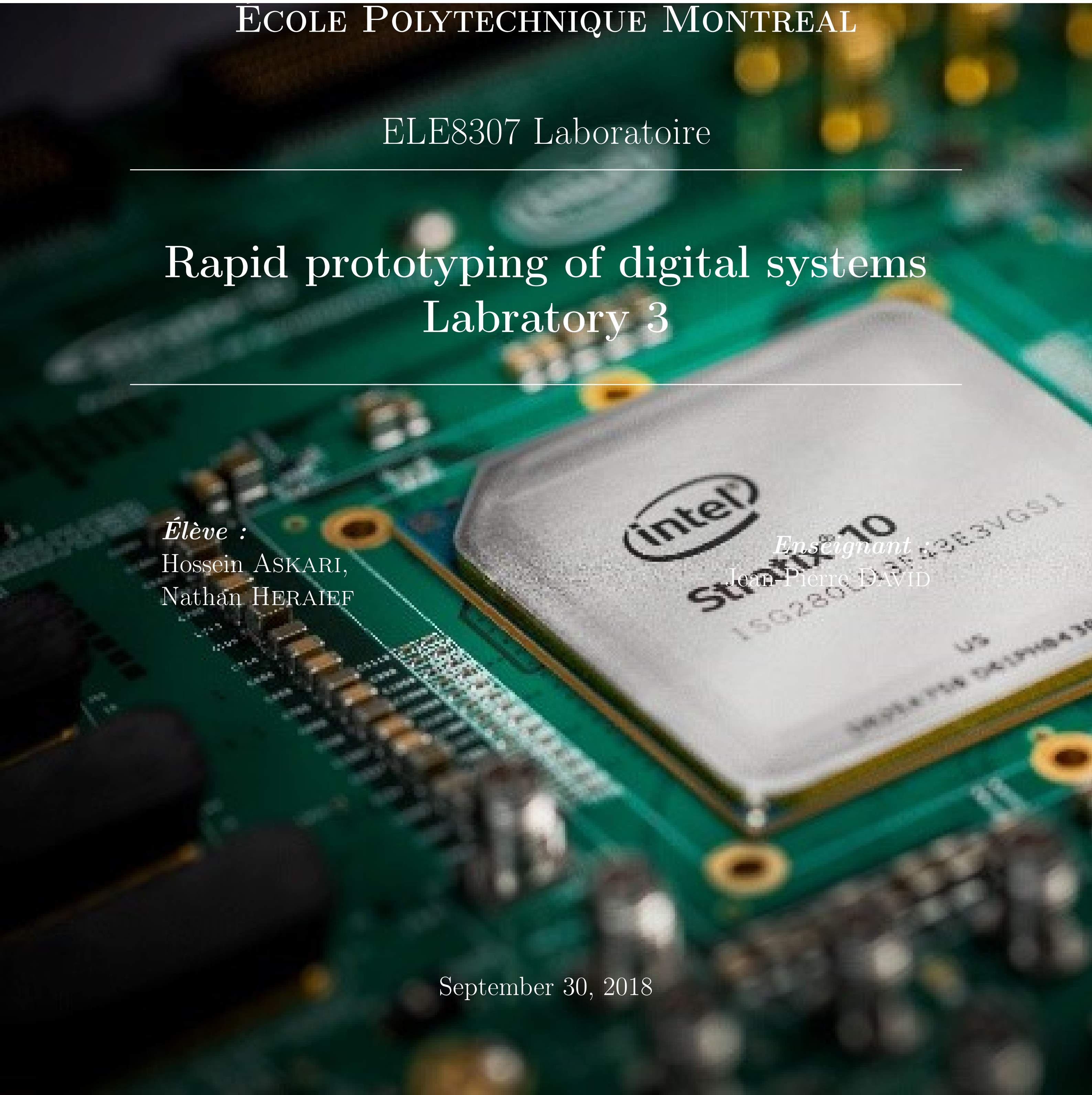
École Polytechnique Montreal

ELE8307 Laboratoire

Rapid prototyping of digital systems
Labratory 3



Élève :
Hossein Askari,
Nathan Heraief

Enseignant :
Jean-Pierre David

September 30, 2018

Contents

1 Lab3 Overview

In this Lab, we will go deeper in the understanding of Intel's NIOSII soft processor. The idea is to start creating our own specialized instructions. These instructions can be combinatorial, multi-cycle and extended. They will allow us to remove the VGA driver used in Lab2 and to implement our own specialized driver.

