ICAR Laboratory : PicoBlaze Lab 5

The aim of this lab is to demonstrate the different processing architectures defined by Flynn’s taxonomy. These classifications describes a processor in terms of how it handles its instructions and data elements. The simplest of these architectures is SISD i.e. a processor that decodes and executes one instruction at a time e.g. the PicoBlaze from the previous labs. Through the addition of extra hardware units this core processing architecture can be expanded, allowing more instructions and data elements to be processed in parallel, therefore, hopefully increasing processing performance.

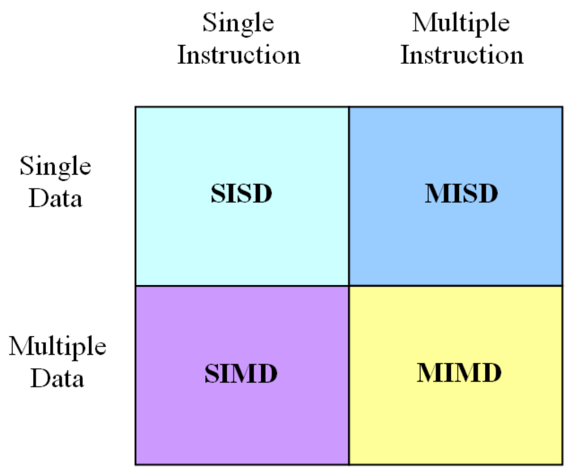


Figure 1: Processing architecture classifications

Processor Architectures and Performance

To assess the processing performance of different processor architectures we shall use the algorithm shown in figure 2. This flowchart performs a vector addition, adding together 16 pairs of data values stored in external memory.

Note: in this first example data pairs are store in sequential memory locations.

An assembly language program to implement this functionality has already been created and can be downloaded from the module's VLE page: `vec_SISD.psm` (link under lab script). Using your preferred web browser, download the file and open it in FIDEx. In this program the address of the memory location from which data is read i.e. `SrcPtr`, is stored in register `S0` and the address of the memory location to which data is written i.e. `DesPtr`, is stored in register `S1`.

Task 1

In the FIDEx create a memory block module for the RAM, as shown in figure 4. Then initialise the memory locations 0 – 31 (0x00 – 0x1F) with the data values 1 – 32 as shown in figure 3. Single step through this program. Confirm the results stored in memory locations 32 – 47 (0x20 – 0x2F) are correct.

Hint: by default memory location contents are displayed in hexadecimal. When entering the data values shown in figure 3 you may wish to switch to decimal mode by left clicking on the DEC icon. To save time you can just enter the values 1 – 10.

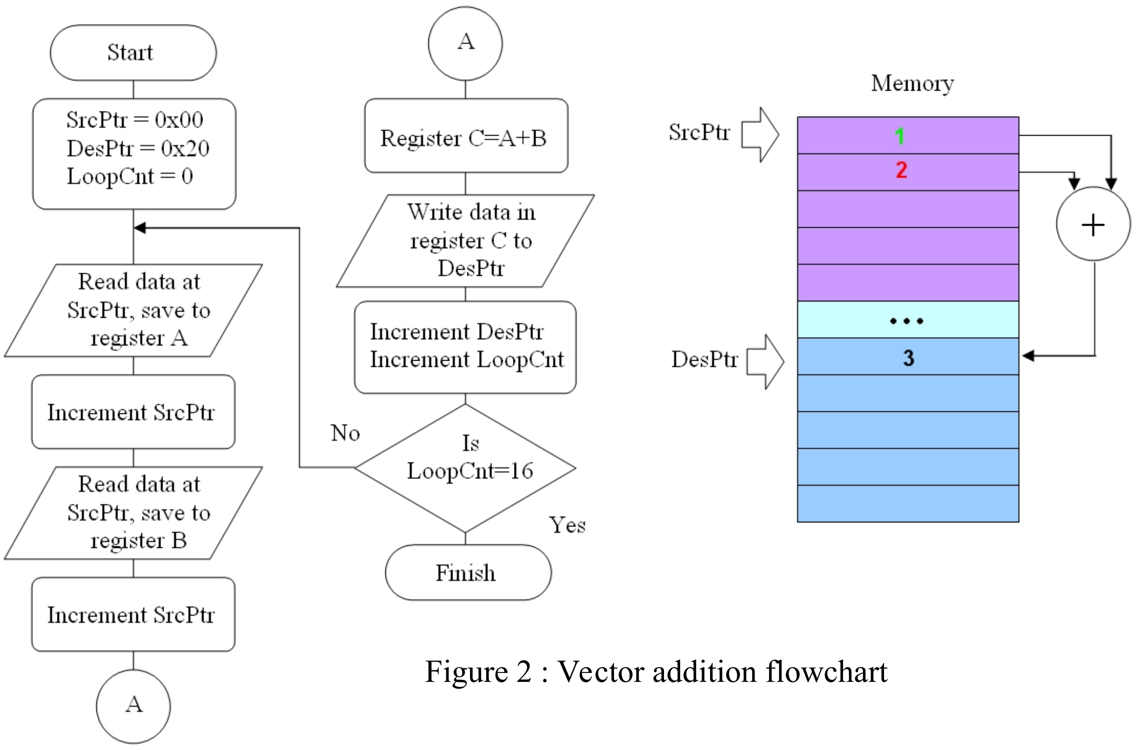
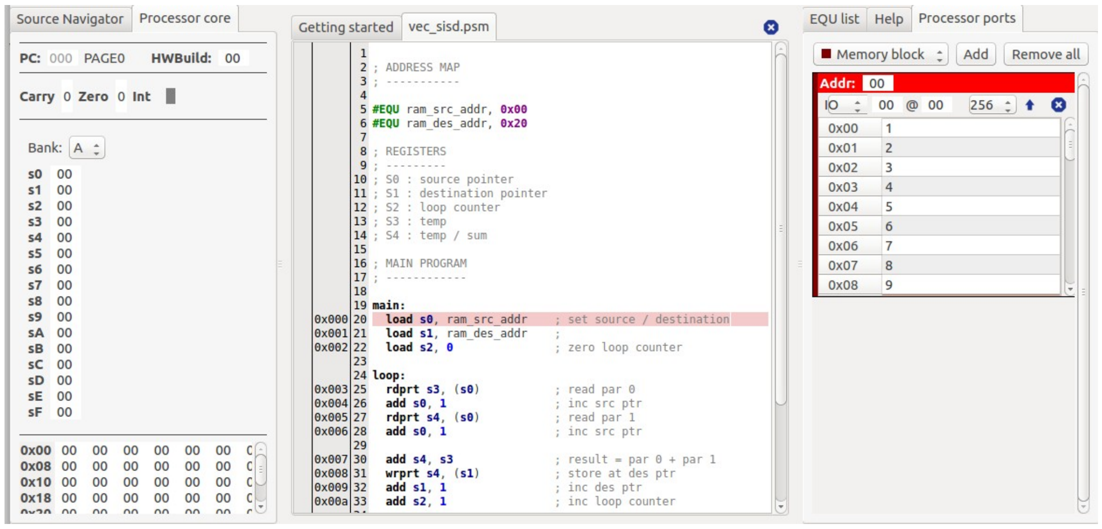


