery time

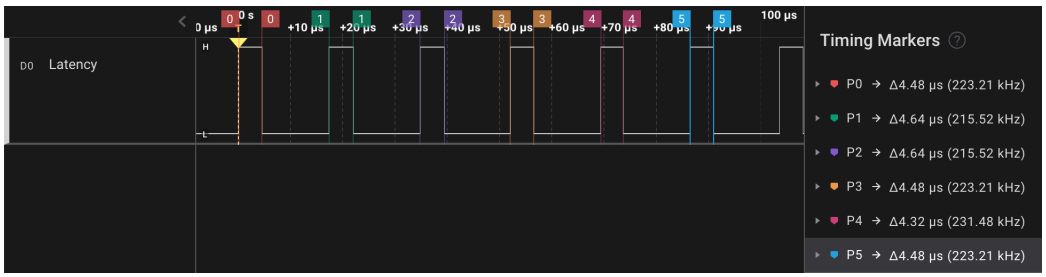

Figure 6: PIO latency

Onchip memory, 64 kb instruction cache, no data cache

```
Testing response time using stock timer and
 counting the average over 10 iterations...
Value at iteration 1 is 0x12b clock cycles
Value at iteration 2 is 0x119 clock cycles
Value at iteration 3 is 0x11a clock cycles
Value at iteration 4 is 0x11c clock cycles
Value at iteration 5 is 0x11f clock cycles
Value at iteration 6 is 0x11c clock cycles
Value at iteration 7 is 0x119 clock cycles
Value at iteration 8 is 0x11b clock cycles
Value at iteration 9 is 0x11a clock cycles
Value at iteration a is 0x117 clock cycles
Average value for response time is 284 clock cycles

Testing recovery time using custom counter and
 counting the average over 10 iterations...
Value at iteration 1 is 0x90 clock cycles
Value at iteration 2 is 0x8a clock cycles
Value at iteration 3 is 0x8a clock cycles
Value at iteration 4 is 0x8a clock cycles
Value at iteration 5 is 0x87 clock cycles
Value at iteration 6 is 0x93 clock cycles
Value at iteration 7 is 0x8a clock cycles
Value at iteration 8 is 0x8a clock cycles
Value at iteration 9 is 0x93 clock cycles
Value at iteration a is 0x8a clock cycles
Average value for recovery time is 140 clock cycles
```

