OpenCL FPGA Lab 2 – Fast Fourier Transformations

Introduction :

By now you have had some experience with writing OpenCL kernels as well as applying optimisations to them. In this Lab, you will apply these optimisations (and some new ones!) to FFT kernels.

Background:

FFT is a critical step in a number of modern scientific applications where the k-space transformation allows time domain computations, such as convolutions, to be performed with O(NlogN) complexity. Radix-2 FFTs match well with FPGAs since the complex communication pattern between the logN stages can be easily supported, stall free, due to flexible interconnects and high throughput of register arrays serving as pipeline registers.

For a refresher on what exactly this form of FFT does, check out the following website:

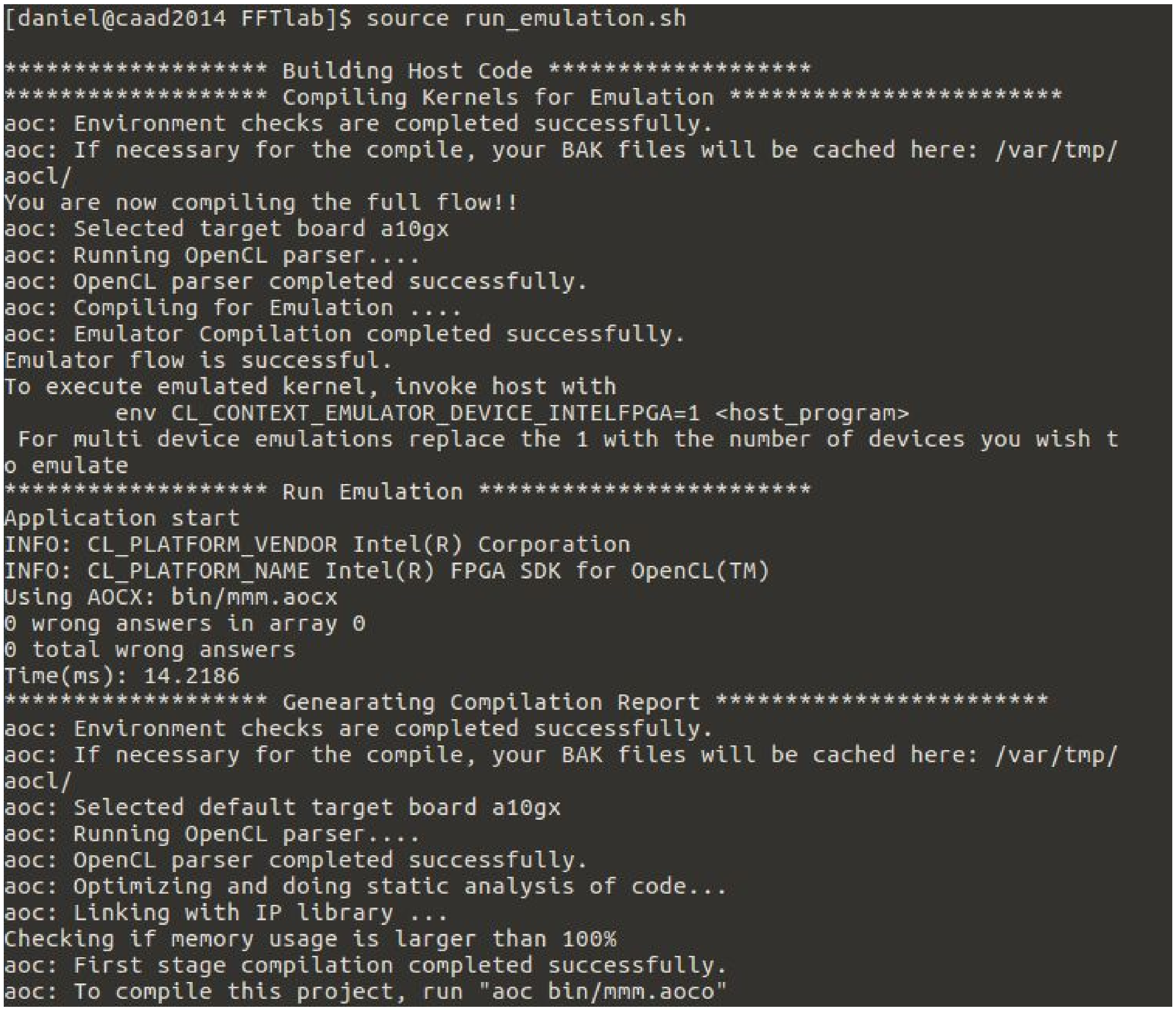
http://en.dsplib.org/content/fft_dec_in_time/fft_dec_in_time.html

Part One:

1. Navigate to lab2/part1/host/src . Open *main.cpp* with your preferred editor. Read through the code.
    a. If you scroll to the bottom, you will see the host-side FFT code. This will run in addition to the kernel with the same inputs to verify correctness. Notice how it uses the complex<> data structure.