The lab3 is split in 3 parts :

- Part 1 : Implementation of a floating point Hardware
- Part 2 : Specialized instructions for VGA display
- Part 3 : Specialized instructions to draw diagonal lines

We used **set_pixel** and **draw_line** custom instructions developed in this lab to further optimize the **SolarSystem3D** application introduced in previous lab.

2 Part1: Instructions spécialisées pour le calcul en point flottant

In this part, we were asked to add a floating point hardware to our NIOSII processor. A FPU is a coprocessor specially designed to operate on floating point numbers with operations such as : addition, soustraction, division and multiplication. This addition should speed up the process by creating different macros in *system.h* file. So these macros can be called from the software and are directly linked to hardware tasks.

Operation [4]	N [5]	Cycles	Result	Subnormal	Rounding	GCC Inference
fdivs	255	16	a ÷ b	Flush to 0	Nearest	a / b
fsubs	254	5	a – b	Flush to 0	Faithful	a – b
fadds	253	5	a + b	Flush to 0	Faithful	a + b
fmuls	252	4	a x b	Flush to 0	Faithful	a * b
fsqrts	251	8	$\sqrt{a}$		Faithful	sqrtf() [6]

Figure 1: Basic floating point Operations available in Nios II

Figure 1 illustrates the available fpu operations in NiosII processor. This is the fist step to optimize the application with specialized instructions. Figure 2