Figure 7: Print-based response and recovery time

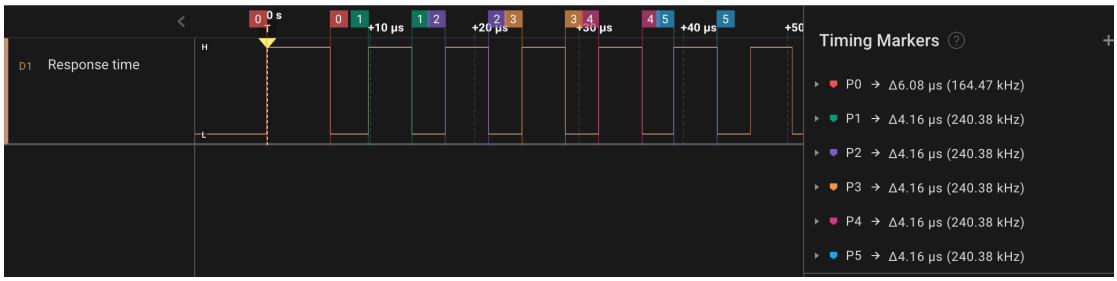

Figure 8: PIO response time

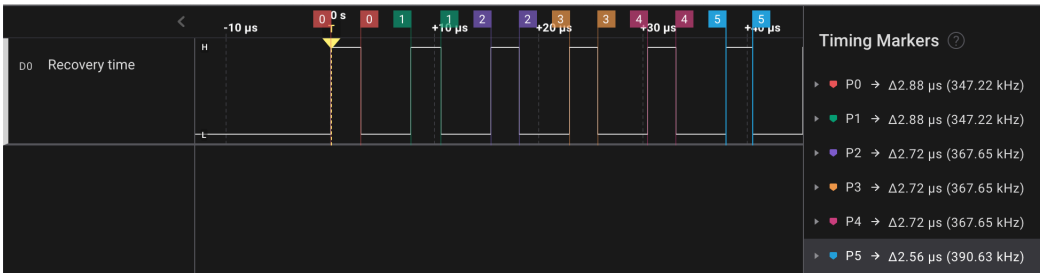

Figure 9: PIO recovery time

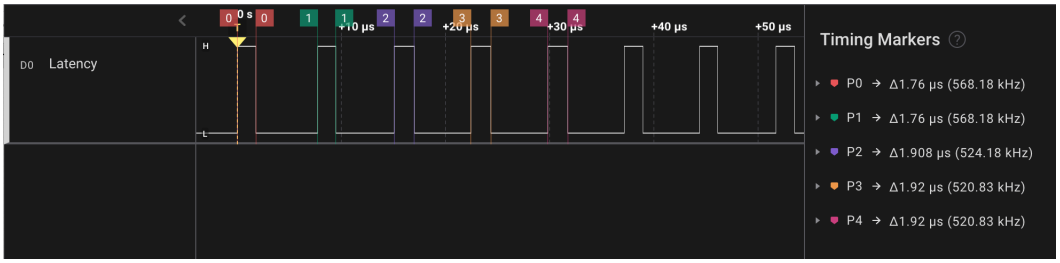

Figure 10: PIO latency

Onchip memory, 64 kb instruction cache, 64 kb data cache

```
Testing response time using stock timer and
 counting the average over 10 iterations...
Value at iteration 1 is 0xbb clock cycles
Value at iteration 2 is 0xac clock cycles
Value at iteration 3 is 0xa9 clock cycles
Value at iteration 4 is 0xa8 clock cycles
Value at iteration 5 is 0xa7 clock cycles
Value at iteration 6 is 0xa8 clock cycles
Value at iteration 7 is 0xa8 clock cycles
Value at iteration 8 is 0xa9 clock cycles
Value at iteration 9 is 0xa7 clock cycles
Value at iteration a is 0xa7 clock cycles
Average value for response time is 170 clock cycles
```

(a)

```
Testing recovery time using custom counter and
 counting the average over 10 iterations...
Value at iteration 1 is 0x59 clock cycles
Value at iteration 2 is 0x57 clock cycles
Value at iteration 3 is 0x51 clock cycles
Value at iteration 4 is 0x4e clock cycles
Value at iteration 5 is 0x4b clock cycles
Value at iteration 6 is 0x4c clock cycles
Value at iteration 7 is 0x56 clock cycles
Value at iteration 8 is 0x50 clock cycles
Value at iteration 9 is 0x49 clock cycles
Value at iteration a is 0x49 clock cycles
Average value for recovery time is 79 clock cycles
```

(b)