2. Navigate to lab2/part1/device . Then, open the kernel *mmm.cl* with your preferred editor. Read through the code. Notice the following:
    a. The input parameters. Four floating point arrays are used because opencl kernels do not support c++'s complex data structure. FFT is done on *N* 8-element long arrays, and separate arrays are used for the real and imaginary values.
    b. The *__constant w_r[]* and *w_i[]* arrays. These correspond to the twiddle factors. Constant memory is loaded into on-chip constant cache, which allows for fast access. Why can twiddle factors be put in constant memory?

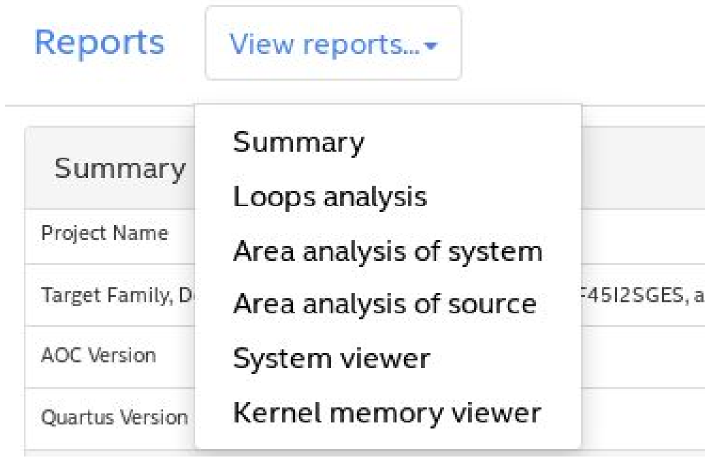
3. To verify the code works, navigate back to lab2/part1 and run the code in emulation.
    a. Type the command "source run_emulation.sh" .The output should resemble the following:



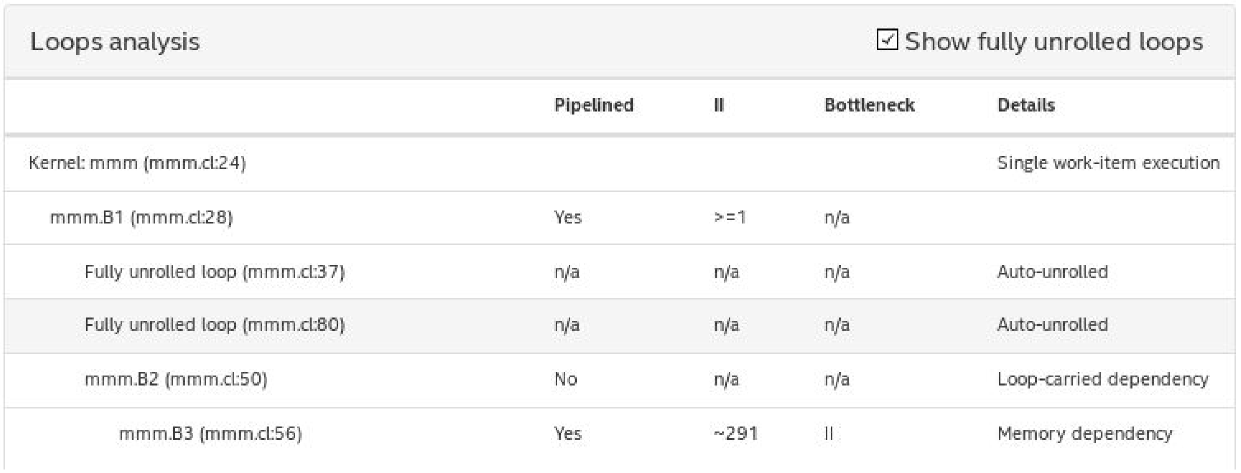
```
[daniel@caad2014 FFTlab]$ source run_emulation.sh

******************* Building Host Code *******************
******************* Compiling Kernels for Emulation ************************
aoc: Environment checks are completed successfully.
aoc: If necessary for the compile, your BAK files will be cached here: /var/tmp/
aocl/
You are now compiling the full flow!!
aoc: Selected target board a10gx
aoc: Running OpenCL parser....
aoc: OpenCL parser completed successfully.
aoc: Compiling for Emulation ....
aoc: Emulator Compilation completed successfully.
Emulator flow is successful.
To execute emulated kernel, invoke host with
        env CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=1 <host_program>
 For multi device emulations replace the 1 with the number of devices you wish t
o emulate
******************* Run Emulation ************************
Application start
INFO: CL_PLATFORM_VENDOR Intel(R) Corporation
INFO: CL_PLATFORM_NAME Intel(R) FPGA SDK for OpenCL(TM)
Using AOCX: bin/mmm.aocx
0 wrong answers in array 0
0 total wrong answers
Time(ms): 14.2186
******************* Genearating Compilation Report ************************
aoc: Environment checks are completed successfully.
aoc: If necessary for the compile, your BAK files will be cached here: /var/tmp/
aocl/
aoc: Selected default target board a10gx
aoc: Running OpenCL parser....
aoc: OpenCL parser completed successfully.
aoc: Optimizing and doing static analysis of code...
aoc: Linking with IP library ...
Checking if memory usage is larger than 100%
aoc: First stage compilation completed successfully.
aoc: To compile this project, run "aoc bin/mmm.aoco"
```

