Rapport Lab 4

wonhol@matnat.uio.no
malina@matnat.uio.no

Task 1:

1)

Generiske regler. Finner svaret ved å plusse (2+2) bit. Vet at dette blir 3 bit, en økning på 1 bit. Mao. blir (16+16)bit = 17 bit når man antar maksimum størrelse.

2)

Generiske regler. Multipliserer (2*2) bit. Vet at dette blir 4 bit. Mao er (16*16)bit = 32 bit når man antar maksimum størrelse

3)

Generiske regler. Prøver med (2+2+2+2)bit = 4 bit, en økning på 2 bit. Derfor blir svaret 18 bit når man adderer 4 16-bits tall.

4)

Vet at (16+16+16+16)bit = 18 bit.
Videre multipliserer vi (2*4)bit = 6 bit for å sjekke de generiske reglene på lavt nivå.
(18*16)bit er derfor 34 bit ved maksimum størrelse på tallet.

6)

34 flip-flops/registre brukes. (16+16+16+16)*14 = 34

7)

69 flip-flops/registre brukes. Tar litt over dobbelt så stor plass som i modul compute, men teoretisk sett kan prosessene gå nesten dobbelt så fort.

Task 2:

1. Simulating AXI4Lite BFM

After adding pos_seg7_ctrl module, setpoint write is done by adding

```
bdata := x"12";
MW(BYTE, LAB4REG_SETPOINT, x"12");
MR(BYTE, LAB4REG_SETPOINT, bvalue);
writef(tb_lab4_log, cycle_no, "write value in testbench : " & lv2strx(bvalue,8));
MC(BYTE, LAB4REG_SETPOINT, x"12");
```

in the test bench. In the simulation, it was observed that sp value was propagated to pos_seg7_ctrl module.

2. Adding registers in register module

Extra registers are added in lab4reg and its addresses are defined in lab4_pck.

Followings were added in lab4reg

In entity :

```
reg32            : out std_logic_vector(31 downto 0);
reg16            : out std_logic_vector(15 downto 0);
```

In RTL, write process :

```
if pif_addr(15 downto 2) = LAB4REG_REG32(15 downto 2) then
  reg32_i(31 downto 0) <=  pif_wdata(31 downto 0);
end if;

if pif_addr(15 downto 2) = LAB4REG_REG16(15 downto 2) then
  reg16_i(15 downto 0) <=  pif_wdata(15 downto 0);
end if;
```

In RTL, read process :

```
if pif_addr(15 downto 2) = LAB4REG_REG32(15 downto 2) then
  reg_data_out <= reg32_i;
end if;

if pif_addr(15 downto 2) = LAB4REG_REG16(15 downto 2) then
```

```
          reg_data_out(15 downto 0) <= reg16_i;
        end if;
```

In lab4_pck :

```
        constant LAB4REG_REG32 : std_logic_vector(31 downto 0) := x"40000014";
        constant LAB4REG_REG16 : std_logic_vector(31 downto 0) := x"40000018";
```

Then in top module, component instantiation for the RAM module is uncommented.

In the testbench, write and read to registers are simulated with following lines :

```
    wdata := x"DEADBEEF";
     MW(SINGLE, LAB4REG_REG32, wdata);
     MR(SINGLE, LAB4REG_REG32, wvalue);
     writef(tb_lab4_log, cycle_no, "Read value from REG32 in testbench : " &
lv2strx(wvalue,32));
     MC(SINGLE, LAB4REG_REG32, x"DEADBEEF");

     MW(HALFWORD, LAB4REG_REG16, x"BEEF");
     MR(HALFWORD, LAB4REG_REG16, hvalue);
     writef(tb_lab4_log, cycle_no, "Read value from REG16 in testbench : " &
lv2strx(hvalue,16));
     MC(HALFWORD, LAB4REG_REG16, x"BEEF");
```