Figure 11: Print-based (a) response and (b) recovery time

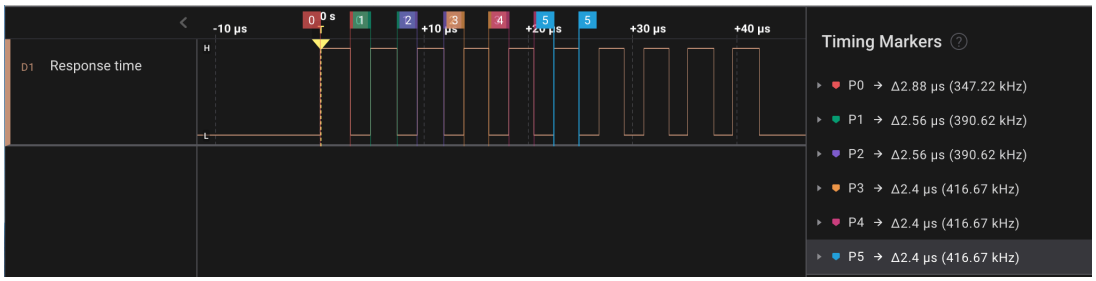

Figure 12: PIO response time

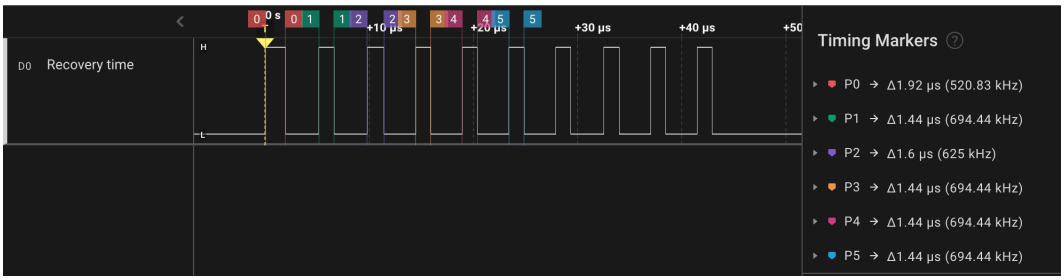

Figure 13: PIO recovery time

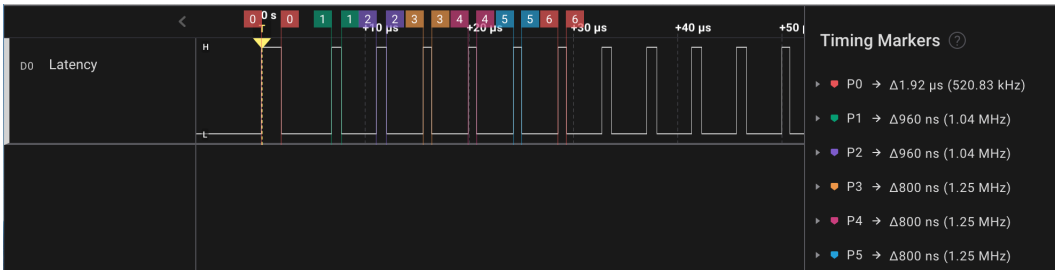

Figure 14: PIO latency

SDRAM, no instruction cache, no data cache

```
Testing response time using stock timer and
 counting the average over 10 iterations...
Value at iteration 1 is 0x3b4 clock cycles
Value at iteration 2 is 0x39a clock cycles
Value at iteration 3 is 0x3a2 clock cycles
Value at iteration 4 is 0x3ab clock cycles
Value at iteration 5 is 0x3a5 clock cycles
Value at iteration 6 is 0x3a8 clock cycles
Value at iteration 7 is 0x394 clock cycles
Value at iteration 8 is 0x3a3 clock cycles
Value at iteration 9 is 0x3a2 clock cycles
Value at iteration a is 0x3ab clock cycles
Average value for response time is 932 clock cycles
```

(a)

```
Testing recovery time using custom counter and
 counting the average over 10 iterations...
Value at iteration 1 is 0x273 clock cycles
Value at iteration 2 is 0x278 clock cycles
Value at iteration 3 is 0x238 clock cycles
Value at iteration 4 is 0x278 clock cycles
Value at iteration 5 is 0x257 clock cycles
Value at iteration 6 is 0x218 clock cycles
Value at iteration 7 is 0x21e clock cycles
Value at iteration 8 is 0x276 clock cycles
Value at iteration 9 is 0x270 clock cycles
Value at iteration a is 0x259 clock cycles
Average value for recovery time is 599 clock cycles
```

(b)