4. Be sure your programme output - the part between Run Emulation and Generating Compilation Report - matches the sample output above.
5. Now, navigate to lab2/part1/reports and open the report with firefox report.html

a. Go to the Loops analysis tab.



b. Take a look at the information it shows and which loop corresponds to which. II stands for Initiation Interval, or how often successive loop iterations are launched. The ideal value is 1.



Loops analysis ☑ Show fully unrolled loops

	Pipelined	II	Bottleneck	Details
Kernel: mmm (mmm.cl:24)				Single work-item execution
mmm.B1 (mmm.cl:28)	Yes	>=1	n/a	
Fully unrolled loop (mmm.cl:37)	n/a	n/a	n/a	Auto-unrolled
Fully unrolled loop (mmm.cl:80)	n/a	n/a	n/a	Auto-unrolled
mmm.B2 (mmm.cl:50)	No	n/a	n/a	Loop-carried dependency
mmm.B3 (mmm.cl:56)	Yes	~291	II	Memory dependency

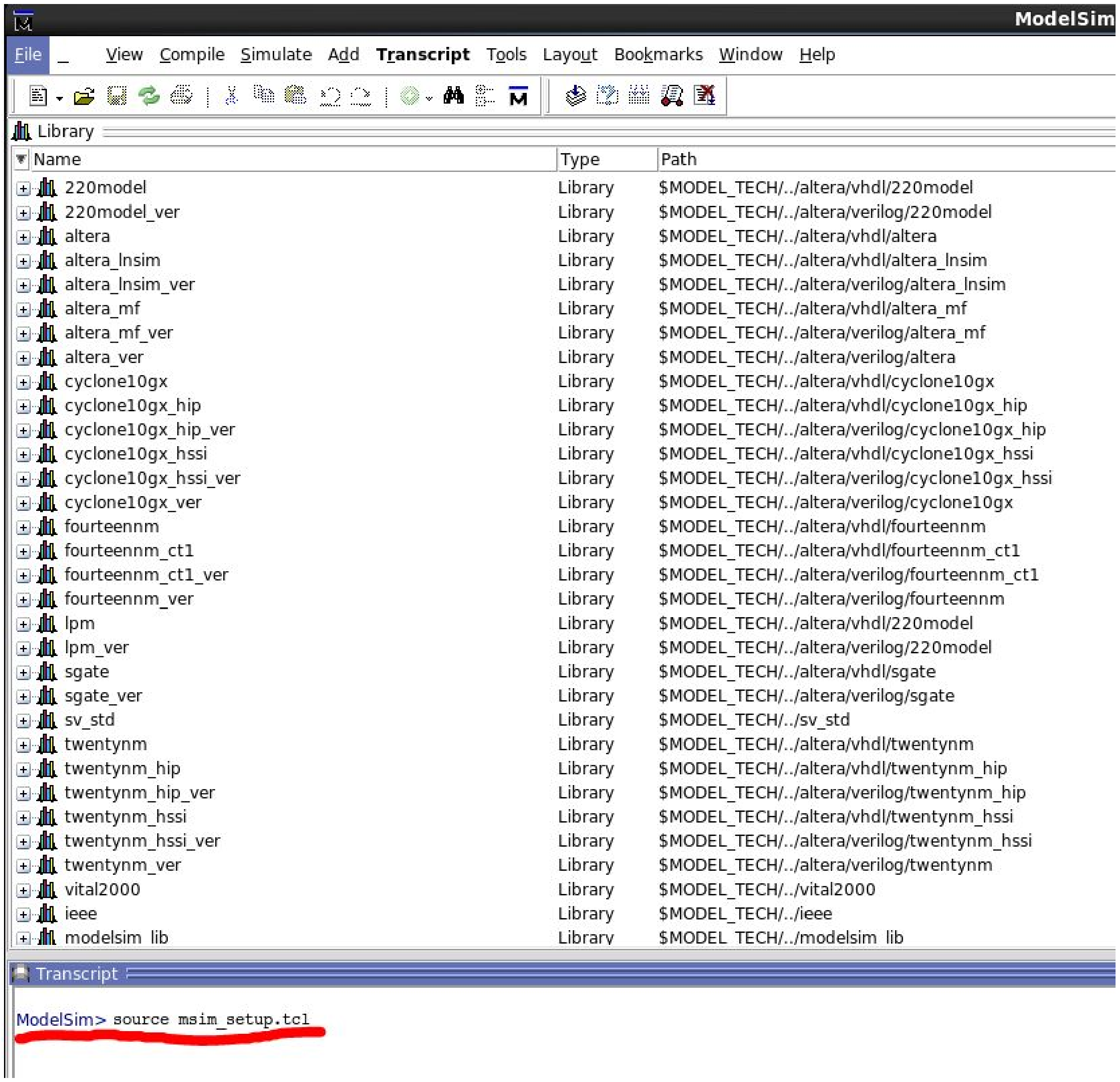
c. Your report will tell you that the first and last loops have been unrolled automatically. These loops load in_real and in_img into the temp array and load the temp array into out_real and out_temp. The compute loops are not unrolled due to a "Loop-carried dependency." So pipeline stages are not evenly divide: one stage does all of the computation while the others load and store. We will resolve that issue in part 2.