Figure 2 : Vector addition flowchart

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Address	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Data	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Address	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Data	3	7	11	15	19	23	27	31	35	39	41	47	51	55	59	63

Figure 3: RAM input data values (top), output data values (bottom)



```
PC: 000  PAGE0    HWBuild: 00

Carry 0 Zero 0 Int

Bank: A

s0  00
s1  00
s2  00
s3  00
s4  00
s5  00
s6  00
s7  00
s8  00
s9  00
sA  00
sB  00
sC  00
sD  00
sE  00
sF  00

0x00  00  00  00  00  00  00
0x08  00  00  00  00  00  00
0x10  00  00  00  00  00  00
0x18  00  00  00  00  00  00
```

```
      ; ADDRESS MAP
      ; -----------

      #EQU ram_src_addr, 0x00
      #EQU ram_des_addr, 0x20

      ; REGISTERS
      ; ---------
      ; S0 : source pointer
      ; S1 : destination pointer
      ; S2 : loop counter
      ; S3 : temp
      ; S4 : temp / sum

      ; MAIN PROGRAM
      ; ------------

      main:
0x000   load s0, ram_src_addr     ; set source / destination
0x001   load s1, ram_des_addr     ;
0x002   load s2, 0                ; zero loop counter

      loop:
0x003   rdprt s3, (s0)            ; read par 0
0x004   add s0, 1                 ; inc src ptr
0x005   rdprt s4, (s0)            ; read par 1
0x006   add s0, 1                 ; inc src ptr

0x007   add s4, s3                ; result = par 0 + par 1
0x008   wrprt s4, (s1)            ; store at des ptr
0x009   add s1, 1                 ; inc des ptr
0x00a   add s2, 1                 ; inc loop counter
```

```
Addr: 00
IO  00 @ 00  256
0x00  1
0x01  2
0x02  3
0x03  4
0x04  5
0x05  6
0x06  7
0x07  8
0x08  9
```

Figure 4: FIDEx RAM configuration

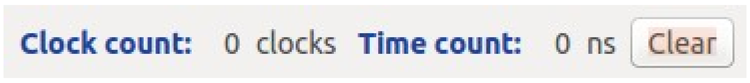
Note, you can see in figure 3 one input vector is stored at even address locations (red addresses), the other at odd address locations (green addresses). The output vector i.e. the sum of the two input vectors is stored at address 32 onwards (blue addresses).

A rough estimate of the processing requirements of any program can be obtained by counting the number of instructions executed. Each instruction in the PicoBlaze instruction set takes two cycles to execute, therefore the time to process this data is given by:

```
Processing time = Instructions Executed × 2 × Clock period
```

If the typical clock speed for the PicoBlaze processor is 50MHz, estimate how long it will take to execute this program.

Hint: the number of instructions in the program may differ from the number of instructions executed i.e. looping software programming structures. Within FIDEx the number of clock cycles executed is shown in the bottom bar.



Task 2

A Vivado project called `PicoBlaze_VHDL_SISD`, has already been created for this program and can be downloaded from the module's VLE page. A block diagram of this processing architecture is shown in figure 5. Using your preferred web browser, download the file `PicoBlaze_VHDL_SISD.zip` to c:\temp (or your home directory). Right click on this file selecting 'Extract all…' to unzip it, start Vivado and open this project as previously described.

