Hochschule Bremerhaven

PI Controller Hardware IP and Software Driver

LAB REPORT

Elemental in the degree of

MASTERS OF SCIENCE IN EMBEDDED SYSTEMS AND DESIGN

By

Arjun Hugar Jagannath [MR.NO: 41583]

Under the guidance of

Prof. Dr. -Ing. Kai Müller

Subject

SYSTEM ON CHIP DESIGN LAB-24

Date of Submission: 05.07.2024

DECLARATION

I affirm that, in my own words, this written submission captures the concept. Additionally, I certify that I have followed all academic honesty and integrity guidelines and have not falsified, fabricated, or misrepresented any ideas, data, facts, or sources in my submission. I am aware that breaking any of the aforementioned will result in disciplinary action from the Institute and may lead to legal action against the sources who were improperly followed or from whom appropriate authorization was not obtained when required.

Arjun Hugar Jagannath

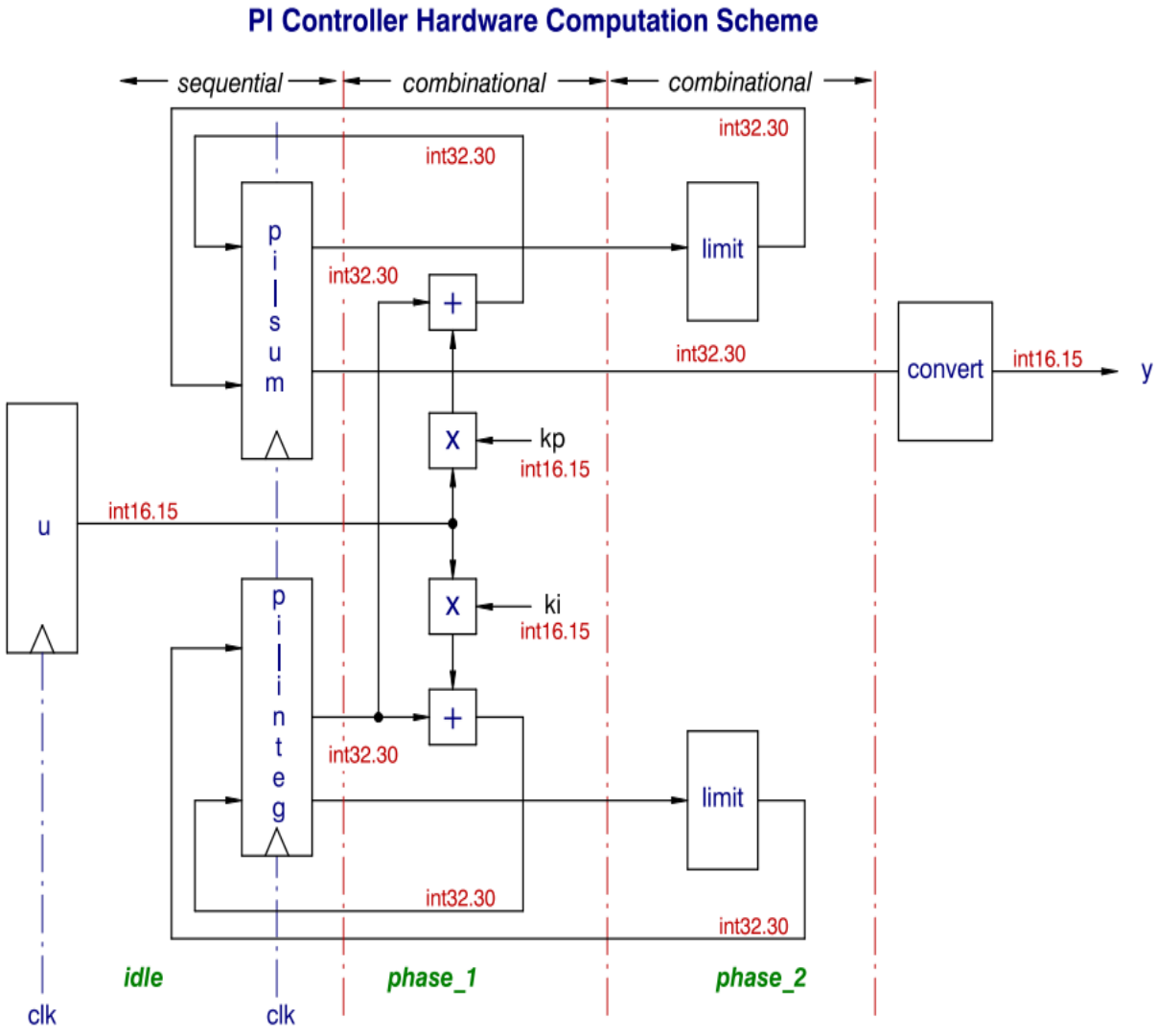
CONTENTS

1. INTRODUCTION

A group of equipment intended to manipulate logical data or physical quantities represented in a digital format is referred to as a digital system. The processing of discrete signals is a characteristic of these systems. Digital systems are widely used and play an important part in the field of electronics. In electronic design automation, a specialised language known as Very High-Speed Integrated Circuit Hardware Description Language (VHDL) is frequently used to define and specify the behaviour of digital systems.

In order to support the use of digital systems, this paper focuses on creating an application that displays a moving LED light pattern. This application acts as an excellent example to highlight the capabilities and efficacy of digital systems by utilising VHDL and the fundamental concepts of digital systems.