The result was successful for both register and RAM test by simulation

```
   20 AXI4Lite single word write to AXI4PIFB          @ 4000000C <= 12345678
   31 AXI4Lite single word read from AXI4PIFB         @ 4000000C <= 12345678
   42 Read value in testbench : 12345678
   42 AXI4Lite single word read from AXI4PIFB         @ 4000000C <= 12345678
   52 AXI4Lite single word write to AXI4PIFB          @ 4000000C <= ABCDEF98
   63 AXI4Lite single word read from AXI4PIFB         @ 4000000C <= ABCDEF98
   74 Read value in testbench : ABCDEF98
   74 AXI4Lite single word read from AXI4PIFB         @ 4000000C <= ABCDEF98
   84 AXI4Lite byte 0 write to AXI4PIFB               @ 40000010 <= 12
   95 AXI4Lite byte 0 read from AXI4PIFB              @ 40000010 <= 12
  106 write value in testbench : 12
  106 AXI4Lite byte 0 read from AXI4PIFB              @ 40000010 <= 12
  116 AXI4Lite single word write to AXI4PIFB          @ 40000014 <= DEADBEEF
  127 AXI4Lite single word read from AXI4PIFB         @ 40000014 <= DEADBEEF
  138 write value in testbench : DEADBEEF
  138 AXI4Lite single word read from AXI4PIFB         @ 40000014 <= DEADBEEF
  148 AXI4Lite halfword lower bytes write to AXI4PIFB  @ 40000018 <= BEEF
  159 AXI4Lite halfword lower bytes read from AXI4PIFB @ 40000018 <= BEEF
  170 write value in testbench : BEEF
  170 AXI4Lite halfword lower bytes read from AXI4PIFB @ 40000018 <= BEEF
  180 AXI4Lite single word read from AXI4PIFB         @ 40400000 <= DEADBEEF
  189 AXI4Lite single word write to AXI4PIFB          @ 40400000 <= 12345678
  196 AXI4Lite single word write to AXI4PIFB          @ 40400004 <= 87654321
  204 AXI4Lite single word write to AXI4PIFB          @ 40400008 <= ABCDEF98
  212 AXI4Lite single word read from AXI4PIFB         @ 40400000 <= 12345678
  220 AXI4Lite single word read from AXI4PIFB         @ 40400004 <= 87654321
  228 AXI4Lite single word read from AXI4PIFB         @ 40400008 <= ABCDEF98
  236 AXI4Lite halfword lower bytes write to AXI4PIFB  @ 40400000 <= 1234
  244 AXI4Lite halfword upper bytes write to AXI4PIFB  @ 40400002 <= 5678
  252 AXI4Lite halfword lower bytes read from AXI4PIFB @ 40400000 <= 1234
  260 AXI4Lite halfword upper bytes read from AXI4PIFB @ 40400002 <= 5678
  268 AXI4Lite single word read from AXI4PIFB         @ 40400000 <= 56781234
  276 AXI4Lite byte 0 write to AXI4PIFB               @ 40400000 <= 01
  284 AXI4Lite byte 1 write to AXI4PIFB               @ 40400001 <= 02
  292 AXI4Lite byte 2 write to AXI4PIFB               @ 40400002 <= 03
  300 AXI4Lite byte 3 write to AXI4PIFB               @ 40400003 <= 04
  308 AXI4Lite byte 0 read from AXI4PIFB              @ 40400000 <= 01
  316 AXI4Lite byte 1 read from AXI4PIFB              @ 40400001 <= 02
  324 AXI4Lite byte 2 read from AXI4PIFB              @ 40400002 <= 03
  332 AXI4Lite byte 3 read from AXI4PIFB              @ 40400003 <= 04
  340 AXI4Lite single word read from AXI4PIFB         @ 40400000 <= 04030201
100347   NOTE: Testbench simulation successful!
```

3. Create Block Design

Block design is added following cookbook chapter 5. After generating block design, corresponding component instantiation is adjusted from top module, using a HDL wrapper. In the process, it was necessary to remember to make a new project with the top module as main component.

4. Custom RAM IP from Xilinx IP block RAM

Custom IP is generated from following cookbook chapter 5.3

For simulation, ram_lab4_sim_netlist.vhdl is added to previous questa simulation project. Dummy processor is added again for the simulation.

Running the simulation as before result in error as reset value to the ram module is no longer 'DEADBEEF'. Changing the MC check value to x"00000000" returns successful simulation.

5. synthesis on ram_lab4 module and compare with Xilinx IP version

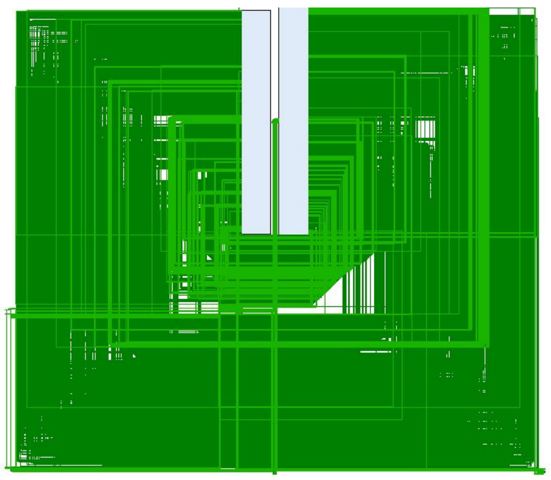
Synthesis of the given ram_lab4 module ends up with 24% resource usage, whereas using IP only uses 0.40% in total. Following tables is taken from utilization report from both synthesises.