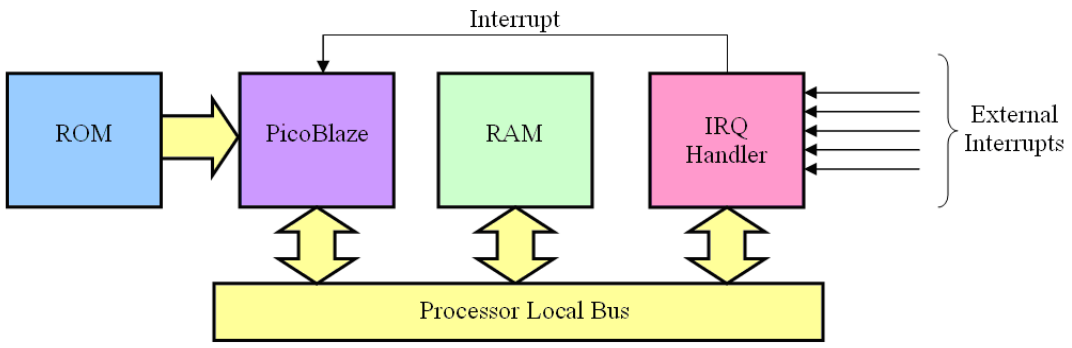


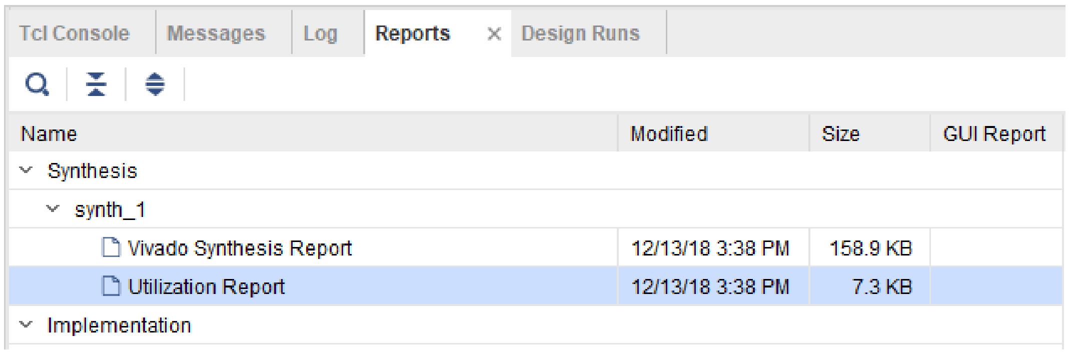
Figure 5: System architecture block diagram

To determine how fast this processor can be clocked on a Xilinx FPGA (the actual silicon) you will need to synthesise the VHDL files i.e. convert these files into low level logic gates.

Click on the ▶ Run Synthesis icon. Click OK to re-run this process.

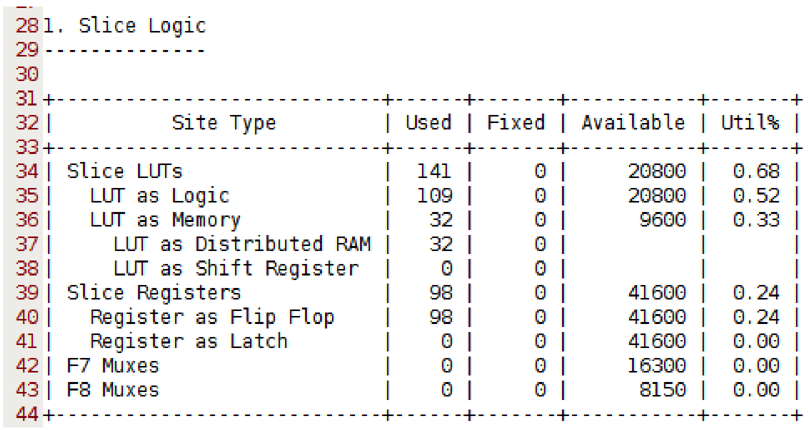
The software tools will now synthesize (convert) the VHDL models into hardware i.e. logic gates & registers, whilst it is doing this a progress bar is displayed in the top right of the screen Running synth_design (message displayed will change as it goes through the various stages)

When complete the *Implementation Completed* box will appear. Select the *View Reports* option, then click on OK to continue. The report window is at the bottom of the screen. Double click on the *Utilisation Report*, as shown in figure 6, then scroll down this report until you see the *Slice Logic* table, as shown in figure 7.



Name	Modified	Size	GUI Report
Synthesis			
synth_1			
Vivado Synthesis Report	12/13/18 3:38 PM	158.9 KB	
Utilization Report	12/13/18 3:38 PM	7.3 KB	
Implementation			

Figure 6: Utilization report



```
1. Slice Logic
--------------

+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs                 |  141 |     0 |     20800 |  0.68 |
|   LUT as Logic             |  109 |     0 |     20800 |  0.52 |
|   LUT as Memory            |   32 |     0 |      9600 |  0.33 |
|     LUT as Distributed RAM |   32 |     0 |           |       |
|     LUT as Shift Register  |    0 |     0 |           |       |
| Slice Registers            |   98 |     0 |     41600 |  0.24 |
|   Register as Flip Flop    |   98 |     0 |     41600 |  0.24 |
|   Register as Latch        |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                   |    0 |     0 |     16300 |  0.00 |
| F8 Muxes                   |    0 |     0 |      8150 |  0.00 |
+----------------------------+------+-------+-----------+-------+
```

Figure 7: Slice LUTs used

The basic building block within the FPGA is a *Slice*. These contain logic: look up tables (LUT) and flip-flops. This implementation of the PicoBlaze needs 141 LUT and 98 flip-flops (numbers may vary slightly with Vivado version). Make a record of these figures so that the hardware requirements of each system can be compared.

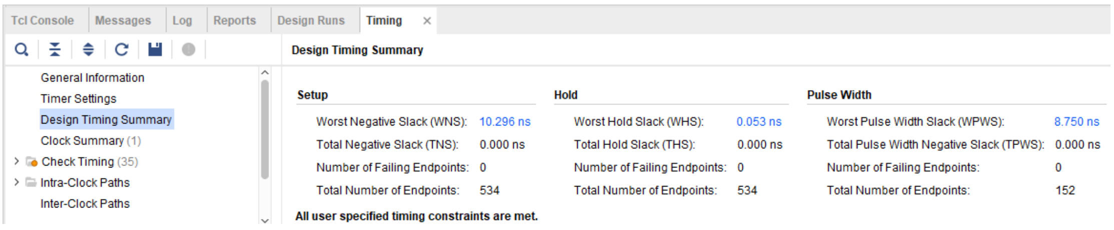
Next, under the Run Synthesis icon click on *Open Synthesized Design* pull-down menu. Then click on *Report Timing Summary*, click OK when the timing window opens. This will open new tab: *Timing*, in the bottom Tcl Console window.
This timing report shows how quickly this hardware can be clocked. Remember the maximum clock speed of the processor is determined by the critical path delay i.e. the “biggest” block of logic gates, with the longest input to output logic path.