2.Explanation of 16-bit PI Controller



A Proportional-Integral (PI) controller with fixed-point arithmetic is represented by the hardware implementation in the diagram. Three phases comprise the whole operation: idle, phase_1, and phase_2. Each signal has its bit-width and fractional bits specified. A detailed breakdown of each stage is provided below:

Certainly! Let's go through each phase in detail, considering each block and the importance of integer and fractional bits in the conversion and computation processes.

Idle Phase:

Input u

- The format of the input to the PI controller, u, is int16.15. This indicates that it employs a total of 16 bits 1 integer bit and 15 fractional bits.

Registers pilsum and pilinteg

- The registers pilsum and pilinteg are used to store the intermediate results of the PI controller in the format int32.30. This indicates that they utilize a total of 32 bits 30 fractional bits and 2 integer bits.

Phase 1:

Multiplication and Addition

1. Proportional Path Multiplication by kp
   The proportionate gain kp in int16.15 is multiplied by the input u in int16.15. Since the integer part of a multiplication in int16.15 format requires 2 bits and the fractional part requires 30 bits, the result of this multiplication is in int32.30 format. This is calculated in the manner described below:
   u*kp = int16.15 * int16.15 = int32.30

2. Integral Path Multiplication by ki and Summation
   The integral increase ki in int16.15 is multiplied by u simultaneously. Additionally, the multiplication's output is formatted in int32.30. After that, this product is added to the value kept in the pi_linteg register, which is formatted in int32.30. After this summing, pi_linteg has an updated value:
   New_pi_linteg = old_pi_linteg + u * ki = int32.30 + int32.30 = int32.30.

