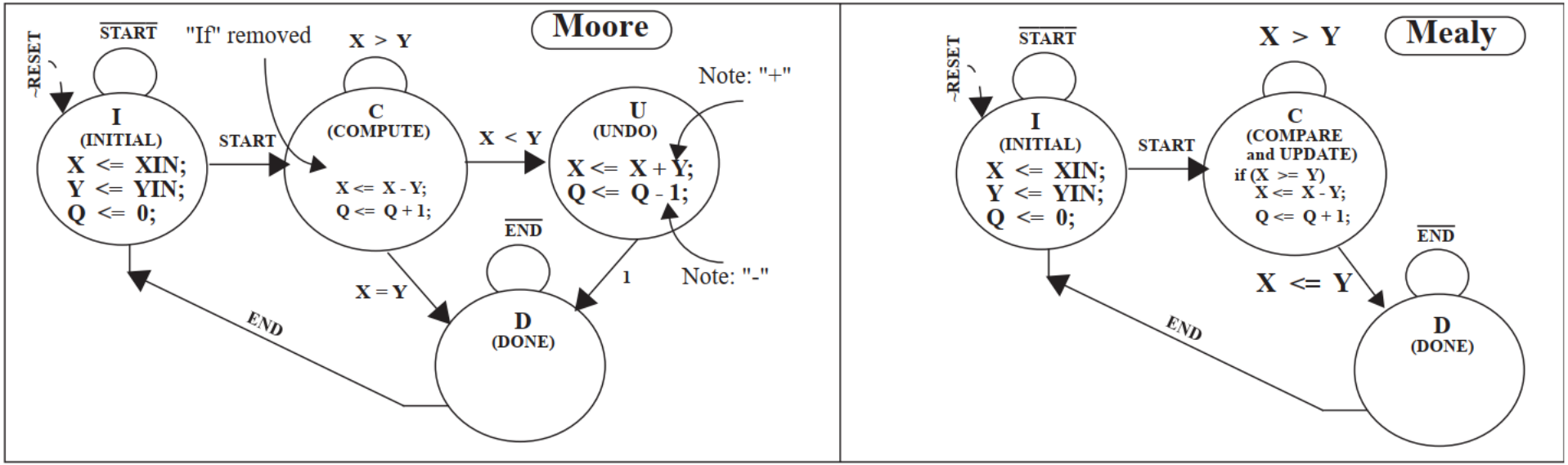
Divider on PicoBlaze

Implementation, Simulation, and Synthesis

Hardware implementation vs. software implementation of a state machine using a divider example

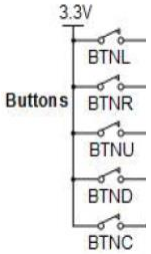


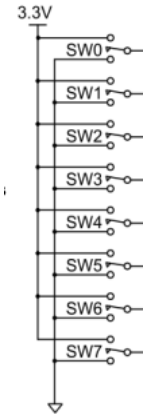
It is always a “Mealy” in software as we do one operation at a time using instructions!

Also all the improvements we thought about for the Mealy machine do not apply to the software implementation as we do one operation at a time in software!

Files and Instances for synthesis

divider_4_top.xdc





FPGA Artix 7 divider_4_top.v module divider_4_top

Input ports related fabric logic

Standard two instances of the picoblaze processor

Instance name processor

picoblaze processor kcpsm6.v

module kcpsm6

address[11:0] →

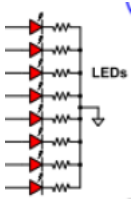
← **instruction[17:0]**

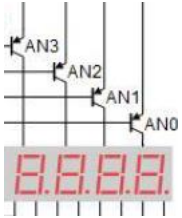
Instance name program_rom

Program memory BRAM prom_divider_4.v produced from prom_divider_4.psm

module prom_divider_4

Output ports related fabric logic and standard SSD scanning logic

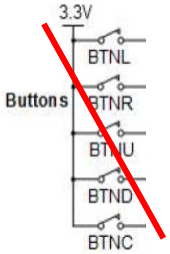


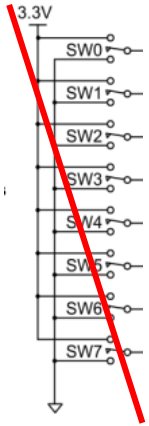


Files and Instances for simulation

~~No xdc for simulation~~

File: divider_4_top_simulation_tb.v **module divider_4_top_simulation_tb**





File: divider_4_top_simulation.v **module divider_4_top_simulation**

Input ports related fabric logic

Standard two instances of the picoblaze processor

Instance name processor

picoblaze processor kcpsm6.v

module kcpsm6

address[11:0] →

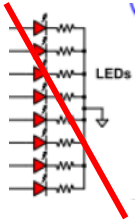
← **instruction[17:0]**

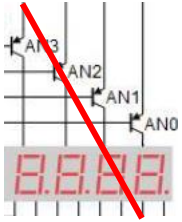
Instance name program_rom

Program memory BRAM prom_divider_4.v produced from prom_divider_4.psm

module prom_divider_4

Output ports related fabric logic and ~~standard SSD scanning logic~~





1. Introduction

- You are given a complete 4-bit divider implemented with PicoBlaze as a reference design
  - The 4-bit divider has a 4-bit Dividend, Divisor, Quotient and a Remainder.
  - The state machine is written in software in the .psm file. You need to read through this state machine completely and see how it differs from a state machine implemented in hardware.
  - The top design contains the fabric logic to interface with the picoblaze, and the normal I/O logic (clock divider, output scanning, etc.) that you are familiar with in top designs (like GCD, etc).
  - A simplified top and a testbench to simulate the simplified top are also given.
- You need to create an 8-bit divider, using the 4-bit divider as a template
  - You will need to make changes to both the .psm and the top design.
  - You are given a new .xdc file for use in the 8-bit divider design.
  - You also need to design a simplified top and a testbench to simulate the simplified top.
- The following slides draw attention to the major changes you will need to make

2.1 Files for the 4-bit divider (all files complete)

Xilinx_projects > Divider_Pico_N4_4bit

Name

- assembly
- sources
- synthesis

Xilinx_projects > Divider_Pico_N4_4bit > synthesis

Name

- synthesis.xpr
- divider_4_top_simulation_tb_behav.wcfg

.wcfg = waveform configuration file

Xilinx_projects > Divider_Pico_N4_4bit > assembly

Name

- prom_divider_4.psm

Xilinx_projects > Divider_Pico_N4_4bit > sources

Name

- divider_4_top.v
- divider_4_top.xdc

For synthesis

- divider_4_top_simulation.v
- divider_4_top_simulation_tb.v

For simulation

- kcpsm6.v
- prom_divider_4.v

For synthesis and simulation

2.2 Files for the 8-bit divider

- The divider_8_top.xdc (Xilinx Design Constraints) file and the divider_8_top_simulation_tb_behav.wcfg (Waveform Configuration) file are given in completed form.

- The .psm file (prom_divider_8.psm) is a mere copy of the prom_divider_4.psm
  You need to revise it as needed and assemble to generate prom_divider_8.v
  A few **** TODO ****** hints were added.

- The three Verilog files, divider_8_top.v, divider_8_top_simulation.v, and divider_8_top_simulation_tb.v are mere copies of the corresponding 4-bit files.
  I have changed the name of the files and also I have revised the module names (and design names of the instances). A few **** TODO ****** hints were added.

- Please watch the video introduction and then read the 4-bit files. Revise the 8-bit files to suit.

Simulation reference pages from the user guide

Pages 45 and 46 from
Picoblaze_KCPSM6_Release9_30Sept14
A reverse assembler was built into the Picoblaze (kcpsm6.v) to facilitate
Dynamic Instruction Execution Trace (DIET or DET)

HDL Simulation Features

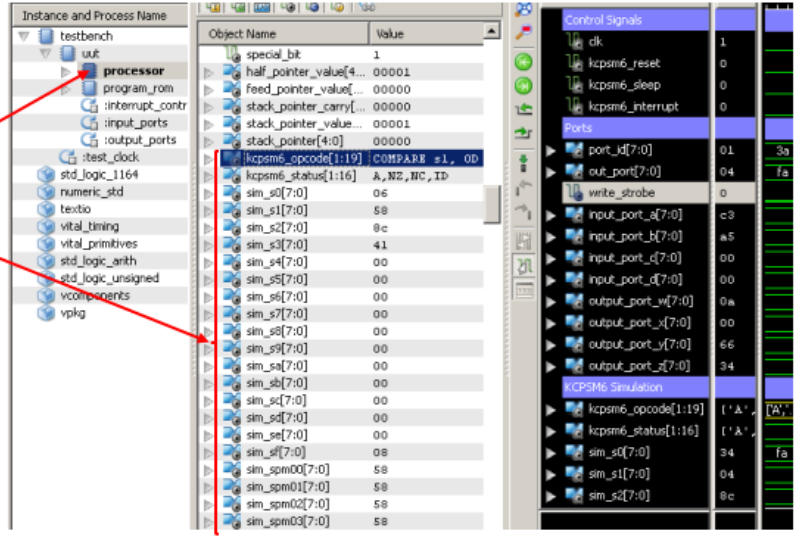
Hint – In most of the cases in which a user reports that KCPSM6 does not simulate at all (e.g. the 'address' does not advance as expected), the cause has been the failure on the part of the user to define valid logic levels for the 'interrupt', 'sleep' and 'reset' controls. So please make sure that all signals are defined at the start of your simulation either in your design or in your simulation test bench.