This timing report compares the performance of this system to a 50MHz clock speed i.e. how much timing 'slack' (spare time) is there compared to a 20ns period.

Therefore, taking the *Setup* slack timing value:

```
Max frequency = 1 ÷ (20ns – 10.296ns) = 103 MHz
```

Again, note down this value as it will be used to assess the performance of each system.



Design Timing Summary

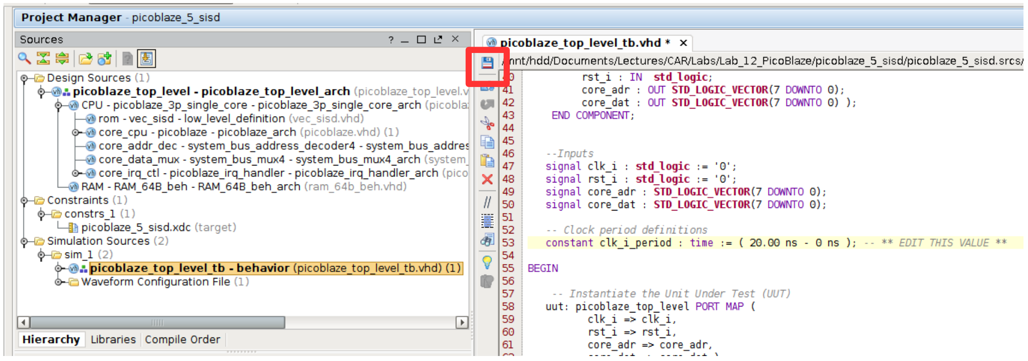
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	10.296 ns	Worst Hold Slack (WHS):	0.053 ns	Worst Pulse Width Slack (WPWS):	8.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	534	Total Number of Endpoints:	534	Total Number of Endpoints:	152

All user specified timing constraints are met.

Figure 8: Timing report

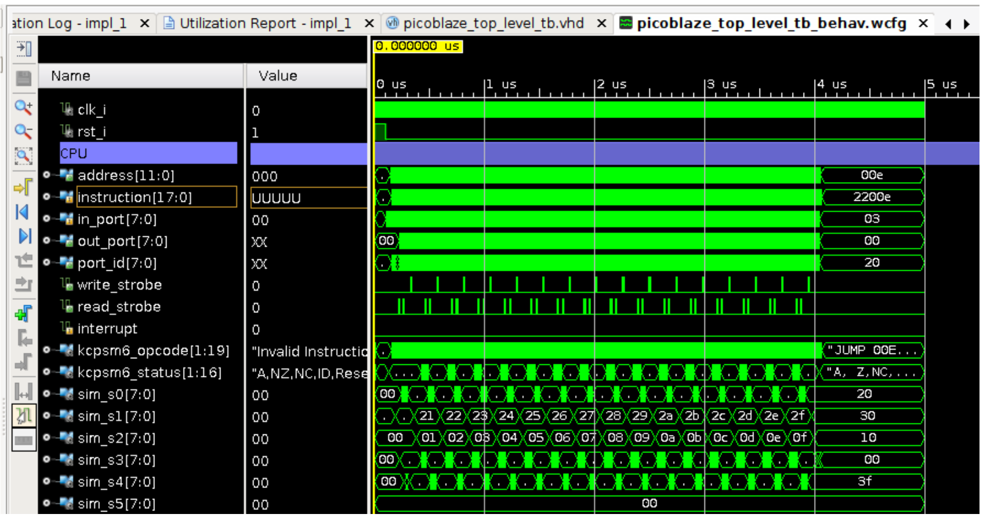
Task 3

Within the Sources window double click on the file `picoblaze_top_level_tb`. This will open the top level test bench file, as shown in figure 9 i.e. the main VHDL simulation model, **update** the line shown below with the processor's calculated minimum clock period, replacing 0ns with 10.296ns. Then save this file by clicking on the save icon 💾, or by pressing `CTRL S`.


Project Manager - picoblaze_5_sisd
Sources
Design Sources (1)
picoblaze_top_level - picoblaze_top_level_arch (picoblaze_top_level.v
CPU - picoblaze_3p_single_core - picoblaze_3p_single_core_arch (picobla
rom - vec_sisd - low_level_definition (vec_sisd.vhd)
core_cpu - picoblaze - picoblaze_arch (picoblaze.vhd) (1)
core_addr_dec - system_bus_address_decoder4 - system_bus_addres
core_data_mux - system_bus_mux4 - system_bus_mux4_arch (system_
core_irq_ctl - picoblaze_irq_handler - picoblaze_irq_handler_arch (pico
RAM - RAM_64B_beh - RAM_64B_beh_arch (ram_64b_beh.vhd)
Constraints (1)
constrs_1 (1)
picoblaze_5_sisd.xdc (target)
Simulation Sources (2)
sim_1 (2)
picoblaze_top_level_tb - behavior (picoblaze_top_level_tb.vhd) (1)
Waveform Configuration File (1)
Hierarchy
Libraries
Compile Order
picoblaze_top_level_tb.vhd *
/mnt/hdd/Documents/Lectures/CAR/Labs/Lab_12_PicoBlaze/picoblaze_5_sisd/picoblaze_5_sisd.srcs/

```
          rst_i : IN  std_logic;
          core_adr : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
          core_dat : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) );
     END COMPONENT;


    --Inputs
    signal clk_i : std_logic := '0';
    signal rst_i : std_logic := '0';
    signal core_adr : STD_LOGIC_VECTOR(7 DOWNTO 0);
    signal core_dat : STD_LOGIC_VECTOR(7 DOWNTO 0);

    -- Clock period definitions
    constant clk_i_period : time := ( 20.00 ns - 0 ns ); -- ** EDIT THIS VALUE **

BEGIN

     -- Instantiate the Unit Under Test (UUT)
    uut: picoblaze_top_level PORT MAP (
           clk_i => clk_i,
           rst_i => rst_i,
           core_adr => core_adr,
```