Figure 15: Print-based (a) response and (b) recovery time

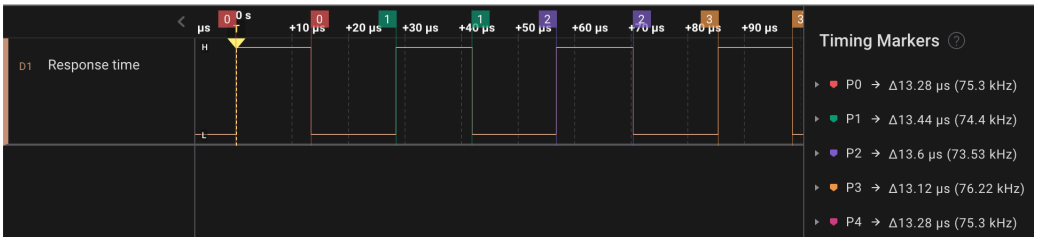

Figure 16: PIO response time

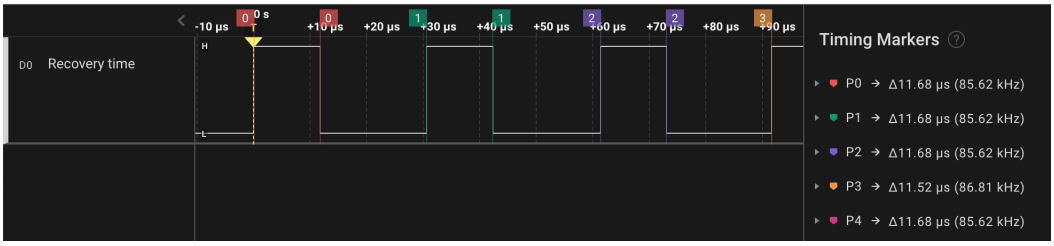

Figure 17: PIO recovery time

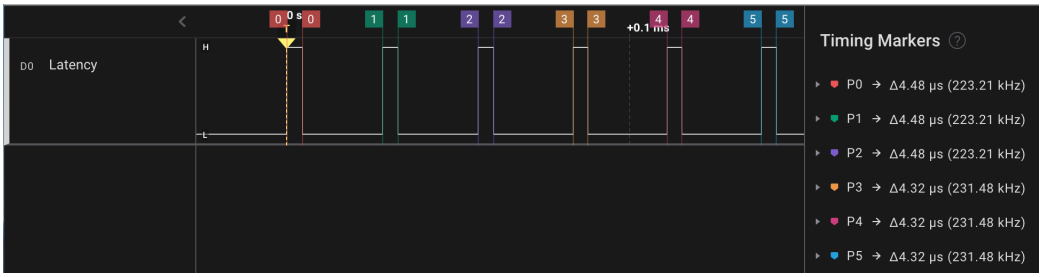

Figure 18: PIO latency

SDRAM, 64 kb instruction cache, no data cache

```
Testing response time using stock timer and
 counting the average over 10 iterations...
Value at iteration 1 is 0x285 clock cycles
Value at iteration 2 is 0x214 clock cycles
Value at iteration 3 is 0x20b clock cycles
Value at iteration 4 is 0x210 clock cycles
Value at iteration 5 is 0x203 clock cycles
Value at iteration 6 is 0x20e clock cycles
Value at iteration 7 is 0x21a clock cycles
Value at iteration 8 is 0x215 clock cycles
Value at iteration 9 is 0x203 clock cycles
Value at iteration a is 0x205 clock cycles
Average value for response time is 537 clock cycles
```

(a)

```
Testing recovery time using custom counter and
 counting the average over 10 iterations...
Value at iteration 1 is 0x177 clock cycles
Value at iteration 2 is 0x16e clock cycles
Value at iteration 3 is 0x155 clock cycles
Value at iteration 4 is 0x155 clock cycles
Value at iteration 5 is 0x154 clock cycles
Value at iteration 6 is 0x15a clock cycles
Value at iteration 7 is 0x155 clock cycles
Value at iteration 8 is 0x16c clock cycles
Value at iteration 9 is 0x154 clock cycles
Value at iteration a is 0x15b clock cycles
Average value for recovery time is 350 clock cycles
```

(b)