Phase 2

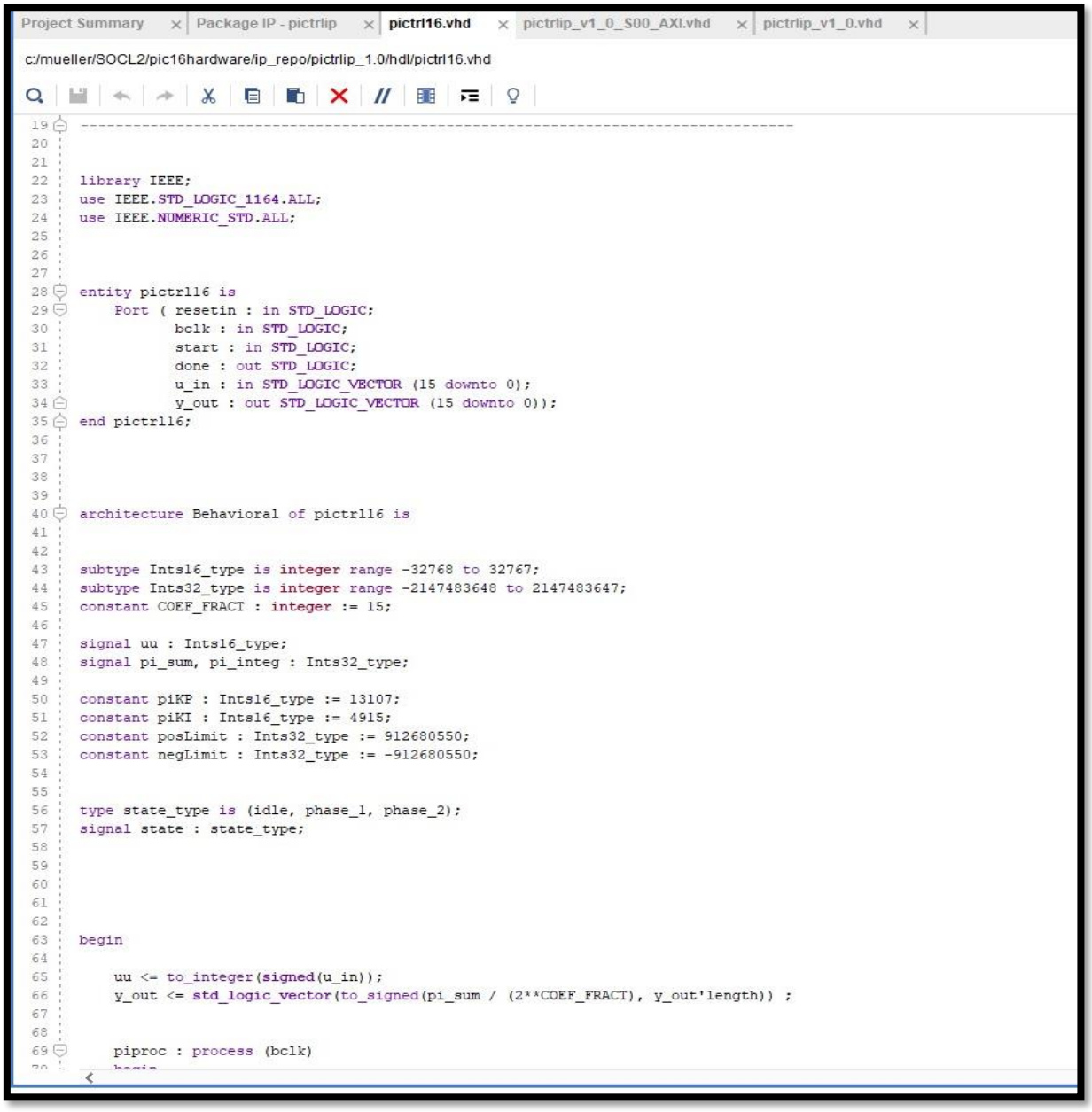
Summation and Limiting

1. Combined Summation:
   Pi_linteg in int32.30 stores the updated integral term, which is added to the output of the proportional path, u * kp. This addition's outcome is still in the format of int32.30:
   combined_sum = u * kp + new_pi_linteg = int32.30 + int32.30 = int32.30.

2. Limiting:
   In order to prevent overflow or underflow, the combined sum is run through a limiting block to make sure it stays within the set bounds. The int32.30 format of the limited value is still used.

3. Conversion to Output Format:
   The limited value, still in int32.30 format, needs to be converted to int16.15 format for the final output y. This conversion involves reducing the bit-width from 32 to 16 bits. Typically, this means taking the most significant 16 bits of the int32.30 value, including any necessary rounding or truncation. The result of this conversion is the final output y in int16.15 format.