Figure 9: Test bench

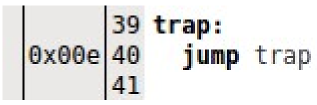


Name	Value
clk_i	0
rst_i	1
CPU	
address[11:0]	000
instruction[17:0]	UUUUU
in_port[7:0]	00
out_port[7:0]	XX
port_id[7:0]	XX
write_strobe	0
read_strobe	0
interrupt	0
kcpsm6_opcode[1:19]	"Invalid Instructio
kcpsm6_status[1:16]	"A,NZ,NC,ID,Rese
sim_s0[7:0]	00
sim_s1[7:0]	00
sim_s2[7:0]	00
sim_s3[7:0]	00
sim_s4[7:0]	00
sim_s5[7:0]	00

Figure 10: Simulation waveform

To simulate this Picoblaze system click on the Run Simulation icon, within the Simulation panel (middle, left side). Then select 'Run behavioural simulation' from the options given. This will open a waveform trace as shown in figure 10.

To step through the simulation click on the icon, this will simulate the hardware in 1us time chunks (defined in the text box). Simulate this program for 5 us, during this time the processor's hardware is simulated executing the vector addition program `vec_SISD.psm`, when it is finished the code will enter an infinite loop, as shown below:



```
      trap:
0x00e   jump trap
```

Within the simulation this point can be identified by looking at the processor's address bus, as it will not change i.e. it will always jump to address 0x0E. How long did it take to execute this program? We can now use this is the architecture's hardware size (slices) and executions time as a baseline, allowing us to compare the relative performance of the other three processor classifications (MISD, SIMD and MIMD).

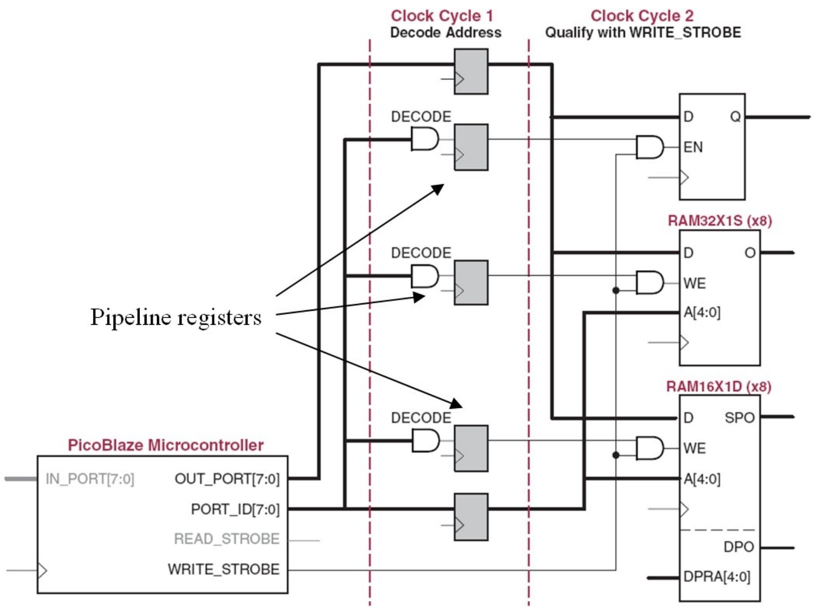


Figure 11: Pipeline registers, separating address decode and peripheral devices

Task 4

Pipelining techniques can be used to increase a processor's maximum clock speed and therefore the number of instructions that are executed per second. This is achieved by dividing up large logic blocks through the insertion of registers, reducing the worst case propagation delays i.e. the critical path delay. A Vivado project called `PicoBlaze_VHDL_MISD`, has already been created for this architecture and can be downloaded from the module web page.

The block diagram of this new architecture is the same as figure 5, as the modifications are internal to the processor, as illustrated in figure 11. Using your preferred web browser, download the file `PicoBlaze_VHDL_MISD.zip` to `c:\`

`temp` (or your home directory). Right click on this file selecting 'Extract all…' to unzip it, start Vivado and open this project.

Note, no modification to the software is required, a significant advantage for legacy code. In general pipelining should always improve the performance of a processor, but as we will see in later laboratories, this is not always true (control hazards).

To determine how fast this new processor can be clocked on a Xilinx FPGA you will need to synthesize the VHDL files as described in task 2. Scroll through the *Utilisation Report* file noting the number of slices and flip-flops required i.e. the size of the hardware. Next, generate the *Timing Report* file to determine the Minimum period (Maximum frequency).

Note, you should see a clock speed improvement, but at what cost?

As before double left click on the file `picoblaze_top_level_tb` and **update** the clock period, replacing 0ns with the *Setup* slack timing value. Simulate this design in 1us time chunks until the 'trap' instruction is executed i.e. address 0x0E. How long did it take to execute this program? Why is it faster?