Since KCPSM6 is a fully embedded part of your hardware design it will simulate along with the rest of your design in an HDL simulator such as iSim or XSim. This means that you can see how KCPSM6 interacts with your design in the same fundamental way in which you might check the operation of a dedicated state machine.

As well as being able to observe any of the input and output signals connecting KCPSM6 to the rest of your design KCPSM6 contains some additional signals specifically for simulation purposes only.

Within the simulator locate the instance of KCPSM6 to be observed. In this case the instance name is 'processor' and the simulator is iSIM (part of ISE).

Then all the internal signals of KCPSM6 can be seen and selected for waveform display as desired. Look down the list and the simulation specific signals can be found.



kcpsm6_opcode – This is a text string displaying the instruction being executed. As well as being easier to understand than the raw codes being read from the program memory they can also be compared with the LOG file from the assembler to directly trace code execution

kcpsm6_status – This is a text string displaying the status…
Active register bank 'A' or 'B'
Zero flag Z or NZ
Carry flag C or NC
Interrupts enabled (IE) or disabled (ID)
Reset or Sleep modes.

e.g. A, Z, NC, IE, Sleep
Bank A, Z=1, C=0, interrupts enabled, in sleep mode

'sim_s0' to 'sim_sf' – The contents of each of the 16 registers in the active register bank (i.e. Contents will reflect bank selection).

'sim_spm00' to 'sim_spmff' – The contents of each of the 256 scratch pad memory locations. Remember that default memory size is 64 bytes (only up to sim_spm3f).

Hint – Adjust the radix of the values displayed.

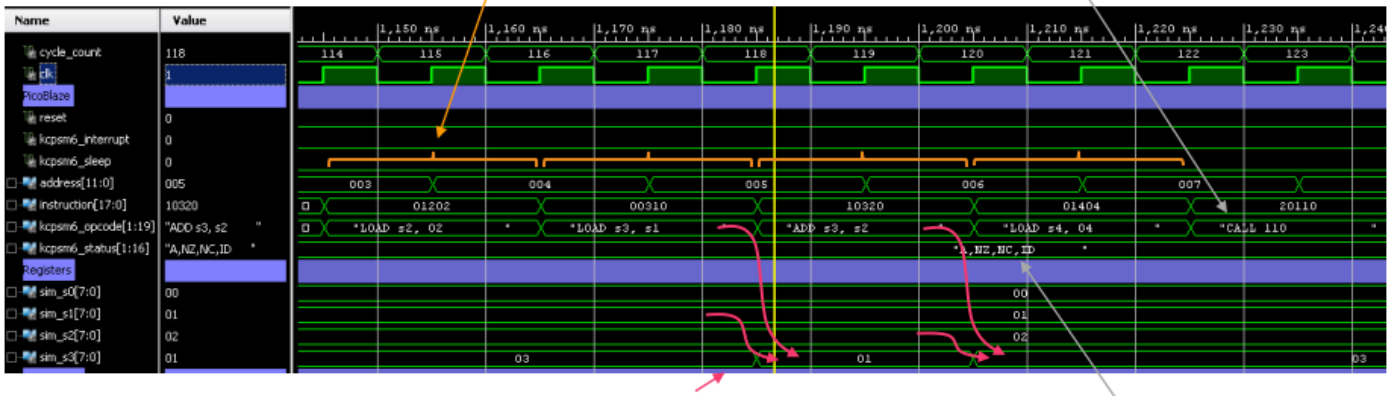
XILINX

HDL Simulation Features

In this iSim or Xsim waveform view the following can be seen…

Each instruction taking 2 clock cycles to execute

Instruction op-codes decoded and displayed as text strings.



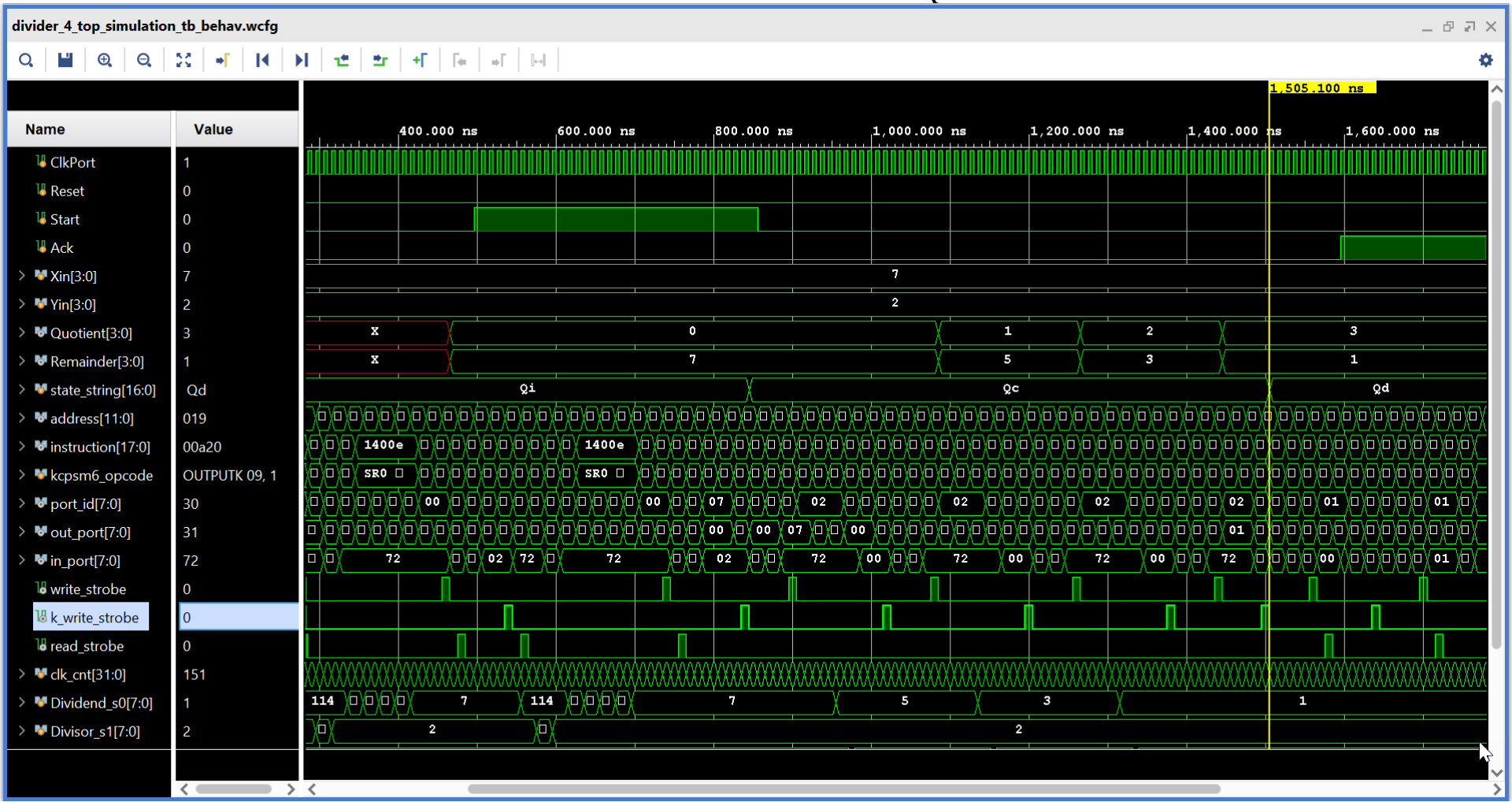
The contents of registers. In this example we can see 's3' being loaded with the contents of 's1' followed by the addition of the contents of 's2'.

Register bank 'A', States of flags and interrupt.