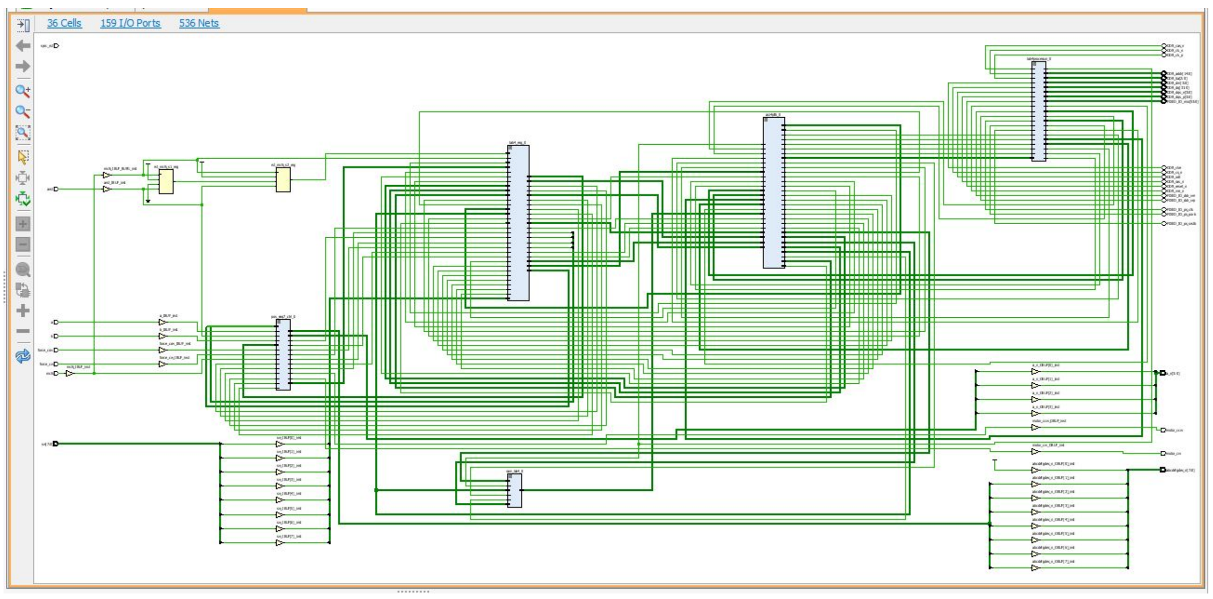
The synthesis, implementation and bitstream generation runs much faster using IP, as a result of the more efficient resource usage.

```
+-------------------------+-------+-------+-----------+-------+
|        Site Type        |  Used | Fixed | Available | Util% |
+-------------------------+-------+-------+-----------+-------+
| Slice LUTs*             | 12791 |     0 |     53200 | 24.04 |
|   LUT as Logic          | 12791 |     0 |     53200 | 24.04 |
|   LUT as Memory         |     0 |     0 |     17400 |  0.00 |
| Slice Registers         | 33772 |     0 |    106400 | 31.74 |
|   Register as Flip Flop | 33772 |     0 |    106400 | 31.74 |
|   Register as Latch     |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                |  4353 |     0 |     26600 | 16.36 |
| F8 Muxes                |  2176 |     0 |     13300 | 16.36 |
+-------------------------+-------+-------+-----------+-------+
```

```
+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  215 |     0 |     53200 |  0.40 |
|   LUT as Logic          |  215 |     0 |     53200 |  0.40 |
|   LUT as Memory         |    0 |     0 |     17400 |  0.00 |
| Slice Registers         |  374 |     0 |    106400 |  0.35 |
|   Register as Flip Flop |  374 |     0 |    106400 |  0.35 |
|   Register as Latch     |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                |    3 |     0 |     26600 |  0.01 |
| F8 Muxes                |    0 |     0 |     13300 |  0.00 |
+-------------------------+------+-------+-----------+-------+
```

Captures of the schematic before and after the switch from RAM file to the IP clearly demonstrates the efficiency.