3. Hardware Design VHDL Code:



Project Summary | Package IP - pictrlip | pictrl16.vhd * | pictrlip_v1_0_S00_AXI.vhd | pictrlip_v1_0.vhd

c:/mueller/SOCL2/pic16hardware/ip_repo/pictrlip_1.0/hdl/pictrl16.vhd

```
63  begin
64
65      uu <= to_integer(signed(u_in));
66      y_out <= std_logic_vector(to_signed(pi_sum / (2**COEF_FRACT), y_out'length)) ;
67      piproc : process (bclk)
68      begin
69          if rising_edge(bclk) then
70              if resetin = '0' then
71                  state <= idle;
72                  pi_sum <= 0;
73                  pi_integ <= 0;
74              else
75                  done <= '0';
76
77                  case state is
78                      when idle =>
79                          done <= '1';
80                          if start = '1' then
81                              state <= phase_1;
82                          end if;
83                      -- calculating The value of pi_sum and pi_integ
84                      when phase_1 =>
85                          pi_sum <= piKP * uu + pi_integ ;
86                          pi_integ <= piKI * uu + pi_integ;
87                          state <= phase_2;
88
89
90                     -- Limiting the value of pi_sum and pi_integ in the range if posLimit and negLimit
91                      when phase_2 =>
92                          if pi_sum > posLimit then
93                              pi_sum <= posLimit;
94
95                          else if pi_sum < negLimit then
96                              pi_sum <= negLimit;
97                          end if;
98                          end if;
99
100                         if pi_integ > posLimit then
101                             pi_integ <= posLimit;
102
103                         else if pi_integ < negLimit then
104                             pi_integ <= negLimit;
105                         end if;
106                         end if;
107                         state <= idle;
108                 end case;
109             end if;
110         end if;
111     end process piproc;
112 end Behavioral;
113
```

4. Testbench Code

Project Summary | pictrl16.vhd | pictrl16_tb.vhd

C:/mueller/SOCL2/pictrl16/pictrl16.srcs/sim_1/new/pictrl16_tb.vhd

```
20
21
22  library IEEE;
23  use IEEE.STD_LOGIC_1164.ALL;
24  use IEEE.NUMERIC_STD.ALL;
25
26
27  entity pictrl16_tb is
28      --  Port ( );
29  end pictrl16_tb;
30
31  architecture Behavioral of pictrl16_tb is
32
33      COMPONENT pictrl16 IS
34          PORT ( resetn : in STD_LOGIC;
35               bclk : in STD_LOGIC;
36               start : in STD_LOGIC;
37               done : out STD_LOGIC;
38               u_in : in STD_LOGIC_VECTOR (15 downto 0);
39               y_out : out STD_LOGIC_VECTOR (15 downto 0));
40      end COMPONENT pictrl16;
41
42      SIGNAL resetn : STD_LOGIC := '1';
43      SIGNAL bclk : STD_LOGIC := '0';
44      SIGNAL start : STD_LOGIC := '0';
45      SIGNAL done : STD_LOGIC := '0';
46      SIGNAL u_in : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"6666"; ---0.8
47      SIGNAL y_out : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"0000";
48
49      CONSTANT clk_period : TIME := 10ns;
50  begin
51
52      uut : pictrl16
53          PORT MAP (  resetn  =>  resetn,
54                   bclk     =>  bclk,
55                   start    =>  start,
56                   done     =>  done,
57                   u_in     =>  u_in,
58                   y_out    =>  y_out  );
59
60      clk_p : PROCESS
61      BEGIN
62          WAIT FOR clk_period / 2;
63          bclk <= '1';
64          WAIT FOR clk_period / 2;
65          bclk <= '0';
66
67      END PROCESS clk_p;
68
69
70
71      stim_p : PROCESS
```