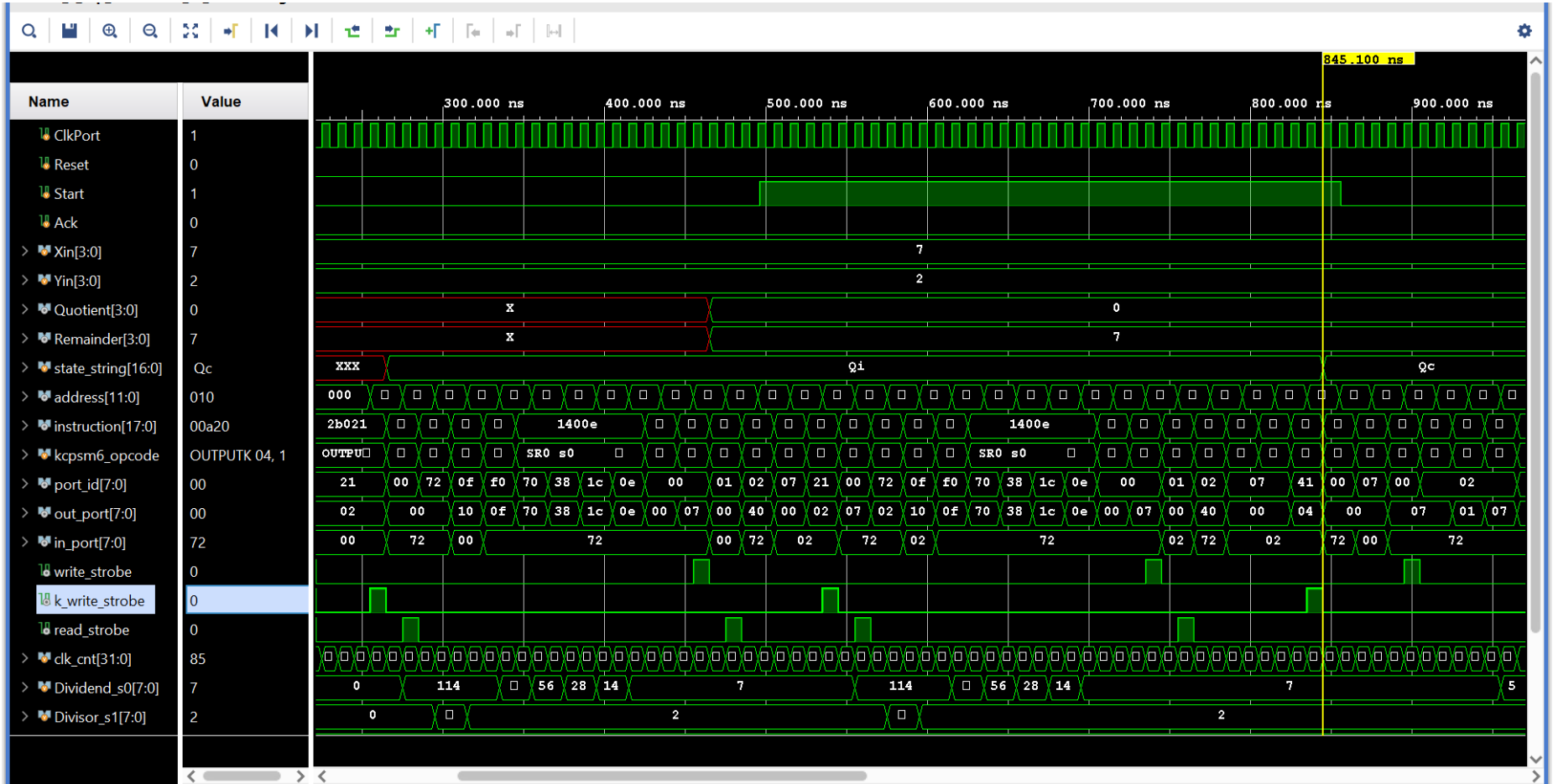
Hint – KCPMS6 programs often contain code that is used to deliberately slow down the progress through the program to service the application correctly either using software delay loops or polling of status signals. For example, when communicating with a UART that has a BAUD rate of 115200 then each character will take 86.8µms to be received and that would equate to 8,680 clock cycles of a 100MHz system clock. Due to this, it is not uncommon for users to become confused by what they perceive as a "lack of activity" in their simulated design simply because KCPSM6 is taking so many clock cycles. So if this is the situation, it may be necessary to alter the PSM code to make the HDL simulation practical but obviously you will need to remember to restore the correct code for the real application. In practice, most PSM code is developed interactively in real-time on the target hardware using JTAG_Loader to facilitate rapid iterations. As such, HDL simulation is best used to confirm your port interfacing logic and generation of particular strobes and waveforms etc.

XILINX

Simulation waveform for the 4-bit divider Dd = 7 Dr = 2 Qt = 3 Rr = 1



2.3 How long the START signal needs to be activated? Safe-side plan: Keep it active until the PSM goes to Qc State!



2.4 Reverse Assembled Instructions facilitate debugging

divider_4_top_simulation_tb_behav.wcfg

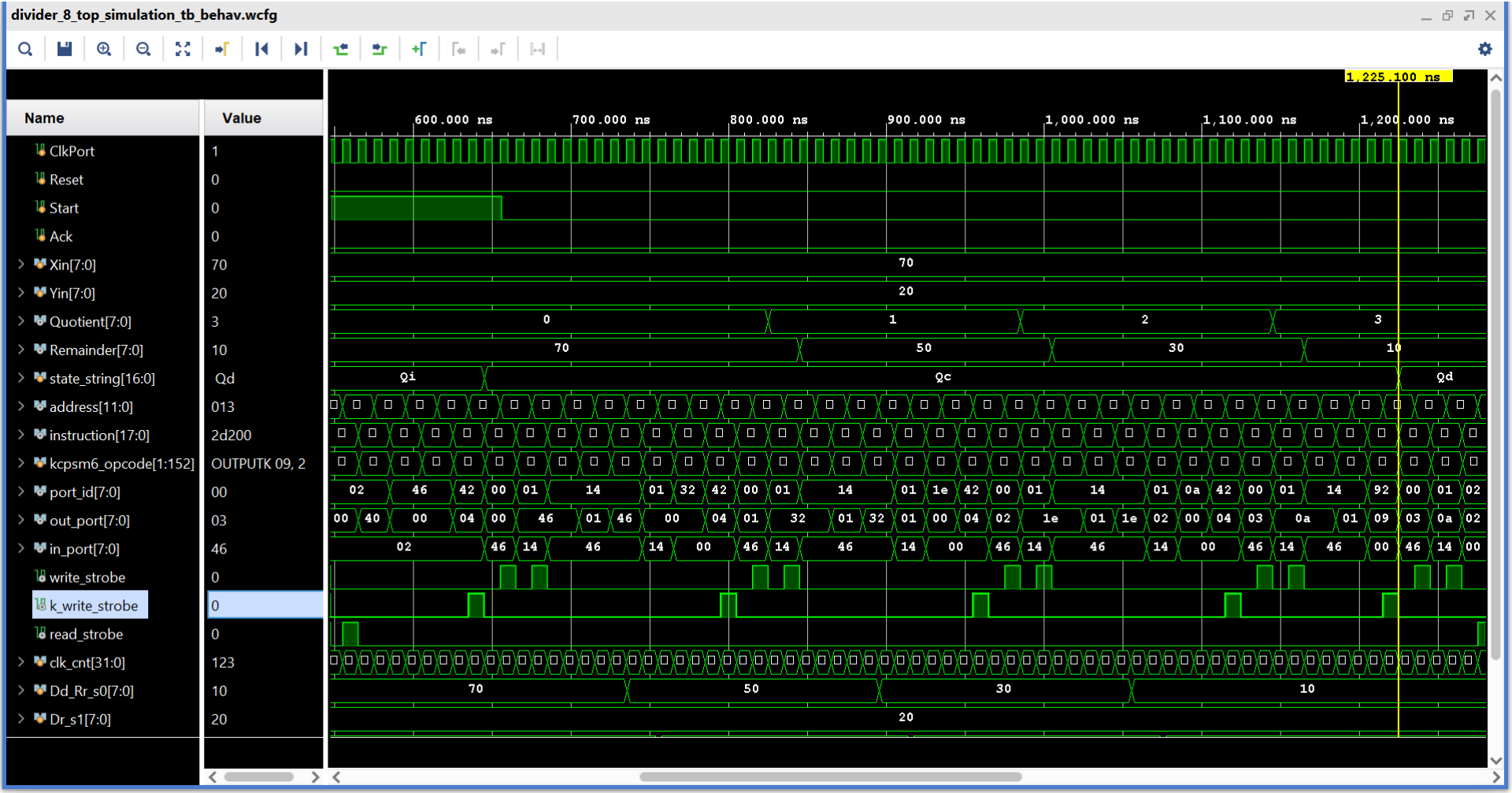
Name	Value
ClkPort	1
Reset	0
Start	0
Ack	0
Xin[3:0]	7
Yin[3:0]	2
Quotient[3:0]	0
Remainder[3:0]	7
state_string[16:0]	Qc
address[11:0]	014
instruction[17:0]	1c010
kcpsm6_opcode	COMPARE s0, s1
port_id[7:0]	02
out_port[7:0]	07
in_port[7:0]	72
write_strobe	0
k_write_strobe	0
read_strobe	0
clk_cnt[31:0]	93
Dividend_s0[7:0]	7
Divisor_s1[7:0]	2