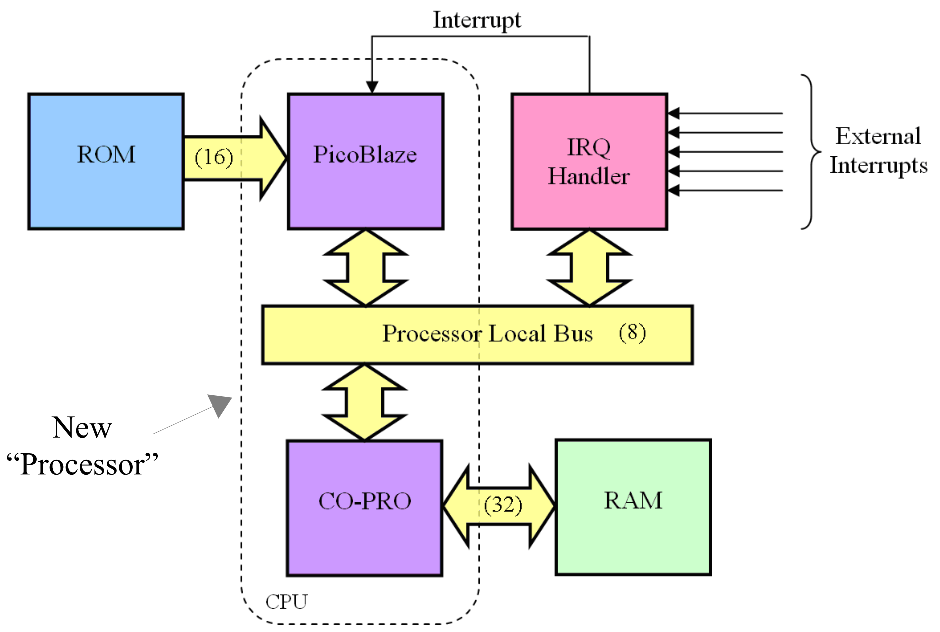


Figure 12: System architecture block diagram

Task 5

Hardware replication techniques can be used to increase the number of operations performed in parallel and therefore reduce the number of instructions required to implement a program. This is achieved through the addition of complex, multi-operand instructions to a processor's instruction set. Unfortunately, the PicoBlaze processor does not directly support these types of instructions. However, they can be approximated using a co-processor i.e. complex data processing hardware, added into the processor's memory space. For the purpose of this lab this new 'CPU' can be considered to be the combination of a SISD PicoBlaze processor and a co-processor, as shown in figures 12.

A Vivado project called `PicoBlaze_VHDL_SIMD`, has already been created for this architecture and can be downloaded from the module webpage. Using your preferred web browser, download the file `PicoBlaze_VHDL_SIMD.zip` to `c:\temp` (or your home directory). Right click on this file selecting 'Extract all…' to unzip it, start Vivado and open this project as previously described.

SIMD architectures process data in packets i.e. an array, containg multiple values. To support these SIMD instructions each memory location will now store 32 bits i.e. a packet of four 8bit data values, as show in figure 14. In this example the address is given in word (32bit) addressable format, each address holding two values of the input vectors. The previous example in figure 3 uses a byte (8bit) addressable format.

The co-processor hardware implements a vector addition, `VADD` 'instruction', which performs the following steps:

(1) Read a data packet from memory (32bits containing four 8bit data values)
(2) Add each data pair (two 8bit additions)
(3) Write two 8bit results to data memory (a 16bit half word)

Note, memory (RAM) has been removed from the PicoBlazes local bus as it now uses a 32bit data bus (shown in figure 12), all memory transactions now pass through the co-processor. The addition of the four data values is performed in the co-processor using the hardware shown in figure 13 e.g. 1+2=3 and 3+4=7 are performed in parallel.

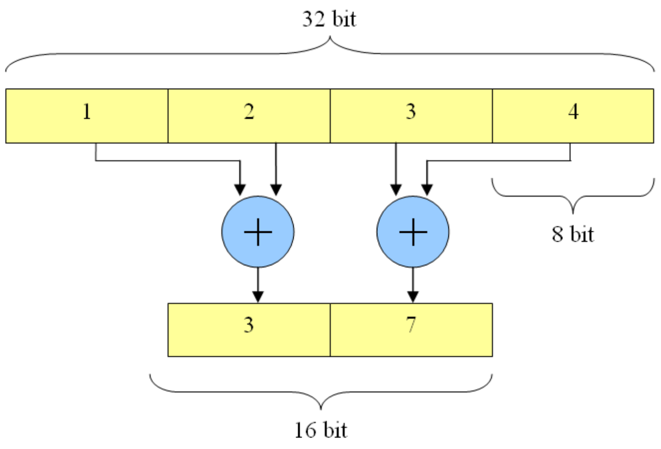


Figure 13: SIMD VADD hardware

Address	0				1				2				3			
Data	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Address	4				5				6				7			
Data	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Address	32				33				34				35			
Data	3	7	11	15	19	23	27	31	35	39	41	47	51	55	59	63

Figure 14: RAM input data values (top), output data values (bottom)

The `VADD` 'instruction' can not be used directly within a PicoBlaze assembly language program, rather the opcode and operands must be passed to the co-processor using `WRPRT` instructions i.e. a remote procedure call (RPC), passing parameters to registers within the co-processor which then performs the specified procedure, as shown in figure 15.

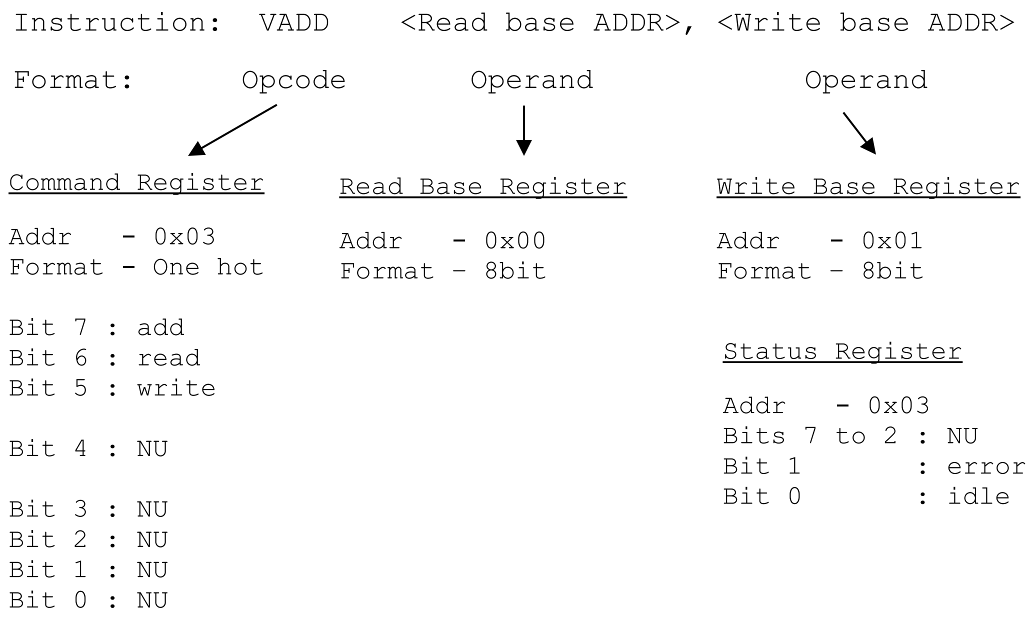


Figure 15: new 'instruction' format

The co-processor has four memory mapped registers:

- Read : address 0x00, pass to the co-processor the 8bit start address of input vectors. In the example code this is `ram_src_addr`, address 0x00. This is address zero in the co-processor's memory i.e. the green memory block in figure 12.
- Write : address 0x01, pass to the co-processor the 8bit result address of output vector. In the example code this is `ram_des_addr`, address 0x20. Again, this address is in the co-processor's memory, address 32 as shown in figure 14.
- Command : address 0x03, pass to the co-processor a one-hot command code. In the example code this is `ADD_CMD`, address 0x80, which triggers the co-processor to start the addition of the two vectors.
- Status : address 0x03, allows the PicoBlaze to read the status of the co-processor e.g. idle, error etc.
- Data : address 0x02, allows the PicoBlaze to read or write to the co-processor's memory i.e. acts as a bridge between the two address spaces.

The vector addition operation is triggered by the PicoBlaze when it writes the command code to address 0x03 (command register), therefore, the read and write registers must be updated first. The processor must then wait for the co-processor to complete this 'instruction' before proceeding, this is achieved by repeatedly testing the status register to see if the co-processor is idle.

Note, originally I was going to modify the PicoBlaze's instruction decoder and increase its data bus width to 32bits so that I could add SIMD instructions, but this proved to be too much fun :).

An assembly language program to implement this functionality has already been created and can be downloaded from the module webpage: `vec_SIMD.psm`, examine this code and identify its key features.

Note, the `VADD` 'instruction' generates a 16bit result, but its memory is 32bits wide, therefore, the co-processor writes data to the high half word for even addresses and the low half word for odd addresses.

To determine how fast this new processor can be clocked on a Xilinx FPGA you will need to synthesize the VHDL files as described in task 2. Scroll through the *Utilisation Report* file noting the number of slices and flip-flops required i.e. the size of the hardware. Next, scroll to the *Timing Report* file to determine the Minimum period (Maximum frequency).

Why do these figures differ from the MISD system? Should its performance be faster or slower than this system?

Note, you should see a clock speed reduction for the SIMD system.

Task 6

How many simple picoBlaze instructions does the `VADD` 'instruction' replace e.g. memory accesses, arithmetic operations etc? Compared to the SISD architecture estimate how much faster this new implementation should be.

As before double left click on the file `picoblaze_top_level_tb` and **update** the clock period, replacing 0ns with the *Setup* slack timing value. Simulate this design in 1us time chunks until the 'trap' instruction is executed i.e. Jump trap. How long did it take to execute this program? Why is it faster when the clock speed is slower?

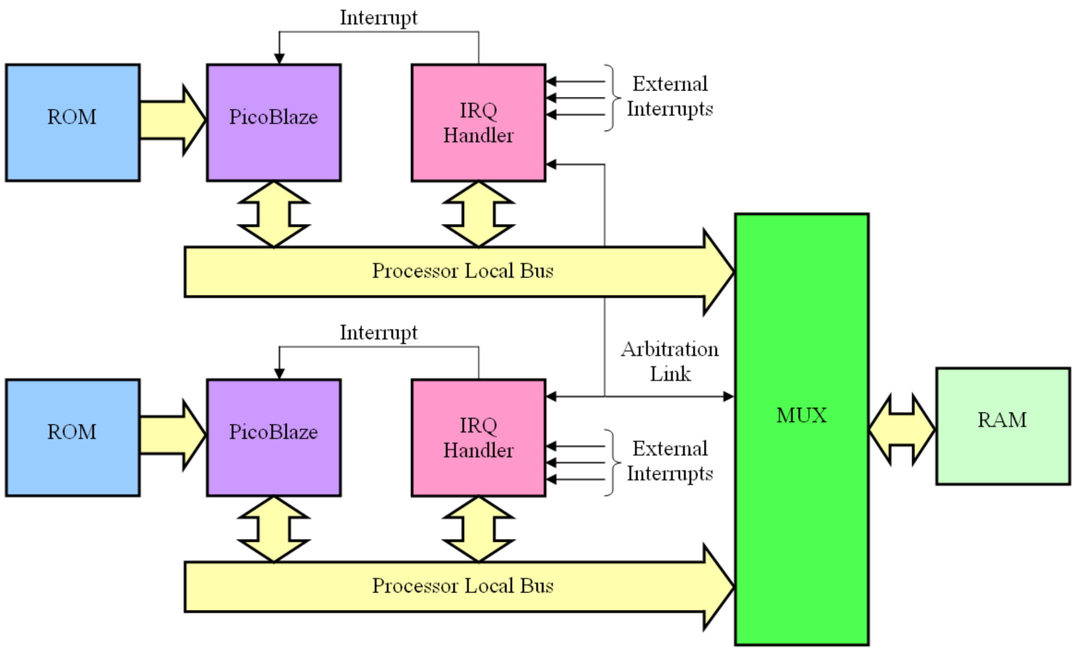


Figure 16: Parallel PicoBlaze architecture

Examine the waveform window, what is the main bottle neck in this program's execution i.e. what does this program spend most of its time doing? If this hardware was integrated in to the PicoBlaze can you estimate how much faster this program would run i.e. if you did not need to write and read to the memory mapped registers?