Project Summary | pictrl16.vhd | pictrl16_tb.vhd

C:/mueller/SOCL2/pictrl16/pictrl16.srcs/sim_1/new/pictrl16_tb.vhd

```
47      SIGNAL y_out : STD_LOGIC_VECTOR (15 DOWNTO 0) := x"0000";
48
49      CONSTANT clk_period : TIME := 10ns;
50  begin
51
52      uut : pictrl16
53          PORT MAP (  resetn  =>  resetn,
54                      bclk    =>  bclk,
55                      start   =>  start,
56                      done    =>  done,
57                      u_in    =>  u_in,
58                      y_out   =>  y_out  );
59
60      clk_p : PROCESS
61      BEGIN
62          WAIT FOR clk_period / 2;
63          bclk <= '1';
64          WAIT FOR clk_period / 2;
65          bclk <= '0';
66
67      END PROCESS clk_p;
68
69
70
71      stim_p : PROCESS
72      begin
73          WAIT FOR clk_period;
74          resetn <= '0';
75          WAIT FOR clk_period;
76          resetn <= '1';
77          FOR k IN 0 TO 9 LOOP
78              start <= '1';
79              WAIT FOR clk_period;
80              start <= '0';
81              WAIT FOR clk_period * 3;
82          END LOOP;
83
84          u_in <= STD_LOGIC_VECTOR(to_signed(-26214, u_in'length));
85          FOR k IN 0 TO 19 LOOP
86              start <= '1';
87              WAIT FOR clk_period;
88              start <= '0';
89              WAIT FOR clk_period * 3;
90          END LOOP;
91
92          WAIT;
93
94      END PROCESS stim_p;
95
96
97  end Behavioral;
```

5. VHDL MAIN CODE EXPLAINATION:

The code starts by including necessary libraries for standard logic and numerical operations. The entity pictrl16 is defined with the following ports: resetn, a reset signal in STD_LOGIC; bclk, a clock signal in STD_LOGIC; start, a start signals in STD_LOGIC; done, a done signal in STD_LOGIC; u_in, an input signal in STD_LOGIC_VECTOR of 16 bits; and y_out, an output signal in STD_LOGIC_VECTOR of 16 bits.

Architecture Declaration

- The architecture Behavioural is defined for the entity pictrl16.
- Subtypes for int16_type and int32_type are defined to represent integer ranges.
- The constant COEF_FRACT is set to 15 to represent the fractional bits.
- Internal signals uu, pi_sum, and pi_integ are declared with respective types.

Constants for Gains and Limits

- The proportional and integral gains, respectively, are represented by the constants pikP and pikI in int16_type.
- The positive and negative bounds in int32_type is represented by the constants posLimit and negLimit, respectively.

State Type and Signal

- Phase_1, Phase_2, and idle states define an enumerated type state_type.
- The current state of the controller is declared as a signal state of type state_type.

Main Process Block

- When the clock's rising edge, bclk, occurs, a process piproc is defined to initiate.
- Inside the process:
    - The status is set to idle and internal signals are reset if resetn is active low.
    - If the state is idle and start is active, the state transitions to phase_1.
    - In ‘phase_1’:
        - pi_sum is updated by adding the product of pikP and uu to pi_integ.
        - pi_integ is updated by adding the product of pikI and uu to the current pi_integ.
        - The state transitions to phase_2.