925.000 ns

900.000 ns 920.000 ns 940.000 ns 960.000 ns 980.000 ns 1,000.000 ns 1,020.000 ns 1,040.0

7															
2															
0															
7															
Qc															
013	014	015	016	017	00f	010	011								
2da00	1c010	3a018	18010	11210	2200f	2b041	00a20								
OUTP	COMPARE s0, s1	JUMP C, 018	SUB s0, s1	ADD s2, 10	JUMP 00F	OUTPUTK 04, 1	LOAD sA, s2								
00	02	10	05	41	10										
07	01	07	00	04	07										
72	00	72													
90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	
7	5														
2															

2.5 Simulation waveform for the 8-bit divider Dd = 70 Dr = 20 Qt = 3 Rr = 10

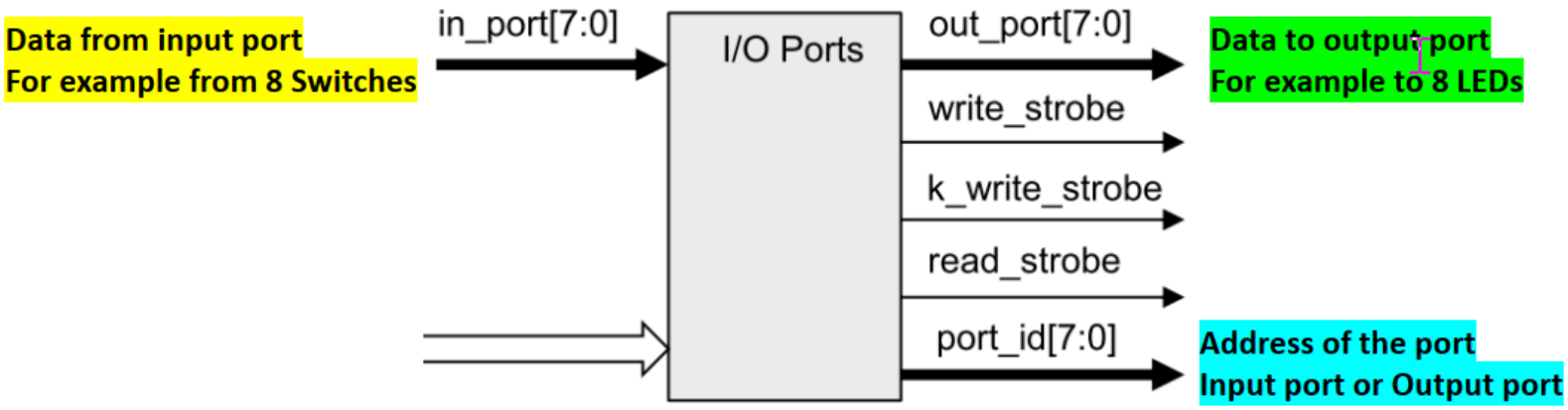


Input and Ouput ports of Picoblaze

Interface with the Fabric Logic via
Input ports and Output ports

Extracts from
Picoblaze_KCPSM6_Release9_30Sept14

Interface with the Fabric Logic via Input and Output ports

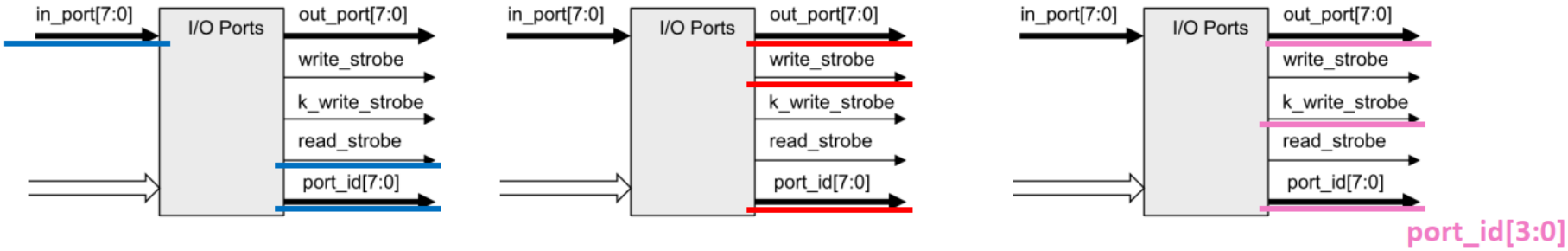


Using the 8-bit port_id[7:0], we can generate 256 port addresses and talk to them (exchange data with them).

So, all together do we have 256 I/O ports or do we have 256 input ports and 256 output ports?

Picoblaze pins associated with Input and Output ports

Interface with the Fabric Logic via Input and Output ports



256 Input ports (Read only)

8-bit port_id qualified by read_strobe produces 256 IDSPs (Input Device Select Pulses)

256 output ports (Write only)

8-bit port_id qualified by write_strobe produces 256 ODSPs (Output Device Select Pulses)

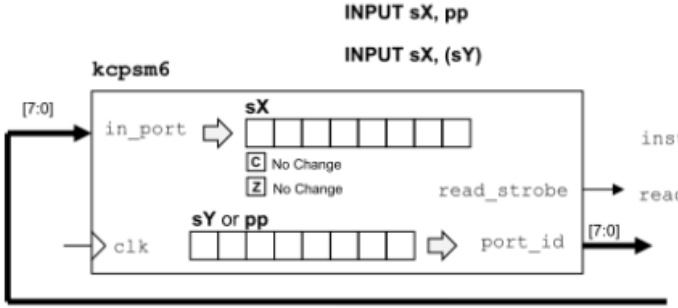
16 output ports (Write only)

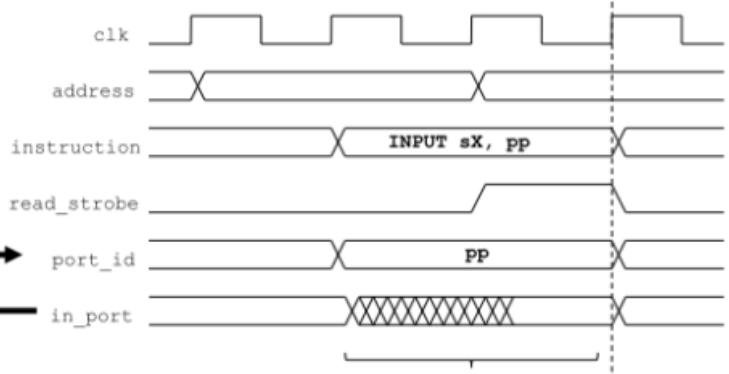
4-bit port_id qualified by k_write_strobe produces 16 k_ODSPs (k_Output Device Select Pulses)

INPUT sX, pp
INPUT sX, (sY)

IMPORTANT

An 'INPUT' instruction enables KCPSM6 to read information ~~from the~~ from your hardware design into a register 'sX' using a general purpose input port specified by an 8-bit constant value 'pp' or the contents of another register '(sY)'. KCPSM6 presents the port address defined by 'pp' or '(sY)' on 'port_id' and your hardware interface is then responsible for selecting and presenting the appropriate information to the 'in_port' so that it can be captured into the 'sX' register. An active High ('1') synchronous pulse is also generated on the 'read_strobe' pin and may be used by the hardware interface to confirm when a particular port has been read.





There are 2 clock cycles available to decode the port address 'pp' or '(sY)' and present the requested information to the 'in_port'.

Data captured into 'sX' on this rising clock edge.

Hint 1 – Assign your input port addresses such that the data selection multiplexer feeding 'in_port' uses the minimum number of 'port_id' signals to make the selection, e.g. port addresses '00' to '0F' provide 16 input ports and only require 'port_id(3:0)' to be selection inputs to the multiplexer resulting in smaller faster designs.

Hint 2 – Unless there is a specific reason not to, the input data selection multiplexer should include a pipeline register (i.e. your case statement should be within a clocked process). In this way the data is selected during the first clock cycle of 'port_id' and presented to 'in_port' during the second clock cycle. Failure to define a pipeline register anywhere in the 'port_id' to 'in_port' path is the most common reason for PicoBlaze designs failing to meet the required performance (a 'false path' for one clock cycle) .