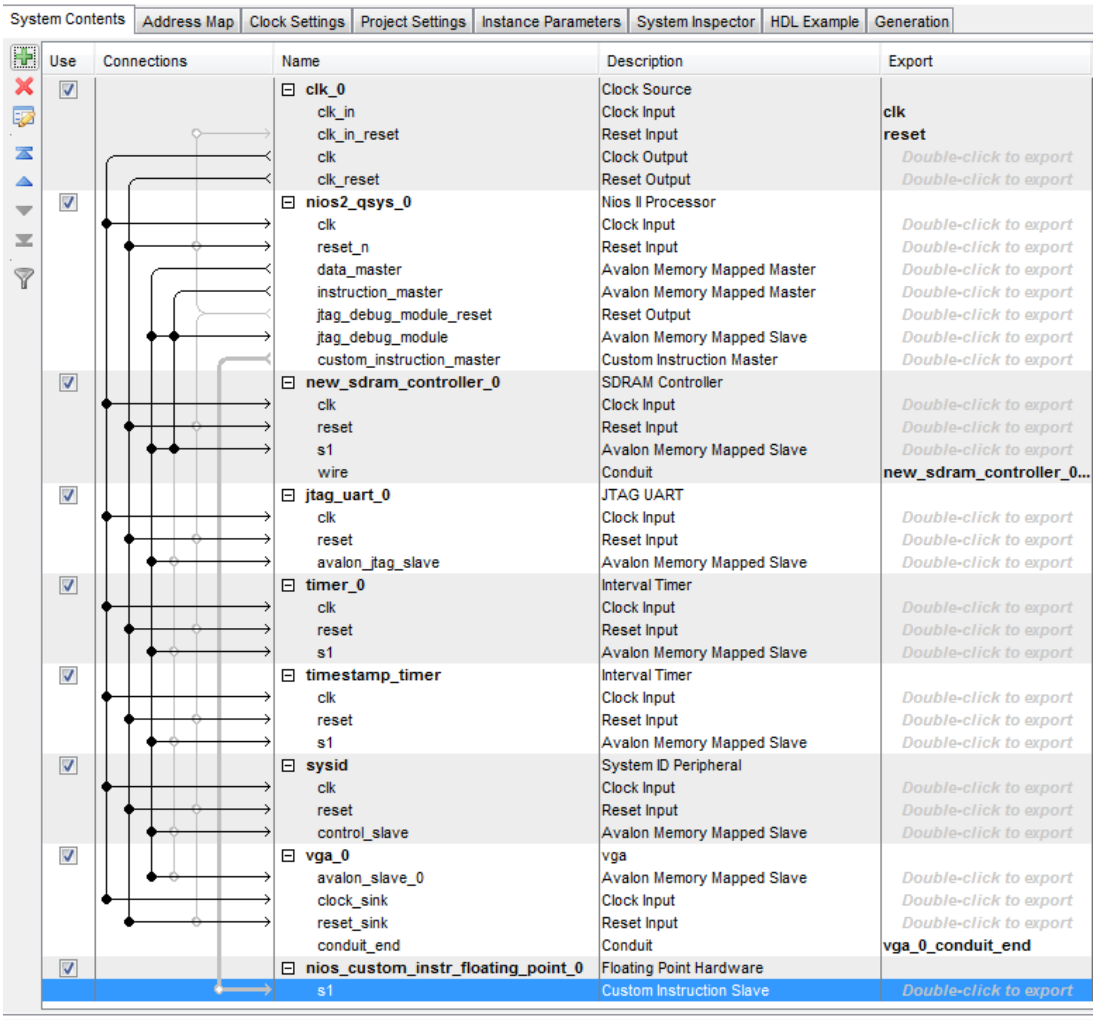


Figure 2: Illustrating Qsys system which shows the floating-point unit is added to NiosII processor.

3 Part 2: Instructions spécialisées pour l'affichage VGA

In this part of the lab, we were asked to :

- First, design a circuit that, given x, y calculate the address where the input color should be stored.
- Second, design a specialized hardware to communicate with VGA controller and calculate address to write in VGA buffer.

For each part, we where asked to connect the specialized hardware to the NiosII processor by means of custom instructions.

3.1 Calcul d'adresse

The idea here is to use a specialized instruction to calculate the address of each pixel. By using a combinatorial instruction, we can calculate a new address in one clock cycle. The following shows, given the input x and y, how $ADDR$ is calcuated.

$$ADDR = \frac{VGA\_0\_BASE}{4} + x + 640 * y$$

We were asked to not use a multiplication in our design. Hence, the above formula becomes as follow:

$$ADDR = \frac{VGA\_0\_BASE}{4} + x + y << 7 + y << 9$$

This is because we can write 640 as sum of 512 (shift left by 9) and 128 (shift left by 7). This is simply accomplished in hardware. The following shows how to design this circuit in systemverilog:

```
module setPixel
    (
        input  logic signed [31:0] dataa, //x
        input  logic signed [31:0] datab, //y
        output logic signed [31:0] result
    );
    logic signed [31:0] BASE_ADDR = 32'h0100_0000;
    assign result = (BASE_ADDR>>2) + dataa + (datab<<7) + (datab<<9);
endmodule // setPixel
```

Figure 3: Systemverilog code showing how to calculate address for VGA buffer based on the input x and y.

The simple circuit that is illustrated in Figure 3 can communicate with Nios II processor. Again, like Part1 of this lab, we used Qsys to add this hardware to our system, only this time, we used it as a custom instruction. NiosII supports multiple format of custom instructions. According to Intel's Nios II Custom Instruction User Guide, for a instructions that are implemented as a Combinational hardware, only `dataa, datab` and `result` are needed to be handled. After we were satisfied with hardware implementation, we used Qsys and Quartus to generate BSP and software project. This step automatically added a instruction to our software toolchain. The following code snippet shows that for our implementation, the added custom instruction is named `ALT_CI_SETPIXEL_0`

```
void ecran2d_setPixel( int x, int y, int color ) {
    int addr  = ALT_CI_SETPIXEL_0(x, y);
    IOWR(0, addr,color);
}
```

Figure 4: C Code snippet showing the use of custom instruction for calculating the address.

3.2 Contrôle direct du contrôleur VGA

In this part of the lab, we used what we learned so far in this lab to design a VGA controller to communicate with NiosII processor through Multicycle custom instruction interface to write to VGA buffer.