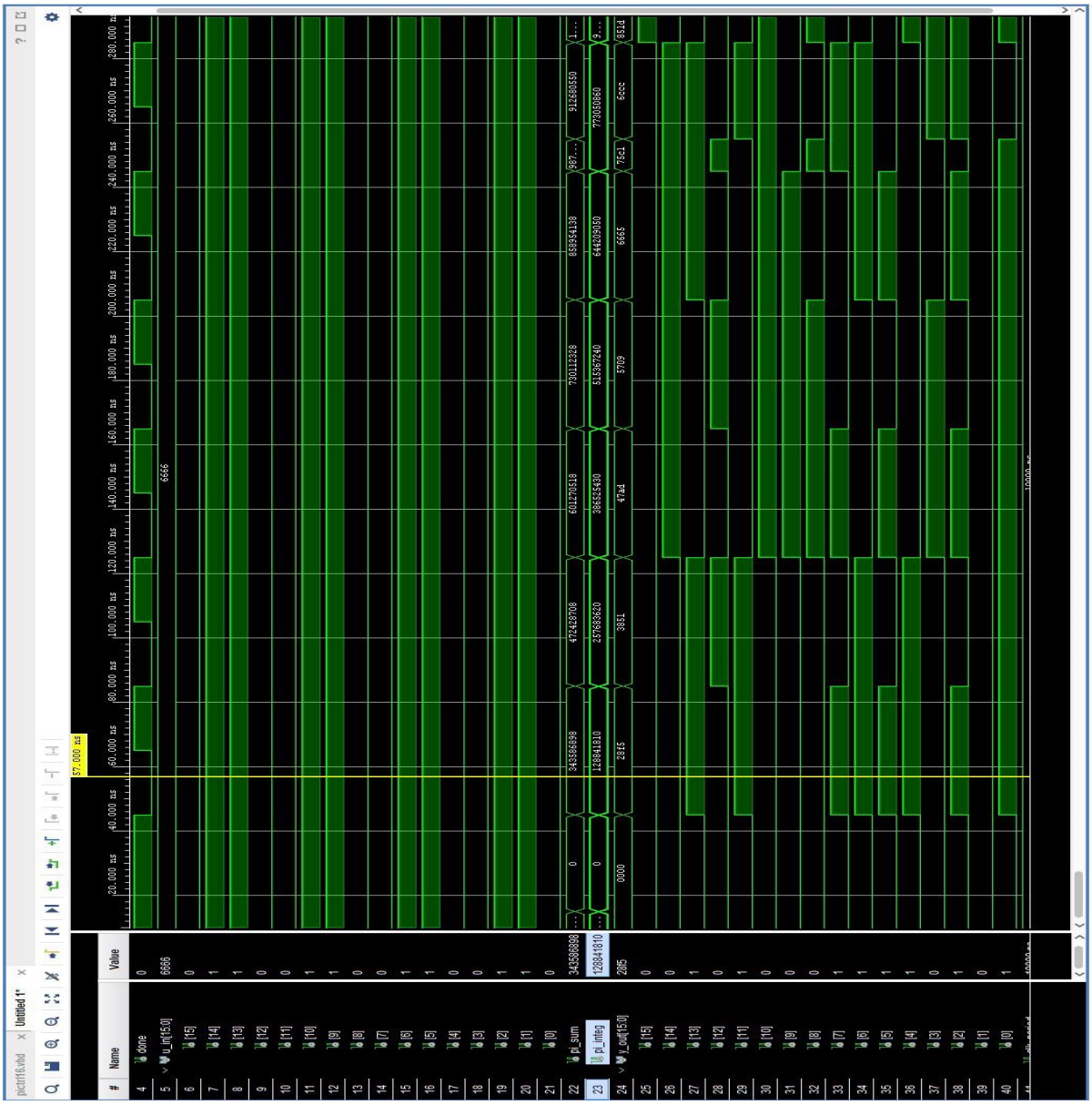
Output Assignment:

- The signal uu is assigned the integer conversion of the signed input u_in.
- The output y_out is assigned the signed conversion of pi_sum divided by a scaling factor to match the int16.15 format.

Summary

- The VHDL code implements a PI controller with initialization and sequential operations.
- The idle state initializes the controller.
- The phase_1 state performs proportional and integral path calculations.
- The phase_2 state limits the values to prevent overflow and transitions back to idle.
- The controller output is scaled and assigned to y_out in the int16.15 format.

6. Simulation Result:



7. Analysis:

Signals:

1. **resetn**: Active-low reset signal.
2. **bclk**: Base clock signal.
3. **start**: Signal to start the operation.
4. **done**: Signal indicating completion.
5. **pi_sum**: Sum value in the PI controller.
6. **pi_integ**: Integration value in the PI controller.
7. **state**: Current state of the state machine (e.g., idle, phase).
8. **u_in[15:0]**: Input signal u.
9. **y_out[15:0]**: Output signal y.