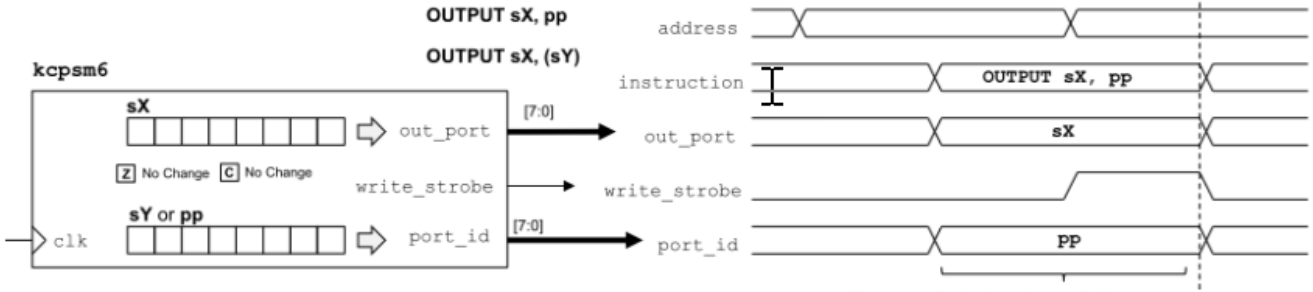
Hint 3 – 'read_strobe' can be ignored in most cases and never needs to be part of the multiplexer feeding 'in_port'. However, some functions such as a FIFO buffer do need to know when they have been read and it is in those situations that 'read_strobe' together with a decode of the appropriate value of 'port_id' would be used to generate a "port has been read" pulse to confirm when a read has taken place.

XILINX

OUTPUT sX, pp
OUTPUT sX, (sY)

IMPORTANT

An 'OUTPUT' instruction is used to transfer information from a register 'sX' to a general purpose output port specified by an 8-bit constant value 'pp' or the contents of another register '(sY)'. KCPSM6 presents the contents of the register 'sX' on 'out_port' and the port address defined by 'pp' or '(sY)' is presented on 'port_id'. Both pieces of information are qualified by an active High ('1') synchronous pulse on the 'write_strobe' pin. Your hardware interface is responsible for capturing the information presented.



There are 2 clock cycle available to decode the port address 'pp' or '(sY)'

The value presented on 'out_port' should be captured on the rising edge of the clock when 'write_strobe' is High.

Note that 'out_port' and 'port_id' will vary during the execution of other instructions but 'write_strobe' will only be active during an OUTPUT instruction.

Hint – In most cases a fixed port address 'pp' is used so CONSTANT directives provide an ideal why track your port assignments and make your code easier to write, understand and maintain.

way to

Examples

```
CONSTANT LED_port, 05
;
LOAD s3, 3A
OUTPUT s3, LED_port
```

If you want to keep your designs small and fast then assign port addresses that facilitate smaller logic functions.

In this example a set of 8 LEDs are mapped to port 05 hex and only 3-bits of 'port_id' together with 'write_strobe' are decoded.

VHDL

```
if clk'event and clk = '1' then
  if write_strobe = '1' then
    if port_id(2 downto 0) = "101" then
      led <= out_port;
    end if;
  end if;
end if;
```

```
OUTPUT s6, (s2)
OUTPUT s4, 40
OUTPUT sB, 64'd
```

Decimal values can be used to specify port addresses but hex or binary values are normally easier to work with when defining the hardware.

XILINX

Input Ports

IMPORTANT

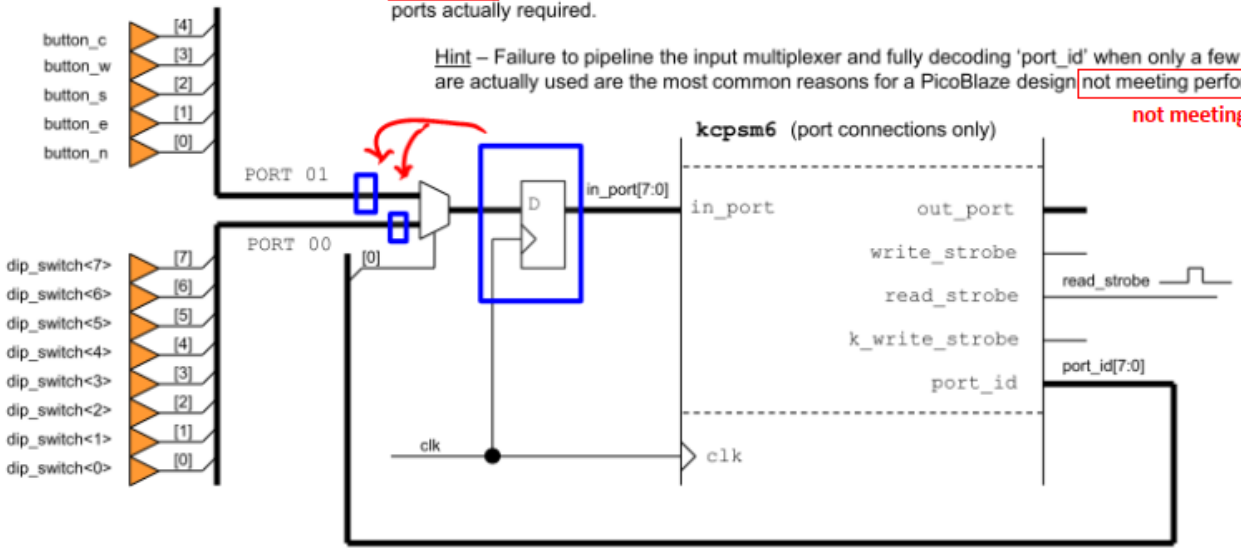
KCPSM6 can read 8-bit values from up to 256 general purpose input ports using its 'INPUT sX, pp' and 'INPUT sX, (sY)' instructions. A complete description is provided in the reference section later in this document but here we can see this put into practice so that KCPSM6 can read the 8 DIP switches and 5 'direction' push buttons on the ML605 board.

Please see next page for the corresponding VHDL and PSM

When KCPSM6 executes an 'INPUT' instruction it sets 'port_id' to specify which of 256 ports it wants to read from and it is the task of the hardware circuit to present that 8-bit data to the 'in_port'. A simple multiplexer can be used to achieve this. It is best practice to pipeline the output of the multiplexer and to only use the appropriate number of bits of 'port_id' to facilitate the number of input ports actually required.

Hint – Failure to pipeline the input multiplexer and fully decoding 'port_id' when only a few input ports are actually used are the most common reasons for a PicoBlaze design not meeting performance.

not meeting clock period.



In a non-FPGA based design such as an ASIC or a custom VLSI, it is common practice to use tristate buffers in place of a mux. You want to enable a tri-state buffer only after confirming that the address is stabilized and is not changing. This is to make sure that multiple tristate buffers are not enabled unintentionally for a short length of time during address transitions. Hence, I have moved the pipeline register to the upstream of the mux. In my design, there is a register for every input port, and it captures data on every clock. The data read by the processor is one clock delayed and in most cases that is not harmful (particularly in our application with manual operation of switches and buttons.

Hint - The 'read_strobe' is also pulsed High when KCPSM6 executes an 'INPUT' instruction but this does not need to be used to qualify the multiplexer selection. This strobe would be used in situations where the circuit being read needs to know when data has been captured. The most obvious example is reading data from a FIFO so that it can discard the oldest information and present the information to be read on its output.

XILINX

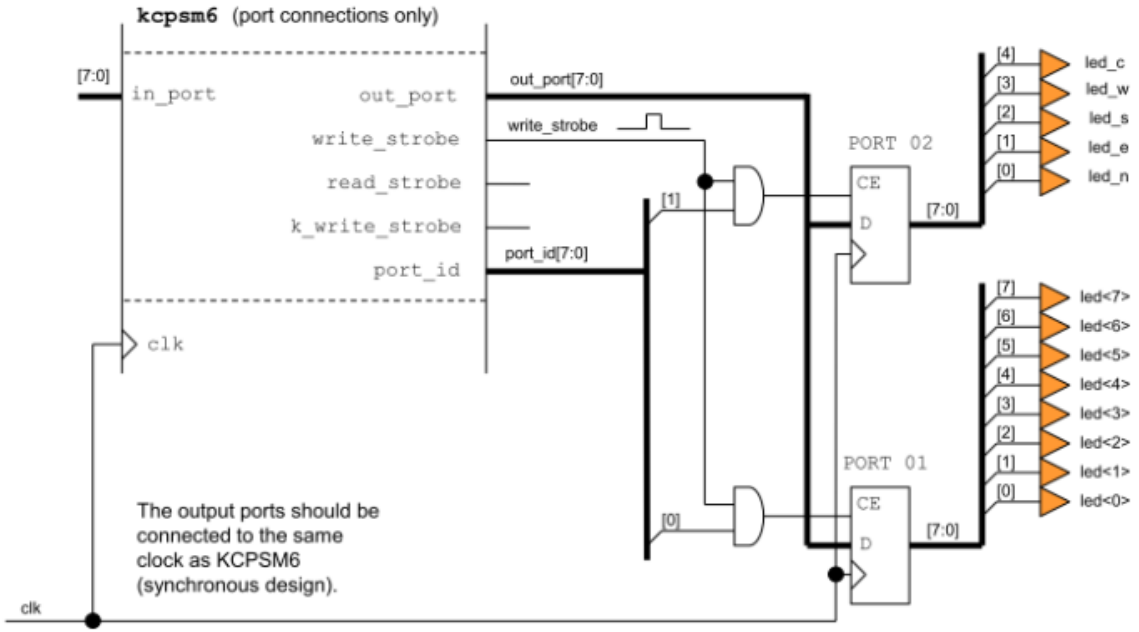
Output Ports

IMPORTANT

KCPSM6 can output 8-bit values to up to 256 general purpose output ports using its 'OUTPUT sX, pp' and 'OUTPUT sX, (sY)' instructions. A complete description is provided in the reference section later in this document but here we can see this put into practice so that KCPSM6 can control the 8 general purpose LEDs and 5 'direction' LEDs on the ML605 board.