6. Now you can run the simulation and get a baseline for kernel performance. Navigate back to lab2/part1/ and type the command "source run_simulation". When the script is finished, you will follow the procedure mentioned in earlier labs for running the simulation. That procedure is included below:

After that, a ModelSim window will be launched. In the transcript window, type:

"source msim_setup.tcl"

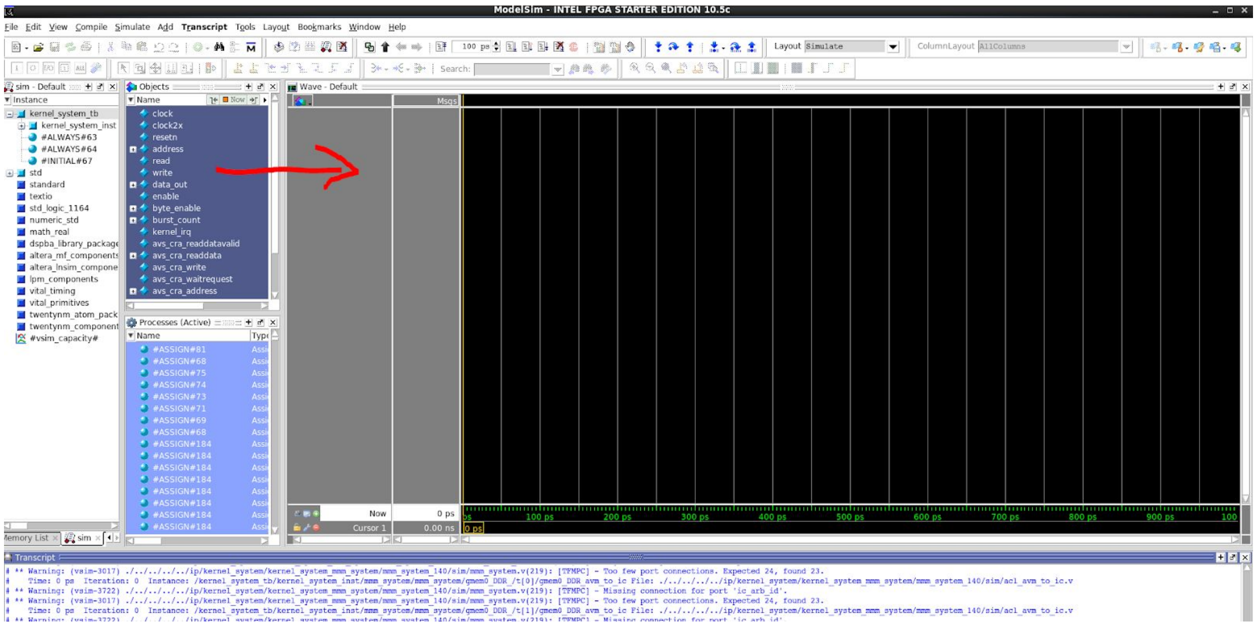
"ld"   (this may take 3-5 minutes)


