Digital Systems Design Report 1

Justin Huang
jh4420@ic.ac.uk
Jubo Xu
jx1820@ic.ac.uk

January 27, 2023

B++

Good work + report, but a number of parts are missing + not explained well.

0.1 System Design

Our design consists of an on-chip memory, a Nios II processor core, a JTAG UART interface, an interval timer, a system ID peripheral and a parallel I/O.

0.1.1 On-Chip Memory

In our design, we instantiate one on-chip memory to store both instructions and data. The on-chip memory is initialized to 20 KB and is enlarged later to accommodate for increasing number of processing data.

0.1.2 Nios II Processor Core

Nios II/f is a fast performance, soft processor made out of FPGA resources. This processor supports single-precision floating-point arithmetic operation as required by our design. The instruction cache on the processor is initialized to 2Kbytes. The processor is connected to the on-chip memory via the Avalon bus with the processor as the master and the memory as the slave.

0.1.3 JTAG UART

The JTAG UART interface provides a convenient way to communicate character data with the Nios II processor through the USB-Blaster download cable. The JTAG signal is used for debug purpose.

0.1.4 Interval Timer

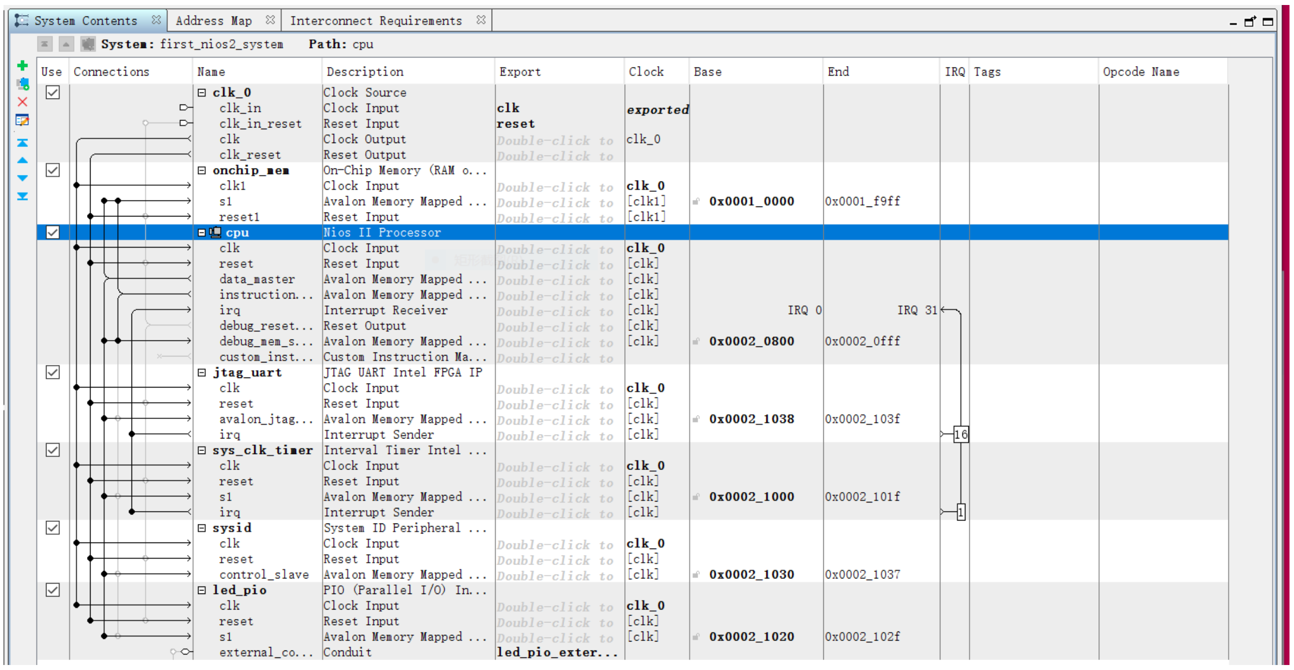
The interval timer provides precise calculation of time and generates system clock tick.

0.1.5 System ID Peripheral

The system ID peripheral prevents accidentally downloading software compiled for a different Nios II system.

0.1.6 PIO

Nios II processor utilizes parallel I/O to receive input stimuli and drive output signals. Our design uses eight PIO signals to drive LEDs on the board.