Please see next page for the corresponding VHDL and PSM

When KCPSM6 executes an 'OUTPUT' instruction it sets 'port_id' to specify which of 256 ports it wants to write the 8-bit data value present on 'out_port'. A single clock cycle enable pulse is generated on 'write_strobe' and **your hardware must use 'write_strobe' to qualify the decodes of 'port_id'** to ensure that only the correct register (or peripheral) captures the 'out_port' value.



The output ports should be connected to the same clock as KCPSM6 (synchronous design).

XILINX

In this lab, the processor informs the fabric logic "in which state it is currently at (Qi or Qc or Qd)" using the OUTPUTK instruction.

Possible to do so in two ways:
using OUTPUT sX, pp or using OUPUTK kk, p

Suppose constant 02 (kk = 02) needs to conveyed to the output port 01 (pp = 01)

```
LOAD s5, 02 ;
OUTPUT s5, 01
```

```
OUTPUTK 02, 01
```

Extracts from prom_divider_4.psm illustrating the use of the OUTPUTK instruction

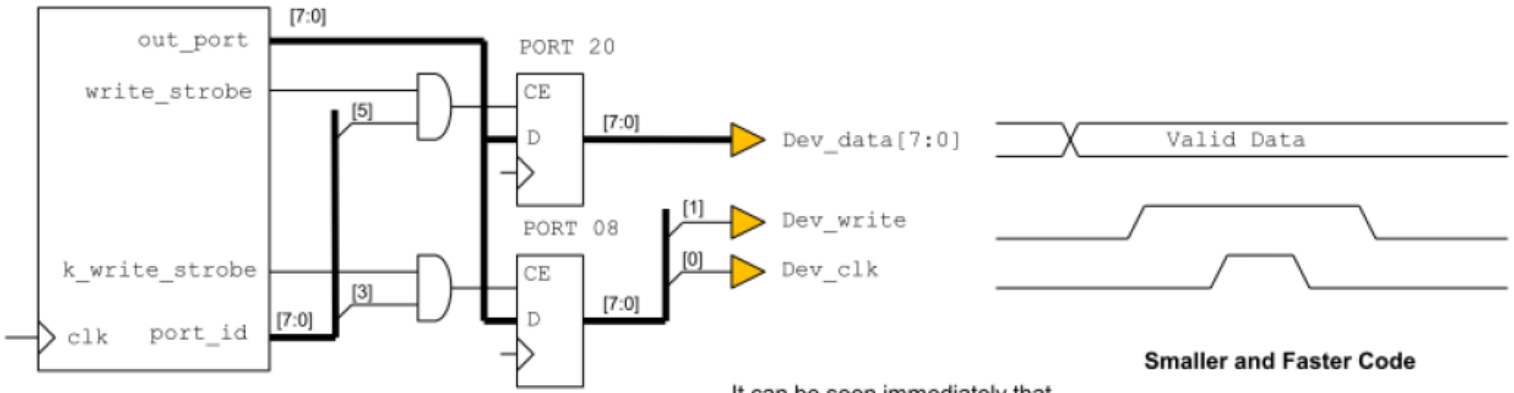
```
11                  CONSTANT Current_State_port,      01'd     ;port01 used for outputting current state info
12
13  ; Current State format from divider_4_top.v
14  ;           Done <= out_port[0];
15  ;           Qi <= out_port[1];
16  ;           Qc <= out_port[2];
17  ;           Qd <= out_port[3];
18
19                  CONSTANT Report_Qi,          00000010'b
20                  CONSTANT Report_Qc,          00000100'b
21                  CONSTANT Report_Qd_Done,     00001001'b
22
23  ; Control signal format from divider_4_top.v
24  ;           1'b1 : in_port <= {6'b000000,Start,Ack};
25
26                  CONSTANT Mask_to_check_Start,    00000010'b
27                  CONSTANT Mask_to_check_Ack,      00000001'b
28
```

```
44    state_initial: OUTPUTK Report_Qi, Current_State_port    ; Indicating Current State as Initial State (QI)

63    state_compute: OUTPUTK Report_Qc, Current_State_port    ; Indicating Current State as Compute State (QC)

76    state_done:    OUTPUTK Report_Qd_Done, Current_State_port   ; Indicating Current State as Done State and also
```

Constant-Optimised Output Ports

Returning to the same example of writing data to an external device we can see that port 08 hex has now been allocated to a constant-optimised output port by using the 'k_write_strobe' whilst port 20 hex is still associated with 'write_strobe' because the data is naturally variable. So there is very little difference in the hardware as long as you remember that only port_id[3:0] are defined during an OUTPUTK instruction. Note also that you could now have two different output ports with the same address; on for variable data and the other for constant values (see page 79).

Using a Constant-Optimised Output Port and a General Purpose Output Port……




```
CONSTANT Dev_data_port, 20
CONSTANT Dev_control_port, 08
```

It can be seen immediately that all the LOAD instructions have been eliminated saving code space and reducing the execution time. This also means that register 's0' used previously to define the sequence of values is now free for another purpose.

Smaller and Faster Code

```
OUTPUT s1, Dev_data_port
OUTPUTK 00000010'b, Dev_control_port
OUTPUTK 00000011'b, Dev_control_port
OUTPUTK 00000010'b, Dev_control_port
OUTPUTK 00000000'b, Dev_control_port
```


XILINX

3. Revising files to suit the 8-bit divider

3.1 Changes to the .psm file

- Port definitions in the 4-bit file prom_divider_4.psm

```
CONSTANT Dividend_Divisor_port,   00'd     ;port00 used for loading info of Dividend and Divisor
CONSTANT Control_signal_port,     01'd     ;port01 used for loading info of Start and ACK signals
CONSTANT Quotient_Remainder_port, 00'd     ;port00 used for outputting Quotient and Remainder
CONSTANT Current_State_port,      01'd     ;port01 used for outputting current state info (Done (QD),
```

The picoblaze processor interfaces with our fabric logic through 8-bit input ports and 8-bit output ports. In the 4-bit divider we have two input ports and two output ports. Let's consider the input ports first: port_id 00 is used for the 4-bit Dividend and 4-bit Divisor concatenated together, and port_id 01 is used for the Start/Ack signals.
How many input ports do we need now that Dividend and Divisor are each 8-bit?

We also have two output ports: port_id 00 is used for the 4-bit Quotient and 4-bit Remainder together, and port_id 01 to display the current state information.
Now that our Quotient and Remainder are each 8-bit, how many output ports do we need?

.psm programmable state machine file design

```
state_initial: OUTPUTK Report_Qi, Current_State_port
               INPUT s0, Dividend_Divisor_port
               LOAD s1,s0
               AND s1,0F
               AND s0,F0
               SR0 s0
               SR0 s0
               SR0 s0
               SR0 s0
               LOAD s2,00
               OUTPUT s0, Quotient_Remainder_port
               INPUT s4, Control_signal_port
               AND s4, Mask_to_check_Start
               JUMP Z, state_initial
               JUMP state_compute
```

In the initial state, we need to load our registers with the Dividend and Divisor. In the 4-bit divider, the Dividend and Divisor are read together as one 8-bit value, so we have to do some masking and shifting operations to extract each individual element. In your 8-bit divider, you are reading the Dividend and Divisor in on seperate input registers.

So your initial state will be much simpler in the 8-bit divider!