ModelSim
File _ View Compile Simulate Add Transcript Tools Layout Bookmarks Window Help
Library
Transcript

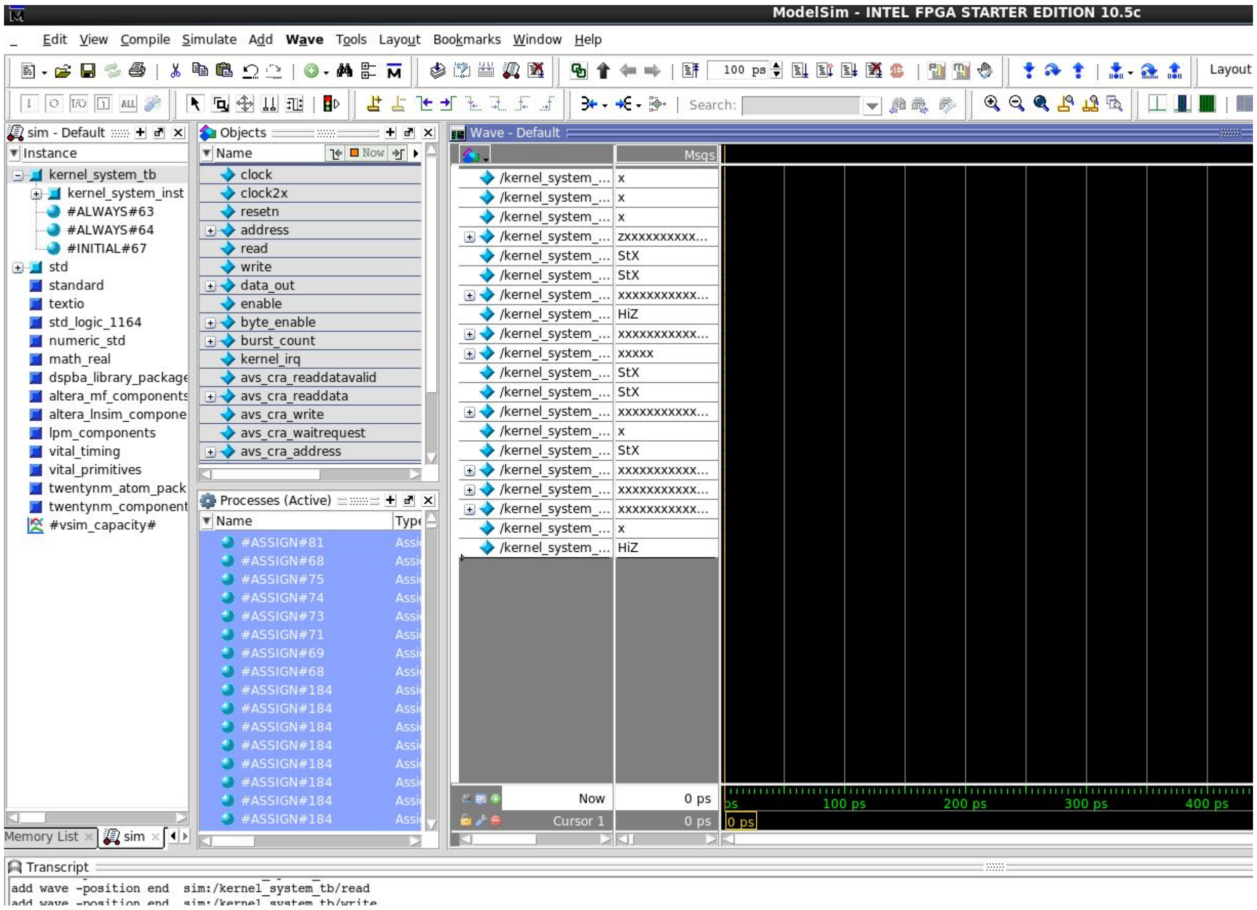
Name	Type	Path
220model	Library	$MODEL_TECH/../altera/vhdl/220model
220model_ver	Library	$MODEL_TECH/../altera/verilog/220model
altera	Library	$MODEL_TECH/../altera/vhdl/altera
altera_lnsim	Library	$MODEL_TECH/../altera/vhdl/altera_lnsim
altera_lnsim_ver	Library	$MODEL_TECH/../altera/verilog/altera_lnsim
altera_mf	Library	$MODEL_TECH/../altera/vhdl/altera_mf
altera_mf_ver	Library	$MODEL_TECH/../altera/verilog/altera_mf
altera_ver	Library	$MODEL_TECH/../altera/verilog/altera
cyclone10gx	Library	$MODEL_TECH/../altera/vhdl/cyclone10gx
cyclone10gx_hip	Library	$MODEL_TECH/../altera/vhdl/cyclone10gx_hip
cyclone10gx_hip_ver	Library	$MODEL_TECH/../altera/verilog/cyclone10gx_hip
cyclone10gx_hssi	Library	$MODEL_TECH/../altera/vhdl/cyclone10gx_hssi
cyclone10gx_hssi_ver	Library	$MODEL_TECH/../altera/verilog/cyclone10gx_hssi
cyclone10gx_ver	Library	$MODEL_TECH/../altera/verilog/cyclone10gx
fourteennm	Library	$MODEL_TECH/../altera/vhdl/fourteennm
fourteennm_ct1	Library	$MODEL_TECH/../altera/vhdl/fourteennm_ct1
fourteennm_ct1_ver	Library	$MODEL_TECH/../altera/verilog/fourteennm_ct1
fourteennm_ver	Library	$MODEL_TECH/../altera/verilog/fourteennm
lpm	Library	$MODEL_TECH/../altera/vhdl/220model
lpm_ver	Library	$MODEL_TECH/../altera/verilog/220model
sgate	Library	$MODEL_TECH/../altera/vhdl/sgate
sgate_ver	Library	$MODEL_TECH/../altera/verilog/sgate
sv_std	Library	$MODEL_TECH/../sv_std
twentynm	Library	$MODEL_TECH/../altera/vhdl/twentynm
twentynm_hip	Library	$MODEL_TECH/../altera/vhdl/twentynm_hip
twentynm_hip_ver	Library	$MODEL_TECH/../altera/verilog/twentynm_hip
twentynm_hssi	Library	$MODEL_TECH/../altera/vhdl/twentynm_hssi
twentynm_hssi_ver	Library	$MODEL_TECH/../altera/verilog/twentynm_hssi
twentynm_ver	Library	$MODEL_TECH/../altera/verilog/twentynm
vital2000	Library	$MODEL_TECH/../vital2000
ieee	Library	$MODEL_TECH/../ieee
modelsim_lib	Library	$MODEL_TECH/../modelsim_lib

```
ModelSim> source msim_setup.tcl
```