We were asked to design the controller using a state machine. The following code snippet show our VGA controller written in systemverilog.

```
    always @(posedge clk) begin
        if(reset) begin
            wr     <= 1'b0;
            etat   <= FETCH_DATA;
            done   <= 1'b0;
            addr   <= {19{1'b0}};
            result <= {32{1'b0}};
            data   <= {32{1'b0}};
        end else begin
            if(clk_en) begin
                case(etat)
                    FETCH_DATA:
                        begin
                            if( start == 1'b1 ) begin
                                x      <= dataa[15:0];
                                y      <= dataa[31:16];
                                color  <= datab;
                                etat   <= WAIT_FOR_BUSY;
                            end else begin
                                etat   <=  FETCH_DATA;
                            end;
                            done   <= 1'b0;
                            wr     <= 1'b0;
                        end
                    WAIT_FOR_BUSY:
                        begin
                            if( busy == 1'b0 ) begin
                                done   <= 1'b1;
                                wr     <= 1'b1;
                                result <= dataa; // debug feature!
                                addr   <= x + (y<<9) + (y<<7) ;
                                data   <= color;
                                etat   <= FETCH_DATA;
                            end else begin
                                etat <= WAIT_FOR_BUSY;
                            end
                        end
                endcase
            end
        end
    end
```

Figure 5: Code snippet showing the state machine part of the VGA controller.

Figure 5 illustrates the state machine part of our VGA controller. As it can be seen, the vga controller has two states, namely `FETCH_DATA` and `WAIT_FOR_BUSY`. The general functionality is to loop in `FETCH_DATA` state until a `start` signal is received form processor. At this point, the inpt data (`dataa` and `datab`) are sampled and the next state is set to `WAIT_FOR_BUSY`. In this state (`WAIT_FOR_BUSY`), we will wait for VGA to `busy` signal to go low which indicates that the vga buffer is ready to be written to. If that's the case, we calculate the addr which we want to write to and set the state to `FETCH_DATA` again.

Warning!

While this code perfectly works for low speed refresh rate, in higher speed (for instance when we send data in burst to vga controller), this controller will not work. A better solution would be to use a fifo in produce/consume manner. We can also design a data sampler circuit to sample data regardless of state machine state.

On the software side, after generating the BSP and NiosII system, we used the generated custom instruction, namely `ALT_CI_SETPIXEL_0`, to send the controller the x, y and color. Unlike the previous part, we had to design the custom instruction in multicycle format. Since we can only send two 32-bit variables to HW in this format, we combined x and y together and pass them to the HW through `dataa` variable. This forces us to use an extra instruction to prepare `dataa` correctly for our custom instruction. Figure 6 shows how we did this in sw.

```c
void ecran2d_setPixel( int x, int y, int color ) {
    int addr;
    x =  ((y & 0x0000FFFF)<<16) + (x & 0x0000FFFF);
    addr = ALT_CI_SETPIXEL_0(x, color);
    //IOWR(0, addr,color);
}
```

Figure 6: Custom instruction to send data to hardware to calculate the correct addr in VGA buffer to write color into.

Finally, Figure 7 illustrates the GNU performance report after we used our custom instruction.

```
Flat profile:

Each sample counts as 0.001 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 17.73     10.60    10.60   893507     0.00     0.00  __muldf3
 16.82     20.67    10.06  2096353     0.00     0.00  __pack_d
 10.49     26.94     6.27   935469     0.00     0.00  _fpadd_parts
  9.76     32.78     5.84  3570444     0.00     0.00  __muldi3
  8.22     37.70     4.92   109036     0.00     0.00  __divdf3
  7.39     42.12     4.42  4230442     0.00     0.00  __unpack_d
  5.80     45.59     3.47                             read
  5.35     48.80     3.20   109362     0.00     0.00  __ieee754_sqrt
  1.60     49.75     0.95    38848     0.00     0.00  __ieee754_acos
  1.32     50.54     0.79   499950     0.00     0.00  __subdf3
  1.12     51.21     0.67   199026     0.00     0.00  __clzsi2
  0.97     51.79     0.58    39564     0.00     0.00  ss_planet_getBall3DIntensity
  0.95     52.36     0.57   240911     0.00     0.00  __pack_f
  0.85     52.87     0.51   435519     0.00     0.00  __adddf3
  0.76     53.32     0.45    79340     0.00     0.00  sqrt
  0.72     53.75     0.43    79776     0.00     0.00  __make_fp
  0.68     54.16     0.41    17922     0.00     0.00  __kernel_cos
  0.62     54.53     0.37   161152     0.00     0.00  __floatsisf
  0.62     54.90     0.37    22078     0.00     0.00  __kernel_sin
  0.61     55.27     0.36    79776     0.00     0.00  __truncdfsf2
```