System Contents | Address Map | Interconnect Requirements

System: first_nios2_system **Path:** cpu

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
✓		**clk_0**	Clock Source							
		clk_in	Clock Input	**clk**	***exported***					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	*Double-click to*	clk_0					
		clk_reset	Reset Output	*Double-click to*						
✓		**onchip_mem**	On-Chip Memory (RAM o...							
		clk1	Clock Input	*Double-click to*	**clk_0**					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk1]	**0x0001_0000**	0x0001_f9ff			
		reset1	Reset Input	*Double-click to*	[clk1]					
✓		**cpu**	Nios II Processor							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		data_master	Avalon Memory Mapped ...	*Double-click to*	[clk]					
		instruction...	Avalon Memory Mapped ...	*Double-click to*	[clk]					
		irq	Interrupt Receiver	*Double-click to*	[clk]	IRQ 0	IRQ 31			
		debug_reset...	Reset Output	*Double-click to*	[clk]					
		debug_mem_s...	Avalon Memory Mapped ...	*Double-click to*	[clk]	**0x0002_0800**	0x0002_0fff			
		custom_inst...	Custom Instruction Ma...	*Double-click to*						
✓		**jtag_uart**	JTAG UART Intel FPGA IP							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		avalon_jtag...	Avalon Memory Mapped ...	*Double-click to*	[clk]	**0x0002_1038**	0x0002_103f			
		irq	Interrupt Sender	*Double-click to*	[clk]			16		
✓		**sys_clk_timer**	Interval Timer Intel ...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	**0x0002_1000**	0x0002_101f			
		irq	Interrupt Sender	*Double-click to*	[clk]			1		
✓		**sysid**	System ID Peripheral ...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		control_slave	Avalon Memory Mapped ...	*Double-click to*	[clk]	**0x0002_1030**	0x0002_1037			
✓		**led_pio**	PIO (Parallel I/O) In...							
		clk	Clock Input	*Double-click to*	**clk_0**					
		reset	Reset Input	*Double-click to*	[clk]					
		s1	Avalon Memory Mapped ...	*Double-click to*	[clk]	**0x0002_1020**	0x0002_102f			
		external_co...	Conduit	**led_pio_exter...**						

Figure 1: Overall system design

0.2 Resource Utilisation and Timing Analysis Results

0.2.1 Memory Allocation

From .map file, it can be seen that .text segment starts at address 0x10244, and the total size is 0x92bc. Text segment is the memory segment for instructions, it contains all the codes from different linked files. Following the .text segment is the .rodata segment, which contains the read only constant variables, and it starts at address 0x19500 with total size of 0x294 bytes. The following memory segment is .rwdata, which contains all the initialized global variables, and it starts at address 0x19794, which is equal to 0x19500+0x294, and the total size is 0x2e4 bytes. The next memory segment is .bss segment, which starts at address 0x19d5c with size of 0x128 bytes. Bss segment contains the uninitialized global variable. The other parts of the onchip memory contains the stack, and it can also be seen from the linker scripts that we don't have memory allocation for heap here since we are not using any dynamic memory allocation. The only thing we changed in our code is the array x, which is located in stack, so increase the size of this array in different test cases would increase the general memory footprint but will not change the memory size of text+data+bss, and that's why the memory size of these segments is not changed in map file under the three test cases.

nice elaboration

✓

Therefore, to estimate the lower bound of how much memory we need for the on-chip memory, we can add the memory size of text+data+bss segments, which is around 39260 bytes, with the memory size of the array. Since we are using single precision floating point representation here, so we need 32 bits which is 4 bytes for each number, and the total size of the array is just how many elements we have inside the array times 4 bytes. Therefore, the memory size we need for the array is $52 \times 4 = 208$ bytes, $2041 \times 4 = 8164$ bytes, and $261121 \times 4 = 1044484$ bytes for the three cases respectively. And that's why the execution fails for our first initialized on-chip memory with size of 2048 bytes because its much smaller than the lower bound. In our test, we define our on-chip memory size as 64000 bytes. By increasing the on-chip memory size, we can successfully run the case 1 and case 2, but we cannot run case 3. This is because the lower bound of memory we need for case 3 is $39260 + 1044484 = 1083744$ bytes, but we only have 397 M10Kb memory blocks in our board, which means the maximum memory size we can have for our on-chip memory is 508160 bytes, which is always less than 1083744 bytes, so we cannot run the case 3 if we just use the on-chip memory. To solve this problem, for example, we could still use the array with smaller size and set a loop for it to go through the generateVector() and sumVector() again and again, and each time after these two operations, we replace the first element of array to be the last element of that array plus the step, and re-go though the process. This can solve the memory shortage problem but will increase the latency.