After the "ld" is finished, you will see the simulation window: drag all the signals from the object window to the Wave window:

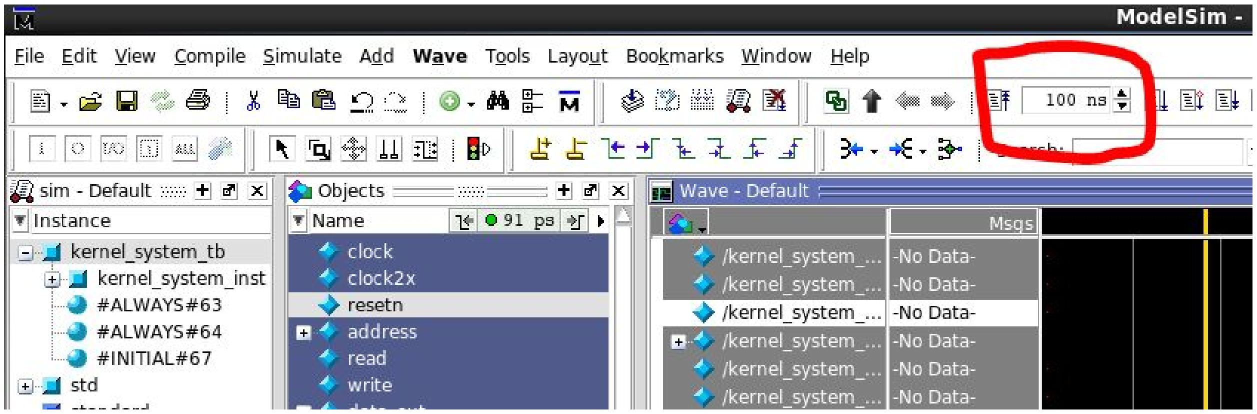


```
# ** Warning: (vsim-3017) ./../../../../ip/kernel_system/kernel_system_mmm_system/mmm_system_140/sim/mmm_system.v(219): [TFMPC] - Too few port connections. Expected 24, found 23.
#    Time: 0 ps  Iteration: 0  Instance: /kernel_system_tb/kernel_system_inst/mmm_system/mmm_system/gmem0_DDR_/t[0]/gmem0_DDR_avm_to_ic File: ./../../../../ip/kernel_system/kernel_system_mmm_system/mmm_system_140/sim/acl_avm_to_ic.v
# ** Warning: (vsim-3722) ./../../../../ip/kernel_system/kernel_system_mmm_system/mmm_system_140/sim/mmm_system.v(219): [TFMPC] - Missing connection for port 'ic_arb_id'.
# ** Warning: (vsim-3017) ./../../../../ip/kernel_system/kernel_system_mmm_system/mmm_system_140/sim/mmm_system.v(219): [TFMPC] - Too few port connections. Expected 24, found 23.
#    Time: 0 ps  Iteration: 0  Instance: /kernel_system_tb/kernel_system_inst/mmm_system/mmm_system/gmem0_DDR_/t[1]/gmem0_DDR_avm_to_ic File: ./../../../../ip/kernel_system/kernel_system_mmm_system/mmm_system_140/sim/acl_avm_to_ic.v
```