Figure 7: GNU performance report generated for `ALT_CI_SETPIXEL_0`

4 Part3: Tracé de lignes obliques

In this part of the lab, we were asked to design a circuit to use the Extended Multi Cycle Instructions in NiosII processor to draw a line. As suggested by the lab document, we started from the software implementation. We were asked to inspire from `ss_orbit_line_draw` function to draw a line given `x0`, `y0`, `x1` and `y1`. The first task for us was to design a state machine that would perform the same task. For that, we changed the code in `ss_orbit_line_draw` function to the following:

```
void ss_orbit_line_draw( ss_orbit_line_t *line, int color ) {
    int Dx,Dy, steep;
    int ystep, xstep,TwoDy, TwoDyTwoDx, E, xDraw, yDraw, x,y;
    int x0,y0,x1,y1;
    Input:      x0 = line->x0; x1 = line->x1;y0 = line->y0;y1 = line->y1;Dx = x1 - x0;Dy = y1 - y0;steep = (abs(Dy) >= abs(Dx));
                goto I0;
    I0:     if(steep)   {swap(&x0, &y0);swap(&x1, &y1);Dx = x1 - x0;Dy = y1 - y0;goto I0n;}
    I0n:    if(1)       xstep = 1; T1:=(Dx < 0); goto I1;
    I1:     if(T1)  {xstep = -1;Dx = -Dx;goto I1n;}
    I1n:    if(1)       ystep = 1; T2:=(Dy < 0); goto I2;
    I2:     if(T2)  {ystep = -1; Dy = -Dy;goto I2n;}
    I2n:    if(1)       TwoDy = 2*Dy; TwoDyTwoDx = TwoDy - 2*Dx;E = TwoDy - Dx; y = y0; x = x0; T3:=(x != x1); goto W0;
    W0:     if(T3){     goto I3;
    I3:     if (steep)  { xDraw = y; yDraw = x; goto I3n;}
            else        {xDraw = x; yDraw = y; goto I3n;}
    I3n:    if(1)       {ecran2d_setPixel( xDraw,yDraw, color); T4:=(E > 0);goto I4;}
    I4:     if(T4)   {E += TwoDyTwoDx; y = y + ystep; x += xstep; T3:=(x != x1); goto W0;}
            else        { E += TwoDy; x += xstep; T3:=(x != x1); goto W0;}
    }
}
```

Figure 8: GNU performance report generated for `ALT_CI_SETPIXEL_0`

As it can be in Figure 8, the original code is transferred to a state machine with 11 states.

After we verified that the new c-code still works, we tried to implement the state machine in hardware. Again, we used SystemVerilog to design the state machine. Looking deeper inside the state machine, we found that 3 of these states can be some how combined with other states. So, our final design had only 8 states. Since we had to support the previous functionality (which was the ability to set a pixel), we used the extended version of the NiosII custom instruction. The interface to our module is illustrated in Figure 9.

```
module drawLine
    (
        output logic [18:0] addr   ,
        output logic [31:0] data   ,
        output logic        wr     ,
        input  logic        busy   ,

        input  logic        clk    ,
        input  logic        reset  ,
        input  logic        clk_en,
        input  logic        start  ,
        output logic        done   ,
        input  logic [31:0] dataa  ,
        input  logic [31:0] datab  ,
        input  logic  [7:0] n      ,
        output logic [31:0] result
    );
```

Figure 9: This code snippet shows the interface to our `drawLine` module.

As it can be seen in Figure 9, the interface to our custom hardware is the same as before except this time we are also passing `n` to the module. According to Intel's Nios II Custom Instruction User Guide, in extended custom instruction, a part from `dataa` and `datab`, we can pass `n` (which can have up to 8 bits) to send extra information to the hardware. We used `n` to distinguish between to type of instructions. The first is the normal operation mode. In this mode, the address is calculated as before (Part2.2). In this mode we set `n=0`. This will bypass most of the internal states in the vga controller. In the second mode (`n=1`), we calculate the coordinates dots within the line and we write them to the vga buffer one after another.

With this in mind, we designed the `drawLine` module and as before, we used Qsys to setup our NiosII based system. Figure 11 illustrates the final system.