Observations:

1. **Clock and Reset**:
   - bclk toggles regularly, indicating the clock cycle.
   - resetn is high, implying the system is not in reset state.

2. **Start and Done**:
   - start is low, suggesting no new operation is triggered.
   - done transitions to high, signaling the completion of an operation cycle.

3. **PI Controller Values**:
   - **The values of pi_sum and pi_integ illustrate the integration and accumulation of inputs over time.**
   - pi_sum and pi_integ changes coincide with state transitions.

4. **State Transitions**:
   - **The status signal alternates between phase and idle to represent various operating phases.**
   - There are other phase states, probably processing phases, that follow each idle state.

5. **Input and Output**:
   - **The inputs values (6666, 285, etc) are displayed by u_in changing at predefined intervals.**
   - In proportion, y_out modifies, indicating the PI controller's output.

6. **Timing**:
   - The simulation lasts for 250ns, during which there are notable state transitions at 40, 60, 100, 140 and 180ns.
   - There seems to be a 40 ns duration for each operation cycle.

8. Explanation of C Code in Vitis

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


#define HW_REG(k) *(volatile unsigned int
*)(XPAR_PICTRLIP_0_S00_AXI_BASEADDR+4*k)
#define CBUF_LEN 64
```

Header File:

- #include <stdio.h>: Includes the standard input/output library for functions like printf and fgets.
- #include "platform.h": Includes platform-specific initialization and cleanup functions.
- #include "xil_printf.h": Includes a library for formatted printing, optimized for Xilinx platforms.
- #include "xparameters.h": Includes definitions for hardware parameters, such as base addresses for peripherals.

Macros:

#define HW_REG(k) *(volatile unsigned int *)
(XPAR_PICTRLIP_0_S00_AXI_BASEADDR+4*k):

- Hardware registers are accessed by this macro. A base address defined in xparameters is XPAR_PICTRLIP_0_S00_AXI_BASEADDR.k is multiplied by 4 (presuming a 4-byte register) and the offset h.
- #define CBUF_LEN 64: Defines the length of the character buffer used for input.

9.Main Function:

```
int main() {
    unsigned int xx;
    char cbuf[CBUF_LEN];
    int tarminate, k;
    int uu;
    short int yy;
    double yy_d;

    init_platform();

    print("--- PIC Test V0.a ---\n\r");

    tarminate = 0;
    do {
        print("> ");
        fgets(cbuf, CBUF_LEN, stdin);
        print("\n\r");
        if (cbuf[0] == 'x') {
            tarminate = 1;
        } else if (cbuf[0] == 'p') {
            xx = HW_REG(0);
            printf("PushB/Switches: %x\n\r", xx);
        } else if (cbuf[0] == 'l') {
            xx = 0;
            if (sscanf(&cbuf[1], "%x", &xx) != 1) {
                printf(" *** illegal int. \n\r");
            } else {
                HW_REG(0) = xx;
            }
        }
```

```
        else if (cbuf[0] == 's') {
          for (k = 0; k < 40; k++) {
            if (k < 10) {
              uu = 0;
            } else if (k < 20) {
              uu = 26214;
            } else {
              uu = -26214;
            }
            HW_REG(2) = *(unsigned int *)&uu;
            HW_REG(1) = 0;
            xx = HW_REG(1);

            if (xx == 0) {
              printf(" *** Done bit is 0. (%d)\n\r", xx);
            }

            xx = HW_REG(2);
            yy = *(short int *)&xx;
            yy_d = yy * 3.0517578125e-05;
            printf("%3d uu: %6d yy: %6d (%7.3f)\n\r", k, uu, yy, yy_d);
          }
        }
    } while (tarminate == 0);