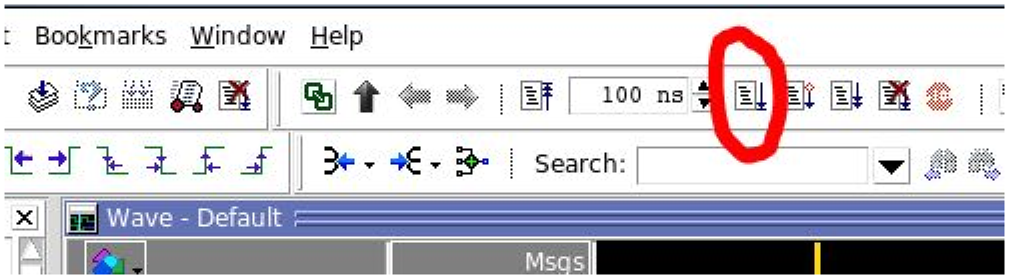
```
add wave -position end  sim:/kernel_system_tb/read
add wave -position end  sim:/kernel_system_tb/write
```

Change the simulation time step to 100ns:

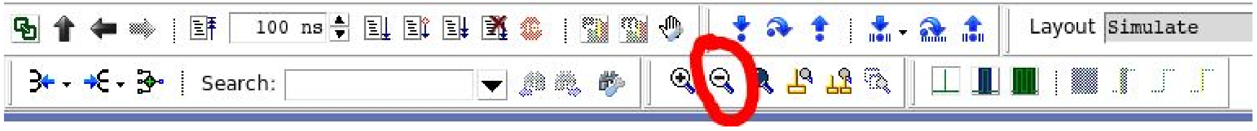


Now launch the simulation by pressing the button to the right of the timestep box:

It may take ten or twenty minutes to run the whole simulation, so feel free to read ahead and familiarise yourself with the rest of the lab material.