Task 7

An alternative method to improve processing performance is to use multiple processors, each working on a subset of the data, as shown in figure 16. In this example each processor calculates the sum of 8 pairs of data values stored in a shared RAM i.e. half the data. As this memory can only read or write one value at a time additional arbitration logic is included in the IRQ handler IP-cores i.e. implementing mutual exclusion synchronisation, only allowing one processor to access memory at a time.

An assembly language program to implement the required functionality has already been created and can be downloaded from the module webpage. Using your preferred web browser, download the file `vec_MIMD.psm`.

Each processor uses the **same** program, therefore, additional coding is required to allow each processor to determine what subset of the data it should process. This is performed in software using the arbiter's unique ID code (hardwired into each IRQ handler) i.e. this hard-coded value is read by each processor, allowing it to select the correct source / destination address pointers.

Note, this approach of using a common program that determines what operations it should perform based on its node ID is used in parallel programming "languages" e.g. Message Passing Interface (MPI) library.

Access to the shared memory is controlled through the arbiter's status register. Before the processor tries to access memory it reads the `arb_status` register. If the value returned is zero this processor can not access the shared memory and must wait i.e. memory is currently being used by the other processor. If the value is a non zero value this processor may access memory. Hardware within the arbiter's control logic then automatically sets the `arb_status` register to zero i.e. locking out the other processor.

When a processor gains control of the shared memory it enters its critical section i.e. parts of the program that read or write to the shared memory. When complete the program can relinquish control of the shared memory by performing a dummy write to the `arb_status` register, resetting the `arb_status` register to a non zero value. This change in status then allows the other processor to access this memory.

Examine the code in `vec_MIMD.psm` and identify these key features. Estimate how long it will take for this system to perform the vector addition. Compared to the SISD architecture estimate how much faster this new implementation should be.

Note, knowing that the program performs the addition of 8 values and that each instruction requires 2 clock cycles, you can estimate the program's execution time by counting the number of instructions executed in the main loop.

Task 8

A Vivado project called `PicoBlaze_VHDL_MIMD`, has already been created for this architecture and can be downloaded from the module webpage. Using your preferred web browser, download the file `PicoBlaze_VHDL_MIMD.zip` to c:\temp (or your home directory). Right click on this file selecting 'Extract all…' to unzip it, start Vivado and open this project as previously described.

To determine how fast this new processor can be clocked on a Xilinx FPGA you will need to synthesize the VHDL files as described in task 2. Scroll through the *Utilisation Report* file noting the number of slices and flip-flops required i.e. the size of the hardware. Next, scroll to the *Timing Report* file to determine the Minimum period (Maximum frequency).

As before double left click on the file `picoblaze_top_level_tb` and **update** the clock period, replacing 0ns with the *Setup* slack timing value. Simulate this design in 1us time chunks until the 'trap' instruction is executed i.e. Jump trap.

How long did it take to execute this program? Why is it slower than the SISD architecture? What is the reason for the difference in performance? Examine the waveform window, what is the main bottleneck in this program's execution?

Note, how and when are the processors accessing the shared memory?

Task 9

Using the synthesis results obtained answer the following:

- Calculate the relative speedup of each architecture compared to a SISD architecture.

$$\text{Speedup} = \frac{\text{Time to execute program on SISD architecture}}{\text{Time to execute program on new architecture}}$$

- Compare the relative hardware requirements of each architecture, which gave the best hardware vs performance return for the selected application.
- Consider what types of applications would be best suited for the pipelined (MISD) and parallel (SIMD & MIMD) architectures. What are the possible advantages and disadvantages of these architectures? What complex instruction would more efficiently support this program?

Appendix A : Interrupt Controller

The PicoBlaze processor has only one interrupt pin, however, in a real system multiple external sources may wish to interrupt the processor, i.e. trigger the execution of specific ISR's to handle external events. To support this a hardware interrupt controller is used. The processor can configure this hardware component by writing data to the `irq_controller_cmd` register. This register can enable 0 – 7 external interrupt sources (pins), logging which interrupt line has been pulsed and assigning it a priority, before interrupting the processor.

```
command register : irq_controller_cmd
-------------------------------------
Bit 7 : irq_7 enable (1=enable, 0=disable)
Bit 6 : irq_6 enable (1=enable, 0=disable)
Bit 5 : irq_5 enable (1=enable, 0=disable)
Bit 4 : irq_4 enable (1=enable, 0=disable)
Bit 3 : irq_3 enable (1=enable, 0=disable)
Bit 2 : irq_2 enable (1=enable, 0=disable)
Bit 1 : irq_1 enable (1=enable, 0=disable)
Bit 0 : irq_0 enable (1=enable, 0=disable)
```

Figure A1 : `irq_controller_cmd` register bit fields

The processor can identify the source of the interrupt by reading the `irq_controller_trig` register, as shown in figure A2. This returns a one-hot value indicating the highest priority interrupt source. Interrupt 0 is the highest priority, interrupt 7 is the lowest priority e.g. if interrupts 2 and 4 occur at the same time the `irq_controller_trig` register will return the value 0x04, indicating that interrupt 2 (bit position 2) was triggered. When the ISR for this interrupt is complete a second interrupt will be generated by the interrupt controller for interrupt 4, returning the value 0x10. Reading this register will clear the active interrupt flag. Whilst processing an interrupt the PicoBlaze automatically disables interrupts, these can be re-enabled using the interrupt return instruction i.e. you can not interrupt and interrupt service routine.

```
Trigger register (one hot) : irq_controller_trig
------------------------------------------------
Bit 7 : irq_7  (1=enable, 0=disable)
Bit 6 : irq_6  (1=enable, 0=disable)
Bit 5 : irq_5  (1=enable, 0=disable)
Bit 4 : irq_4  (1=enable, 0=disable)
Bit 3 : irq_3  (1=enable, 0=disable)
Bit 2 : irq_2  (1=enable, 0=disable)
Bit 1 : irq_1  (1=enable, 0=disable)
Bit 0 : irq_0  (1=enable, 0=disable)
```

Figure A2 : `irq_controller_trig` register bit fields