    HW_REG(0) = 0x56;
    xx = HW_REG(0);
    printf("PushB/Switches: %x\n\r", xx);

    print("Thank you for using PIC test V0.a \n\r");
    cleanup_platform();
    return 0;
}
```

Initialization and Setup

1. Variable Declarations:
   - Unsigned int xx: A generated-purpose variable used to read and write values from registers.
   - char cbuf[CBUF_LEN]: Buffer for storing user input.
   - loop control variables are int tarminate, k.
   - int uu: The integer that the's' command block uses for calculations.
   - The's' command block uses a short integer, short int yy.
   - double yy_d: A floating-point variable with double precision for calculations on a scale.

2. **Platform Initialization**:
   - C Code = init_platform();
     Initializes the platform-specific settings, preparing the hardware and software environment.
3. Welcome Message:
   - C Code = print ("--- PIC Test V0.a ---\n\r");
     Prints a welcome message to indicate the start of the program.

Main Loop:

1. **Command Prompt and User Input**:

```
print("> "); fgets(cbuf, CBUF_LEN, stdin);
print("\n\r");
```

- Prompts the user with > and waits for input.
- Reads the input into cbuf.
- Prints a newline after reading input.

2. **Exit Condition**:

```
if (cbuf[0] == 'x') {
   tarminate = 1;
}
```

If the user inputs 'x', the program sets terminate to 1, which will break the loop.

3. **Read Register Value ('p' Command)**:

```
else if (cbuf[0] == 'p') {
   xx = HW_REG(0);
   printf("PushB/Switches: %x\n\r", xx);
}
```

- Reads the value from the hardware register at offset 0.
- Prints the value in hexadecimal format.

4. **Load Register Value ('l' Command)**:

```
else if (cbuf[0] == 'l') {
   xx = 0;
   if (sscanf(&cbuf[1], "%x", &xx) != 1) {
      printf(" *** illegal int. \n\r");
   } else {
      HW_REG(0) = xx;
   }
```

- Initializes xx to 0.
- Attempts to parse the user input following 'l' as a hexadecimal integer and store it in xx.
- If the input is invalid, prints an error message.
- Otherwise, writes xx to the hardware register at offset 0.

5. **Sequence of Operations ('s' Command)**:

```
else if (cbuf[0] == 's') {
   for (k = 0; k < 40; k++) {
      if (k < 10) {
         uu = 0;
      } else if (k < 20) {
         uu = 26214;
      } else {
         uu = 26214;
```

```
        } else {
            uu = -26214;
        }
        HW_REG(2) = *(unsigned int *)&uu;
        HW_REG(1) = 0;
        xx = HW_REG(1);

        if (xx == 0) {
            printf(" *** Done bit is 0. (%d)\n\r", xx);
        }

        xx = HW_REG(2);
        yy = *(short int *)&xx;
        yy_d = yy * 3.0517578125e-05;
        printf("%3d uu: %6d yy: %6d (%7.3f)\n\r", k, uu, yy,
yy_d);
    }
}
```

- Loops 40 times, performing different actions based on the value of k.
  - For k from 0 to 9, uu is set to 0.
  - For k from 10 to 19, uu is set to 26214.
  - For k from 20 to 39, uu is set to -26214.
- offset 2, writes uu to the hardware register.
- writes 0 at offset 1 to the hardware register.
- reads the value into xx from the hardware register located at offset 1.
- A message stating that the "Done bit" is 0 is printed if xx is 0.
- reads the value into xx from the hardware register located at offset 2.
- xx is interpreted as the short integer yy.
- factors yy and stores the outcome in yy_d.
- outputs k, uu, yy, and yy_d, the loop indexes.

Cleanup and Exit

1. **Final Register Write**:

```
HW_REG(0) = 0x56;

xx = HW_REG(0);

printf("PushB/Switches: %x\n\r", xx);
```

- Writes the value 0x56 to the hardware register at offset 0.
- Reads back the value and prints it.

10. CONCLUSION:

- In order to communicate with hardware registers, the application reads user commands.
- It allows orders to be sent to the hardware to read (p), write (l), and carry out a predetermined series of activities (s).
- After the user enters "x," the loop keeps running until the application exits and completes any necessary cleanup.