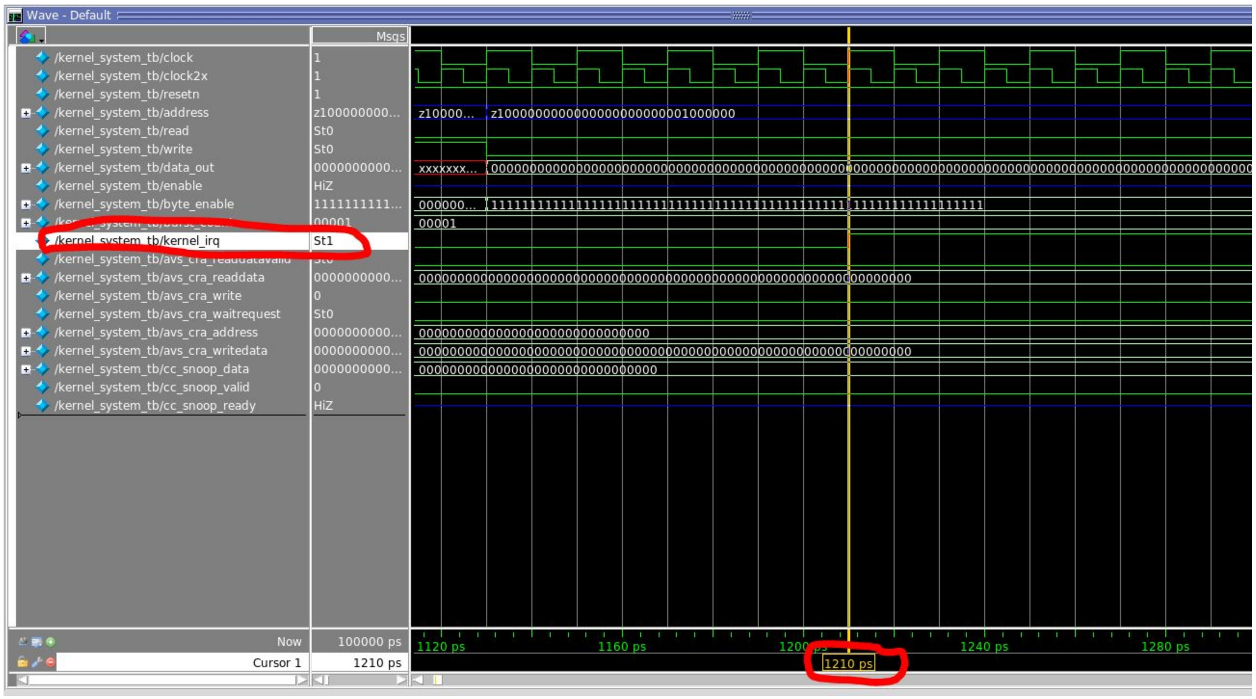


After the simulation is done, zoom out the simulation waveform:



The key signal we are looking for is the kernel_irq signal, when this one turns to high, denotes the kernel has finished the job. By locating the timestamp of the point this one turns to high, we now know the “latency” of this kernel:

Use this information to determine the number of cycles it takes for the kernel to run.



```
Wave - Default
                                          Msgs
/kernel_system_tb/clock                   1
/kernel_system_tb/clock2x                 1
/kernel_system_tb/resetn                  1
/kernel_system_tb/address                 z100000000...
/kernel_system_tb/read                    St0
/kernel_system_tb/write                   St0
/kernel_system_tb/data_out                0000000000...
/kernel_system_tb/enable                  HiZ
/kernel_system_tb/byte_enable             1111111111...
/kernel_system_tb/burst_count             00001
/kernel_system_tb/kernel_irq              St1
/kernel_system_tb/avs_cra_readdatavalid   St0
/kernel_system_tb/avs_cra_readdata        0000000000...
/kernel_system_tb/avs_cra_write           0
/kernel_system_tb/avs_cra_waitrequest     St0
/kernel_system_tb/avs_cra_address         0000000000...
/kernel_system_tb/avs_cra_writedata       0000000000...
/kernel_system_tb/cc_snoop_data           0000000000...
/kernel_system_tb/cc_snoop_valid          0
/kernel_system_tb/cc_snoop_ready          HiZ

Now                                       100000 ps
Cursor 1                                  1210 ps
```

In this testbench, we set the clock cycle as 10ps. So when the kernel finished at 1210ps, it actually runs for 121 cycles which is the latency of the kernel. THIS IS DIFFERENT THAN THE LAST LAB. HERE THE CLOCK CYCLE IS SET AS 10ps.

5. So that is our baseline. Time to improve it! Read the following and then proceed to Part 2

   Think about the temp array The inner compute loop takes data from one half, manipulates it, and stores it in the other half. Then it switches which half it reads from and which half it writes too. By using one array, we are preventing the compiler from making a pipeline stage for each iteration of the outer compute loop.

   In the next part, we will *manually* unroll the outer compute loop and divide our temp array into smaller stage arrays.

Part Two:

1. Navigate to lab2/part2/device and open *mmm.cl* in your preferred editor.
    a. Here, one of the three compute loops are shown. Using the first as a template, write the code for the second and third stages. Remember, data flows from the current stage array to the next one.
    b. Add unroll statements to each loop.
2. Now, go back to the lab2/part2/ directory and run the emulation to insure your code is correct with the command source run_emulation. If you had any errors, go back to your kernel code and debug. The T.A. will be glad to help you with any problems you may be having.
3. When your code runs without errors or wrong answers, give yourself a pat on the back. After that, it's time to see how much your hard work paid off.
    a. Open the report.html again and see if there is a difference in the Loops analysis between this kernel and the kernel from part 1. Pay close attention to II, the Initiation interval. Did it change?
    b. Also take a look at the System viewer section
4. Run the simulation with source run_simulation. Follow the previously-mentioned steps and calculate how many cycles it takes. Calculate the speedup.

Part Three:

1. There is one more important optimisation that assists the compiler in making hardware that compute all the inner compute loop iterations simultaneously. This can be accomplished by adding intermediate values.
2. Navigate to lab2/part3/device and open *mmm.cl*. Here you will find that the first loop has intermediate values c0_r, c0_i, etc. Write the rest of the loops in the same way.
3. Go back to the lab2/part3/ directory, run the emulation and look at the reports. Are there any noticable improvements or differences?
4. Lastly, run the simulation and follow the previously-mentioned steps. Find out how many cycles the kernel requires to produce outputs as well as the speedup.