✓

✓

0.3 System performance

It can be seen that we need 1 tick for the first case and 28 ticks for the second case, this is because for the first case, the array size is 52, which is much smaller than that of the second case, which means that the second needs more instruction cycles to operate, and that's why the second case takes more time. If we make the array size of two cases be the same, for example, both 2041, then the latency for step equals 5 is larger than the latency for step equals 1/8. In this case, the instruction cycles are the same, but since the number of step 5 is larger than the number of step 1/8, then more bits need to be manipulated for case 1, and therefore the central path for carrier of adder will be longer, and then it would take more time to run. To speed up the performance, we cannot do too much about the algorithm in this case, but we could modify the code and the compiler configuration. For example, considering about the for loop inside sumVector(), instead of letting i increases from 0 to N, we could make it starts from N and decreases until it equals 0, such kind of manipulation could eliminate the compare instruction in each loop, which could reduce the number of instructions a little bit.

This does not make sense. The size of the operands is fixed for your system, and the instruction take the same clock cycles regardless on the numbers they operate on.

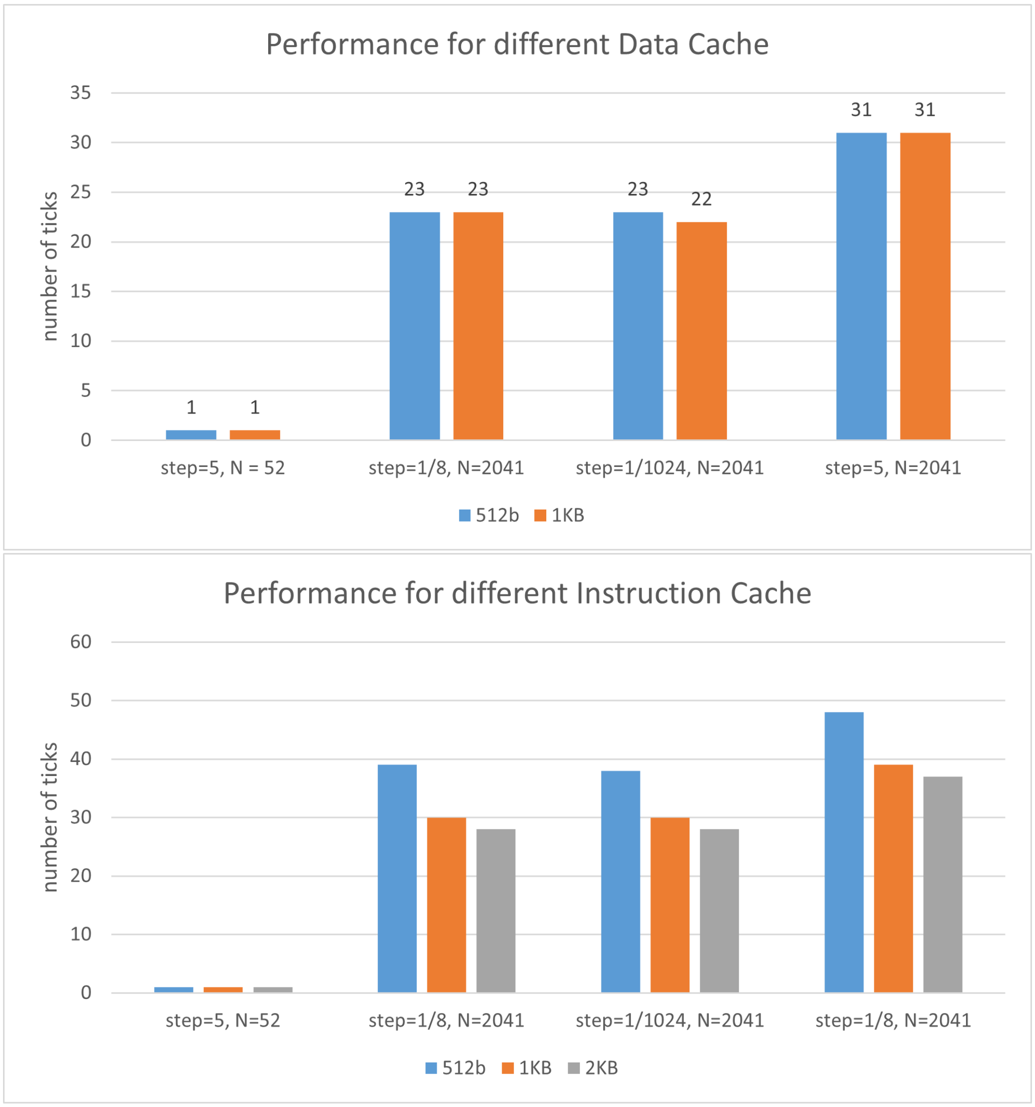
→ This is not in the focus of the work.

flow.