6. C-program

SDK project is set up following the cookbook 2.4. Then ‘hello world’ code is used as template to write some sp values to the register. It was important to point at the address incrementing with 0x10 from the base address.

```
        #include <stdio.h>
        #include "platform.h"
        #include "xil_printf.h"
        #include "xparameters.h"
        #include "xil_io.h"
        #include <sleep.h>



        int main()
```

```
{
    init_platform();
    while(1){
    Xil_Out8(XPAR_M00_AXI_BASEADDR+0x10, 0x71);
    sleep(5);
    Xil_Out8(XPAR_M00_AXI_BASEADDR+0x10, 0x3d);
    sleep(3);
    Xil_Out8(XPAR_M00_AXI_BASEADDR+0x10, 0x4f);
    sleep(3);
    Xil_Out8(XPAR_M00_AXI_BASEADDR+0x10, 0x20);
    sleep(5);
    }
    cleanup_platform();
    return 0;
}
```

7. Adding system_ILA

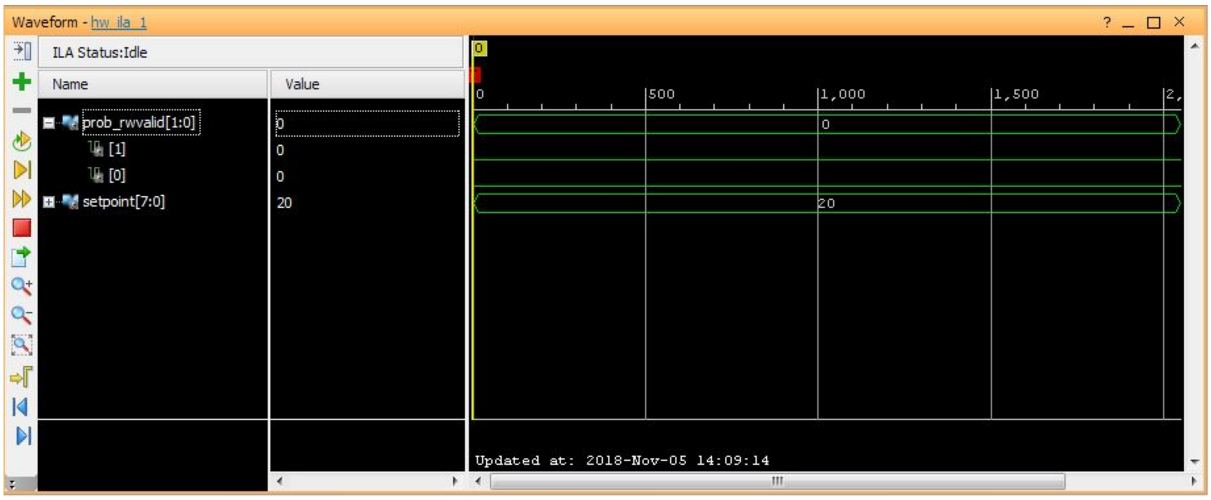
ILA was implemented from integrated IP selection with 2 probe with data depth of 2048. The two probe was set up for 2 bit for read and write valid signal and 8 bit for setpoint value. Then component instantiation and port mapping are added as following

```
-- instansiation
COMPONENT ila_0
PORT (
    clk : IN STD_LOGIC;
    probe0 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    probe1 : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
);
END COMPONENT  ;

-- signal
signal prob_rwvalid      : std_logic_vector(1 downto 0);

-- port mapping
ILA : ila_0
PORT MAP (
    clk => mclk,
    probe0 => prob_rwvalid,
    probe1 => setpoint
);

-- concurrent statement
prob_rwvalid <= pifb_axi_arvalid & pifb_axi_awvalid;
```



After re-synthesis and implementation, the program is uploaded with added ILA module. Probing was successful as show on the figure above.

8. writing 20 positions to RAM module and use the values to move motor

Using ram base address defined in lab4_pck.vhd, a short C-program is written to utilize RAM module. At the start of the program, it writes 20 values incrementing from 0 with interval 4 to the address starting from the base address. Then these addresses are accessed and read values are written to register module.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xil_io.h"
#include <sleep.h>

#define RAM_BASEADDR 0x4040000

int main()
{
    init_platform();

    for (int i = 0 ; i < 20 ; i++ ){
        Xil_Out8(RAM_BASEADDR+i*8, 0+8*i);
    }

    for (int i = 0 ; i < 20 ; i++ ){
        unsigned int rval = Xil_In8(RAM_BASEADDR+i*8);
        Xil_Out8(XPAR_M00_AXI_BASEADDR+0x10, rval);
        sleep(3);
    }
    cleanup_platform();
    return 0;
}
```