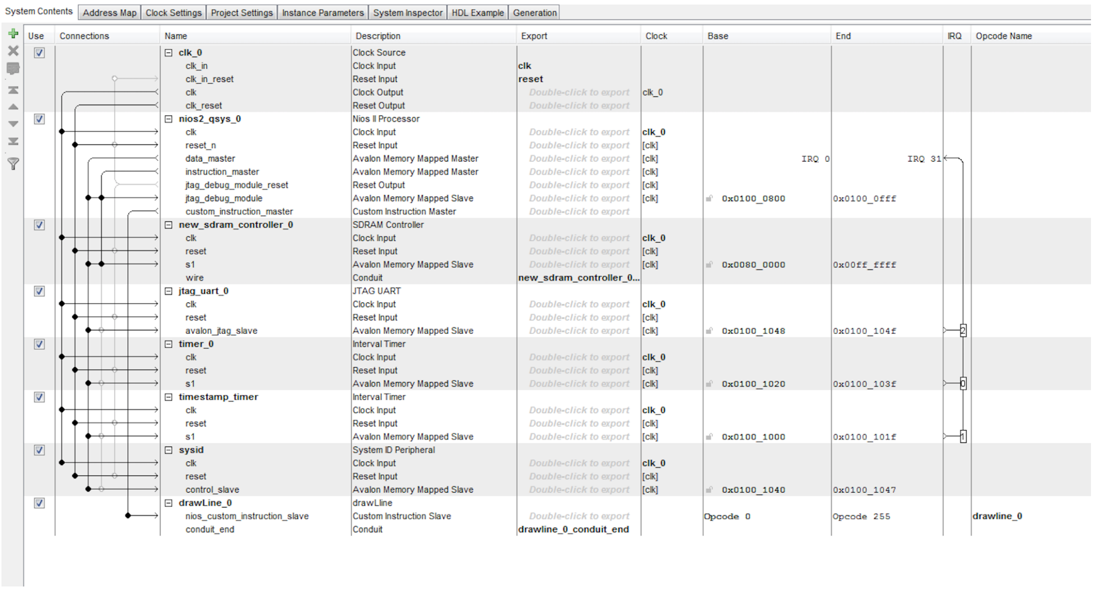


Figure 10: Qsys system that has draw line module connected to the processor.

As it can be seen, we only have `drawLine` module as hw for the custom instruction. Finally, we used this custom instruction in our code. The following code snippet illustrates the `ss_orbit_line_draw` function which utilizes the new custom instructions.

```c

void ss_orbit_line_draw( ss_orbit_line_t *line, int color ) {

    int word1, word2, ret;
    int x0,y0,x1,y1;
    int Dx,Dy, steep;
    int ystep, xstep,TwoDy, TwoDyTwoDx, E, xDraw, yDraw, x,y;
    x0 = line->x0;
    x1 = line->x1;
    y0 = line->y0;
    y1 = line->y1;
    word1 =  ((x1 & 0x000003FF)<<20) + ((y0 & 0x000003FF)<<10) + (x0 & 0x000003FF);
    word2 =  ((color & 0x000003FF)<<10) + (y1 & 0x000003FF);
    ret   =  ALT_CI_DRAWLINE_0(1, word1, word2);
}
```

Figure 11: Code snippet that shows how we used the new custom instruction to draw a line.

As it was explained before, since we only have `dataa` and `datab` to send the parameters of the line to the hw, we have to encode them within these two variables. Figure 11 illustrates that we encode `x1,y0` and `x0` inside `dataa` and we encode `y1` and `color` inside `datab`. We use 10bits for each of them since vga coordinates can only take values between `0` to 480 for `y` and `0` to 640 for `x`. And for color we still need no more than 10bits.

In the last part of this lab, we were asked to write a code to test our custom instruction. We wrote a code to draw a cross on the monitor. The following code snippet shows how we wrote this.

```
void draw_cross(int with_specialization)
{
    ss_orbit_line_t line;
    int color = 255;
    ecran2d_clear();
    line.x0 = 0;
    line.y0 = 0;
    line.x1 = 640;
    line.y1 = 480;
    ss_orbit_line_draw(&line, color, with_specialization);
    line.x0 = 640;
    line.y0 = 0;
    line.x1 = 0;
    line.y1 = 480;
    ss_orbit_line_draw(&line, color, with_specialization);
}

int main()
{
    draw_cross(1);

    exit(0);

 }
```

Figure 12: Code snippet that shows how to draw a cross on the vga monitor using the created custom instruction.

The rest of this code is to import the necessary functions from the original code so that the data types such as `ss_orbit_line_t` code be used.