0.4 The Effects of Cache

Caches are used in a processor to hide memory access latencies and improve the overall system performance for the given application. This is because the targeted program performs a sum of vector product operation. This program involves the use of for loop structures, and the instructions present good temporal locality. The amount of instruction reuse depends on the number of iterations N in the application. The boost in performance is not evident for small number of iterations in test case 1 ($N = 52$) because the amount of instruction reuse is limited. The boost in performance is significant for a large number of iterations in test case 2/4/5 ($N = 2041$). Given the same benchmark ($N = 2041, step = 5$) and data cache size, the execution time for the $2KB$ instruction cache is 37 clock ticks and the execution time for the $512B$ instruction cache is 48 clock ticks. The reusable instructions can be stored in a larger instruction cache without being evicted to accommodate for incoming instructions, which improves the system performance significantly.

According to our measurement, increase the size of data cache has no impact on the system performance for the given application. This is because in each iteration, each vector element $x[i]$ is used only one time and is not being used later in the program. The data used by the program present poor temporal locality. Given the same benchmark ($N = 2041, step = 5$) and instruction cache size, the execution time for both the 1KB data cache and the $512B$ data cache is 31 clock ticks. The vector element can be evicted immediately from data cache after its square is added to the sum. If the same element is accessed repetitively in the program, the effect of increasing data cache may be more evident. ✓



✓

✓

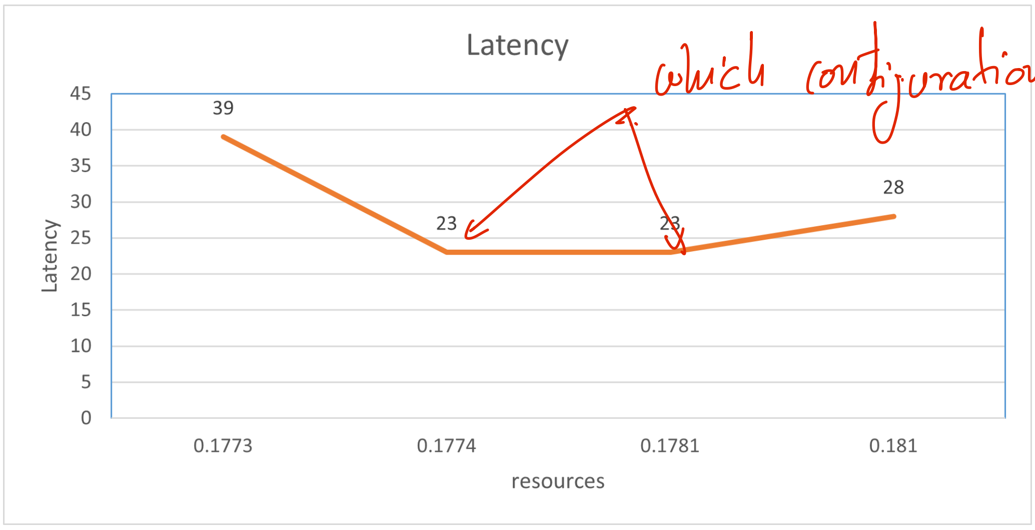
Figure 2: Effects of Data-Cache and Instruction-Cache

0.5 Compiler Flags

For compiler flags, we mainly consider about APP_CFLAGS_OPTIMIZATION, which is default as O0 in generated Makefile. We can change that to Os, O1, or O2. Os is optimized for space so it will generate smaller binary, but not the fastest one. After changing O0 to Os, the operation time didn't change a lot, but the memory utilization decreases a little bit, which is expected.

Resources	Logic Use	Total Reg	Data Cache	Instruction Cache	On-chip memory	Worst-Slack setup	Worst-Slack hold
0.1773	1384	2169	2K Bytes	512 bits	64000	9.798	0.024
0.1774	1384	2139	512 bits	2K Bytes	64000	10.956	0.019
0.1781	1375	2127	1K Bytes	2K Bytes	64000	10.267	0.009
0.1810	1400	2170	2K Bytes	2K Bytes	64000	10.833	0.006

Figure 3: Resources Table



which configurations are these?

✓

Figure 4: Latency vs Resources

∘ accuracy vs Matlab?

- Conclusions?