- Compute State

```
state_compute: OUTPUTK Report_Qc, Current_State_port
               LOAD sA,s2
               OR  sA, s0
               OUTPUT sA, Quotient_Remainder_port
               COMPARE s0, s1
               JUMP C, state_done
               SUB s0, s1
               ADD s2, 10
               JUMP state_compute
```

In the 4-bit divider, we have a single 8-bit register s2 holding the quotient in the upper 4-bits. That’s why we add 10 hex (0001_0000 in binary) to s2 to increment the Quotient, because the Quotient is the upper 4-bits of this register. We used sA to merge Quotient and the Remainder for outputting Quotient-Remainder pair to the Top design. Make the necessary changes now that we have a full 8-bit register for each (the Quotient and the Remainder).

- Done State

```
state_done:     OUTPUTK Report_Qd_Done, Current_State_port
                LOAD sA,s2                                ;
                OR  sA, s0                                ;
                OUTPUT sA, Quotient_Remainder_port        ;
                INPUT s4, Control_signal_port             ;
                AND s4, Mask_to_check_Ack                 ;
                JUMP Z, state_done                        ;
                JUMP state_initial                        ;
```

Consider how having the Quotient and Remainder on separate 8-bit registers will change your instructions in the Done state

3.2 Changes to the top design
divider_4_top.v ➔ divider_8_top.v

- Port list
  - In the 8-bit divider we need to use all 16 switches, instead of just 8 switches in the 4-bit divider
  - We need to use 8 SSDs instead of 4 SSDs

- Variable declaration sizes
  - Consider changes that need to be made to the sizes of Xin, Yin, Quotient, Remainder now that we are doing 8-bit division

Top design - fabric logic to interface with the PicoBlaze Processor

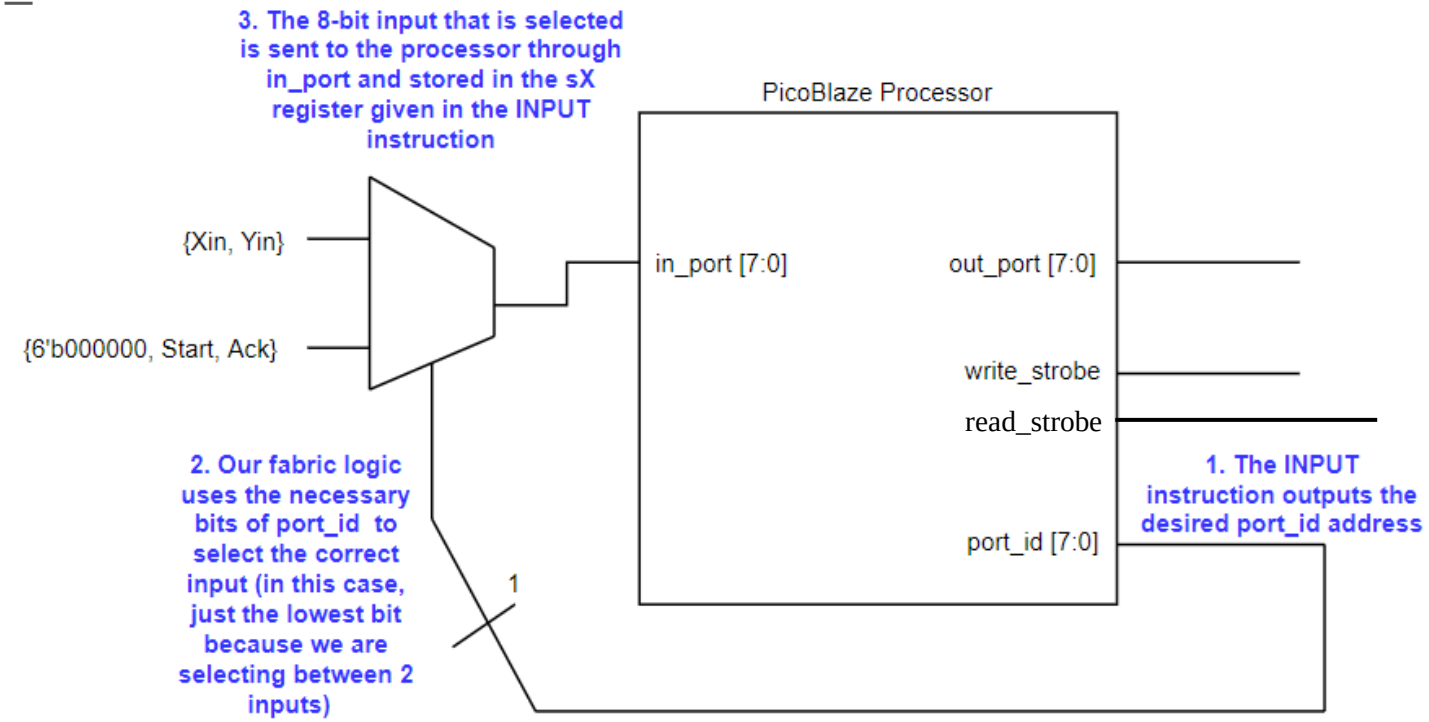
- How is our top design interfacing with the picoblaze processor through the INPUT and OUTPUT instructions? Let’s consider the INPUT instruction.

  INPUT sX, pp

  “pp” is the 8-bit port address in hex that the processor will output on port_id

  sX is one of the 16 8-bit registers (s0 through sf) inside the processor that will store the data coming through in_port[7:0]

Our fabric logic needs to present all the possible inputs to the in_port of the processor, and allow the port_id to select which one is desired by the processor. Since the in_port pins are dedicated input pins of the processor, it is OK to drive them without checking the read_strobe here.



in_port interface in the top design

- The mux shown in the previous slide is implemented in the 4-bit divider top design with this always block:

```
always @ (*)
begin
    case (port_id[0])
        1'b0 : in_port <= {Xin,Yin};
        1'b1 : in_port <= {6'b000000,Start,Ack};
        default : in_port <= 8'bXXXXXXXX ;
    endcase
end
```

- You need to expand this mux to account for 3 input ports in your 8-bit design. How many bits of port_id will you use? How many 8-bit inputs do you need for the mux?

out_port interface

- We have two output instructions available to us, OUTPUT and OUTPUTK

  The regular OUTPUT instruction lets us output the 8-bit data of a register:

  OUTPUT sX, pp

  sX is the register containing the data we are outputting, pp is the 8-bit address we output on port_id, write_strobe signal goes active.

  The OUTPUTK instruction lets us output an 8-bit constant:

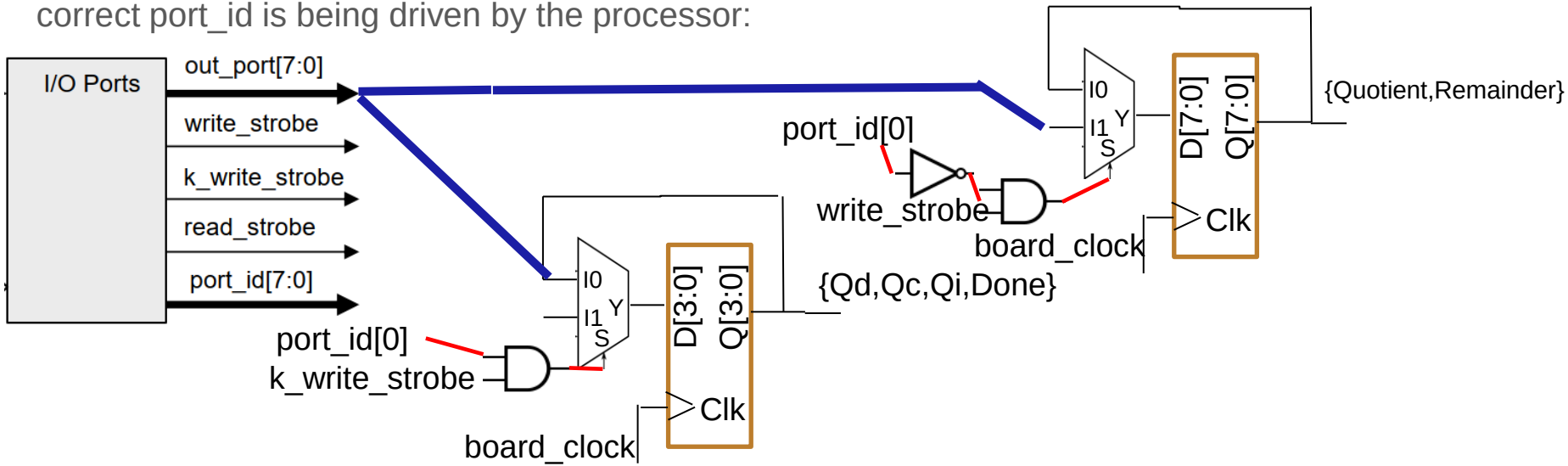
  OUTPUTK kk, p

  “kk” is the 8-bit constant we are outputting, p is the 4-bit address we output on port_id , k_write_strobe signal goes active.