Figure 19: Print-based (a) response and (b) recovery time

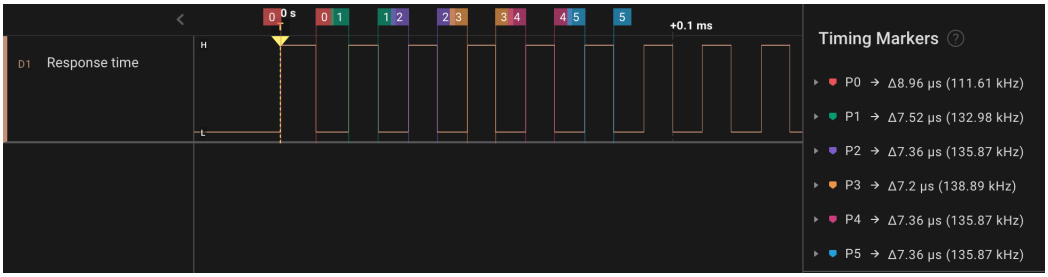

Figure 20: PIO response time

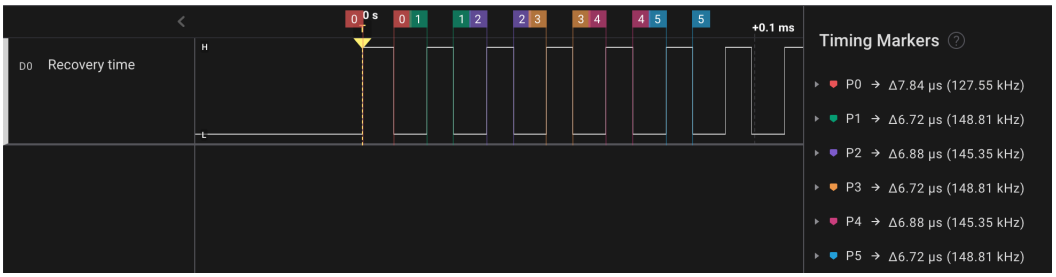

Figure 21: PIO recovery time

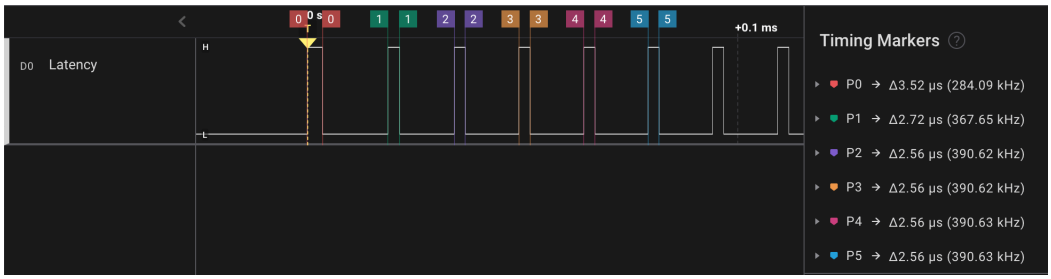

Figure 22: PIO latency

SDRAM, 64 kb instruction cache, 64 kb data cache

```
Testing response time using stock timer and
 counting the average over 10 iterations...
Value at iteration 1 is 0xfc clock cycles
Value at iteration 2 is 0xa5 clock cycles
Value at iteration 3 is 0xa5 clock cycles
Value at iteration 4 is 0xa6 clock cycles
Value at iteration 5 is 0xa5 clock cycles
Value at iteration 6 is 0xa5 clock cycles
Value at iteration 7 is 0xa7 clock cycles
Value at iteration 8 is 0xa5 clock cycles
Value at iteration 9 is 0xa7 clock cycles
Value at iteration a is 0xa5 clock cycles
Average value for response time is 174 clock cycles
```

(a)

```
Testing recovery time using custom counter and
 counting the average over 10 iterations...
Value at iteration 1 is 0x69 clock cycles
Value at iteration 2 is 0x4d clock cycles
Value at iteration 3 is 0x56 clock cycles
Value at iteration 4 is 0x4e clock cycles
Value at iteration 5 is 0x4e clock cycles
Value at iteration 6 is 0x49 clock cycles
Value at iteration 7 is 0x48 clock cycles
Value at iteration 8 is 0x56 clock cycles
Value at iteration 9 is 0x54 clock cycles
Value at iteration a is 0x48 clock cycles
Average value for recovery time is 81 clock cycles
```

(b)