out_port interface

With an output instruction, the data on out_port is only available during the execution of the instruction. We need to register all output data because we need to use it for longer than the time it is available. The picoblaze processor provides us with a write_strobe and k_write_strobe signal to help us with this. These signals go high during the second clock of an output instruction to signal to our fabric logic that valid information is on out_port

We should only write to our output registers if 1) write_strobe or k_write_strobe is true and 2) if the correct port_id is being driven by the processor:



out_port Interface in the top design divider_4_top.v

- The fabric logic for the outputs in the previous slide is implemented in the 4-bit divider top design with the following clocked always block:

```
always @(posedge board_clk)
begin
    // 'write_strobe' is used to qualify all writes to general output ports using OUTPUT.
    if (write_strobe == 1'b1)
    begin
        if(port_id[0] == 1'b0)begin
            {Quotient,Remainder} <= out_port;
        end
    end

    // 'k_write_strobe' is used to qualify all writes to general output ports using OUTPUTK.
    if (k_write_strobe == 1'b1)
    begin
        // Write to output_port at port address 01
        if (port_id[0]  == 1'b1)
        begin
            Done <= out_port[0];
            Qi <= out_port[1];
            Qc <= out_port[2];
            Qd <= out_port[3];
        end
    end
end
```

out_port Interface in the top design divider_8_top.v

- You need to expand this in your 8-bit design, as the Quotient and Remainder are now each 8-bit and each have their own port_id

SSD Display

- In the 4-bit design, we only use 4 SSD’s to display the 4-bit Dividend, Divisor, Quotient and Remainder
- In your 8-bit design, you will need all 8 SSD’s, using 2 SSD’s for each of the 4 values we need to display
- How does this affect our scanning mechanism?
  - Instead of using 2 bits of Div_clk for the scan clock, we now need 3 bits.
  - The 2-to-4 decoder used to drive the 4 anodes needs to be expanded to a 3-to-8 decoder to drive all 8 anodes
  - The 4-bit wide 4-to-1 mux used to select the SSD information needs to be expanded to 4-bit wide 8-to-1 mux.
- Refer to the test_nexys4_verilog design that you went through before

3.3 Design discussion related to `-- divider_4_top_simulation.v` `-- divider_4_top_simulation_tb.v`

- Substantial discussion is provided in the files themselves.
- Issues related to the length of the START and ACK pulses
- Waiting for the picoblaze processor to receive the stimulus and waiting for the picoblaze to report results to us.
- In the introduction video we discussed these aspects at length.

4.1 Simulation output results file extract

divider_4_top.v | prom_divider_4.v | divider_4_top_simulation.v | divider_4_top_simulation_tb.v | instruction_trace_divider_4.txt | results_divider_4.txt

```
Reporting from the Qd Done state
Xin: 7 Yin: 2 Quotient: 3 Remainder: 1
It took          73 clocks to compute the quotient
Reporting from the Qd Done state
Xin:15 Yin: 3 Quotient: 5 Remainder: 0
It took         109 clocks to compute the quotient
Reporting from the Qd Done state
Xin:14 Yin: 4 Quotient: 3 Remainder: 2
It took          73 clocks to compute the quotient
```

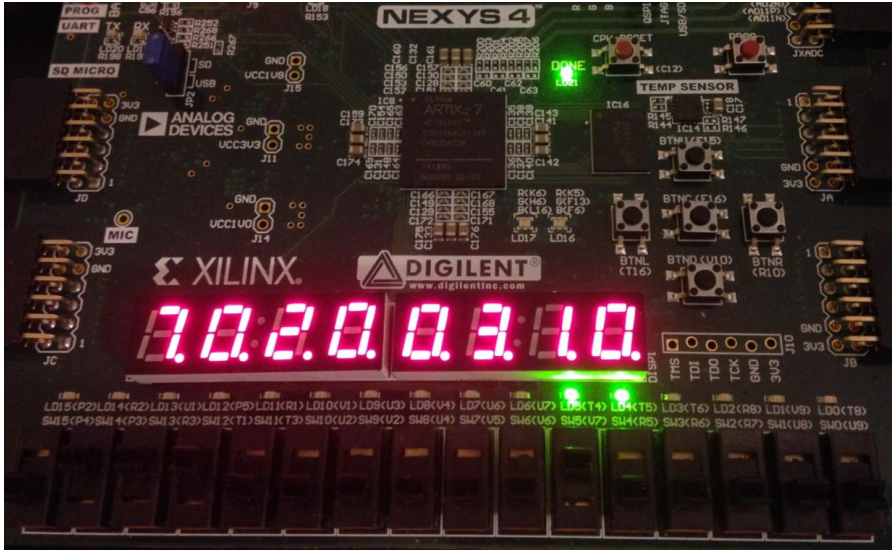
4.2 Simulation DIET extract
DIET = Dynamic Instruction Execution Trace

divider_8_top.v | prom_divider_4.psm | prom_divider_8.psm | divider_4_top_simulation_tb.v | divider_8_top_simulation_tb.v | divider_8_top_simulation.v | results_divider_8.txt | instruction_trace_divider_8.txt

```
Addr: 000  Instr: 2b022 OUTPUTK 02, 2      Dividend_Remainder_s0: 00 Divider_s1: 00 Quotient_s2: 00  Control_s4: 00  cc: 26
Addr: 001  Instr: 09000 INPUT s0, 00       Dividend_Remainder_s0: 46 Divider_s1: 00 Quotient_s2: 00  Control_s4: 00  cc: 28
Addr: 002  Instr: 09101 INPUT s1, 01       Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 30
Addr: 003  Instr: 01200 LOAD s2, 00        Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 32
Addr: 004  Instr: 2d200 OUTPUT s2, 00      Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 34
Addr: 005  Instr: 2d001 OUTPUT s0, 01      Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 36
Addr: 006  Instr: 09402 INPUT s4, 02       Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 38
Addr: 007  Instr: 03402 AND s4, 02         Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 40
Addr: 008  Instr: 32000 JUMP Z, 000        Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 42
Addr: 000  Instr: 2b022 OUTPUTK 02, 2      Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 44
Addr: 001  Instr: 09000 INPUT s0, 00       Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 46
Addr: 002  Instr: 09101 INPUT s1, 01       Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 48
Addr: 003  Instr: 01200 LOAD s2, 00        Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 50
Addr: 004  Instr: 2d200 OUTPUT s2, 00      Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 52
Addr: 005  Instr: 2d001 OUTPUT s0, 01      Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 00  cc: 54
Addr: 006  Instr: 09402 INPUT s4, 02       Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 02  cc: 56
Addr: 007  Instr: 03402 AND s4, 02         Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 02  cc: 58
Addr: 008  Instr: 32000 JUMP Z, 000        Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 02  cc: 60
Addr: 009  Instr: 2200a JUMP 00A           Dividend_Remainder_s0: 46 Divider_s1: 14 Quotient_s2: 00  Control_s4: 02  cc: 62
```

5.1 Demonstrate to your TA/Mentor

Demonstrate working of your 8-bit divider in simulation and on the FPGA board to your TA/Mentor and show your project directory Divider_Pico_N4_8bit to him/her.



Submit files on Unix as per the following posting on the Bb on the next page

5.2 Blackboard posting and Files for submission

Given a completed Picoblaze-based 4-bit divider, design, implement, and simulate a Picoblaze-based 8-bit divider
Directory: http://ee-classes.usc.edu/ee254/ee254l_lab_manual/PicoBlaze/Divider_Pico_N4

Assignment pdf: PicoBlaze_Divider_handout.pdf
Videos (more may be added) 1. Divider_Pico_N4_4bit_xsim_operation.mp4

Two .zip files to be downloaded and extracted into C:\Xilinx_projects:
A completed 4-bit divider design: Divider_Pico_N4_4bit.zip
An incomplete 8-bit divider design: Divider_Pico_N4_8bit.zip
Both designs contain TA's completed .bit files (with dot points glowing on SSDs).
The incomplete 8bit zip file also contains a completed .xdc file and a completed .wcfg file.
General reference: PicoBlaze/Picoblaze_Design_Steps_Demo_README_r1.pdf

Please demonstrate your completed 8-bit design to your TA. Show your simulation waveform and show your FPGA board running the 8-bit divider.
Also submit your files to the class Unix account ee201@viterbi-scf1.usc.edu or ee201@viterbi-scf2.usc.edu using the following submit command

submit -user ee201 -tag Divider_Pico_N4_8bit prom_divider_8.psm divider_8_top.v divider_8_top_simulation.v divider_8_top_simulation_tb.v instruction_trace_divider_8.txt results_divider_8.txt names.txt

The last two text (.txt) files can be found in the following project subdirectory after you finish simulation.
C:\Xilinx_projects\Divider_Pico_N4_8bit\synthesis\synthesis.sim\sim_1\behav\xsim
Please exit simulation. Then only the results_divider_8.txt file gets populated. Until then it remains empty.