Figure 23: Print-based (a) response and (b) recovery time

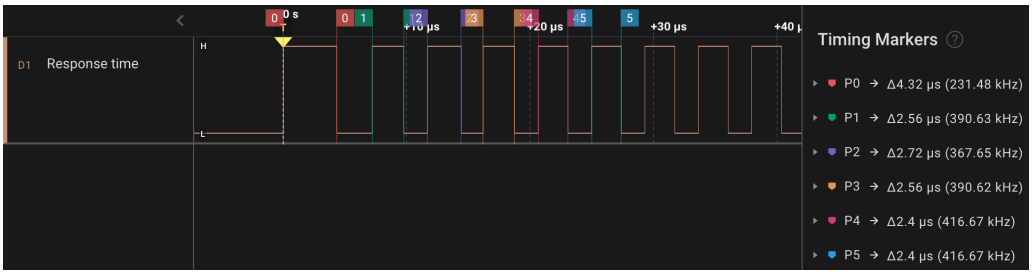

Figure 24: PIO response time

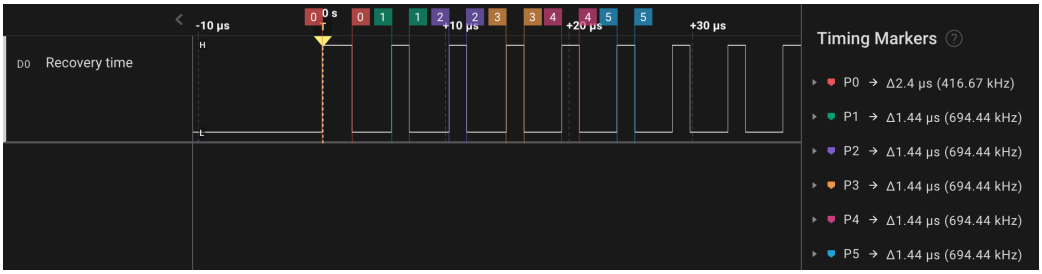

Figure 25: PIO recovery time

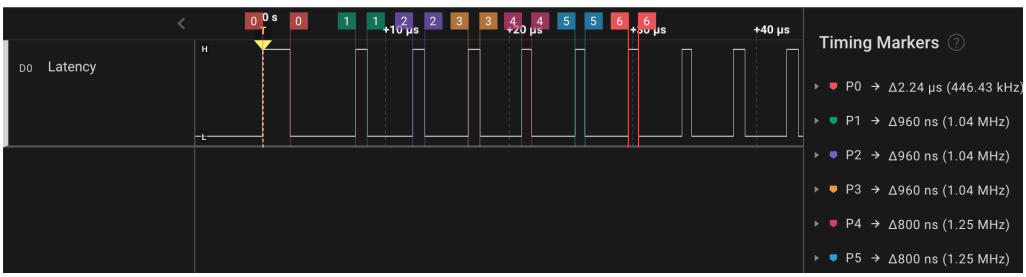

Figure 26: PIO latency

5.2 uC/OS II results

```
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
Getting semaphore in : 360 cycle
```

Figure 27: Semaphore measurement

```
Testing flag OR version
Getting flag in : 788 cycle
Testing flag OR version
Getting flag in : 465 cycle
Testing flag OR version
Getting flag in : 468 cycle
Testing flag OR version
Getting flag in : 466 cycle
Testing flag OR version
Getting flag in : 466 cycle
Testing flag OR version
Getting flag in : 466 cycle
Testing flag OR version
Getting flag in : 466 cycle
Testing flag OR version
Getting flag in : 466 cycle
Testing flag OR version
Getting flag in : 466 cycle
Testing flag OR version
Getting flag in : 470 cycle
Testing flag OR version
Getting flag in : 466 cycle
Testing flag OR version
Getting flag in : 466 cycle
Testing flag OR version
Getting flag in : 472 cycle
```

(a)

```
Testing flag AND version
Getting flag in : 762 cycle
Testing flag AND version
Getting flag in : 471 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
Testing flag AND version
Getting flag in : 466 cycle
```

(b)

Figure 28: Flag measurement, (a) OR (b) AND

```
Getting message in : 439 cycles with button number 2 and edge 00
Getting message in : 433 cycles with button number 4 and edge 00
Getting message in : 437 cycles with button number 8 and edge 00
Getting message in : 433 cycles with button number 8 and edge 00
Getting message in : 433 cycles with button number 4 and edge 00
Getting message in : 433 cycles with button number 2 and edge 00
Getting message in : 433 cycles with button number 8 and edge 00
Getting message in : 433 cycles with button number 2 and edge 00
Getting message in : 433 cycles with button number 1 and edge 80
Getting message in : 439 cycles with button number 1 and edge 80
Getting message in : 437 cycles with button number 2 and edge 00
Getting message in : 433 cycles with button number 4 and edge 00
Getting message in : 433 cycles with button number 8 and edge 00
Getting message in : 433 cycles with button number 8 and edge 00
Getting message in : 433 cycles with button number 4 and edge 00
Getting message in : 433 cycles with button number 2 and edge 00
Getting message in : 433 cycles with button number 1 and edge 80
Getting message in : 435 cycles with button number 1 and edge 80
Getting message in : 433 cycles with button number 2 and edge 00
Getting message in : 433 cycles with button number 8 and edge 00
```

Figure 29: Mail measurement

```
Getting message from queue in : 793 cycles with button number 8 and edge 00
Getting message from queue in : 375 cycles with button number 4 and edge 00
Getting message from queue in : 375 cycles with button number 2 and edge 00
Getting message from queue in : 378 cycles with button number 1 and edge 80
Getting message from queue in : 375 cycles with button number 1 and edge 80
Getting message from queue in : 375 cycles with button number 1 and edge 80
Getting message from queue in : 381 cycles with button number 1 and edge 80
Getting message from queue in : 378 cycles with button number 1 and edge 80
Getting message from queue in : 378 cycles with button number 2 and edge 00
Getting message from queue in : 381 cycles with button number 4 and edge 00
Getting message from queue in : 381 cycles with button number 8 and edge 00
Getting message from queue in : 378 cycles with button number 8 and edge 00
Getting message from queue in : 381 cycles with button number 4 and edge 00
Getting message from queue in : 378 cycles with button number 2 and edge 00
Getting message from queue in : 384 cycles with button number 1 and edge 80
Getting message from queue in : 375 cycles with button number 2 and edge 00
Getting message from queue in : 381 cycles with button number 2 and edge 00
Getting message from queue in : 378 cycles with button number 4 and edge 00
Getting message from queue in : 381 cycles with button number 8 and edge 00
```

Figure 30: Queue measurement