Study of NiosII Processor

Kalyani Jayaprakash
Embedded Systems
KTH Royal Institute of Technology
Stockholm, Sweden
Email: kalyani@kth.se

Rucha Thorat
Embedded Systems
KTH Royal Institute of Technology
Stockholm, Sweden
Email: thorat@kth.se

***Abstract*—This paper illustrates details about the architecture of the NiOS II multiprocessor, stating it's features and limitations. The main focus is on the interconnect of processors and peripherals and communication mechanism. It also focuses on types and sizes of memories. Based on the study of architecture, this paper suggests the safe mechanism for shared memory and message passing comuunications.**
Analysis of given source codes and scripts is carried out and certain observations like, possible optimizations, memory footprint have been discussed.
The demo application has been designed and developed, to use shared memory and message passing communication for interprocessor communication.

I. Introduction

A Nios II processor system is equivalent to a microcontroller or computer on a chip that includes a processor and a combination of peripherals and memory on a single chip. It is a configurable soft IP core, as opposed to a fixed, on-the-shelf microcontroller. Soft means the processor core is not fixed in silicon and can be targeted to any Altera FPGA family. You can add or remove features on a system-by-system basis to meet performance or price goals.

Features:

The Nios II processor is a general-purpose RISC processor core consisting of a Nios II processor core, a set of on-chip peripherals, on-chip memory, and interfaces to on- chip memory, all implemented on a single Altera device. Like a microcontroller family, all Nios II processor systems use a consistent instruction set and programming model. Nios II processor system consumes only 5% of a large Altera FPGA, leaving the rest of the chips resources available to implement other functions. By using custom instructions, the system designers can fine-tune the system hardware to meet performance goals and also the designer can easily handle the instruction as a macro in C. Optional JTAG debug module enhancements, including hardware breakpoints, data triggers, and real-time trace. Separate instruction and data caches (configurable from 512 bytes to 64 KB). Access to up to 4 GB of external address space[1].

Limitations:

Processing speed is limited by the speed of the fabric. Large data processing is difficult due to small memory and cache compared to hard processors.

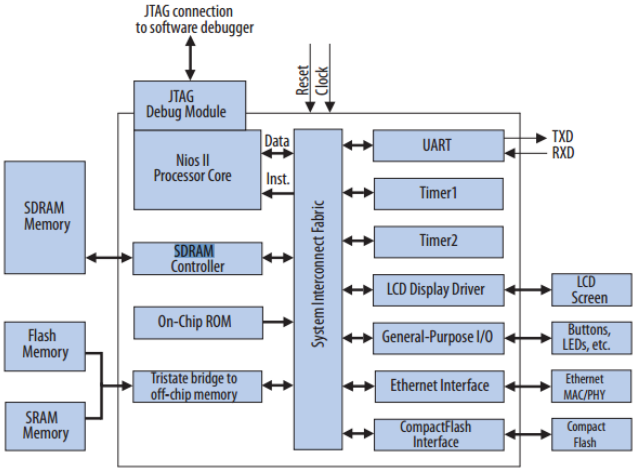

Fig. 1. Nios II Processor System

II. Architecture of NiOS II

The Nios II architecture describes an instruction set architecture (ISA). The ISA in turn necessitates a set of functional units that implement the instructions. A Nios II processor core is a hardware design that implements the Nios II instruction set and supports certain functional. The processor core does not include peripherals or the connection logic to the outside world. It includes only the circuits required to implement the Nios II architecture.

The Nios II architecture supports separate instruction and data buses, classifying it as a Harvard architecture. Both the instruction and data buses are implemented as Avalon-MM master ports that adhere to the Avalon-MM interface specification. The data master port connects to both memory and peripheral components, while the instruction master port connects only to memory components. The Nios II data bus is implemented as a 32-bit Avalon-MM master port. The data master port performs two functions: 1. Read data from memory or a peripheral when the processor executes a load instruction 2. Write data to memory or a peripheral when the processor executes a store instruction[2][3]

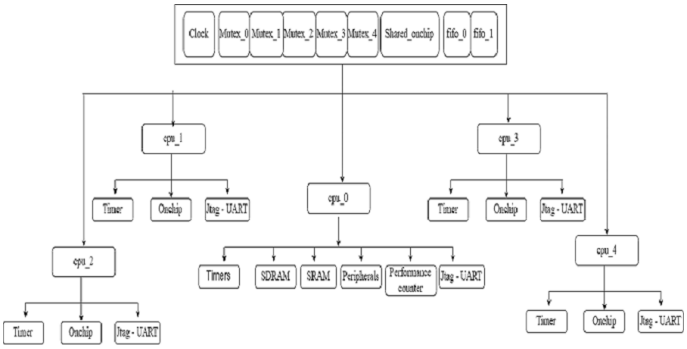

Fig. 2. Nios II Interconnection network

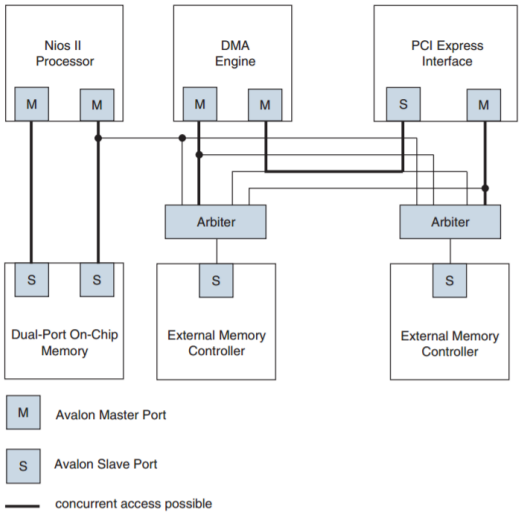

Fig. 3. Avalon-MM interconnection fabric

A. Interconnection of cores and pheriperals

The Nios II architecture does not specify anything about the existence of memory and peripherals; the quantity, type, and connection of memory and peripherals are system-dependent. Typically, Nios II processor systems contain a mix of fast on-chip memory and slower on-chip memory. Peripherals typically reside on-chip, although interfaces to on-chip peripherals also exist.

The Nios II architecture is a load-store architecture. Load and store instructions handle all data movement between registers, memory, and peripherals. The Nios II architecture provides memory-mapped I/O access. Both data memory and peripherals are mapped into the address space of the data master port.

Usually the instruction and data master ports share a single memory that contains both instructions and data. While the processor core has separate instruction and data buses, the overall Nios II processor system might present a single, shared instruction/data bus to the outside world. The outside view of the Nios II processor system depends on the memory and peripherals in the system and the structure of the system interconnect fabric. The data and instruction master ports never cause a gridlock condition in which one port starves the other. For highest performance, assign the data master port higher arbitration priority on any memory that is shared by both instruction and data master ports.

B. Analysis of interconnection based on given .qsys

We analysed the given qsys file and the interconnection network generated. Our observations are as follows: The given architecture corresponds to a Nios II multiprocessor with five CPUs.The CPUs are connected to the shared onchip memory and CPU 0 has access to other pheriperals. All these interconnections are achieved via avalon interconnect fabric and the communication between the shared memory and CPUs are secured using mutex logic. There are a total of four mutex available in the architecture and two fifo buffers which is shared between all CPUs.

Shared onchip Memory: This memory can be configured as either RAM or ROM. In the given architecture, it is configured as RAM with a total memory size of 8192 bytes. The data width is 32 bytes.

FIFO: There are two fifo buffers of depth 16 available for the CPUs. Each with a data size of 32 bytes capable of Avalon MM- Read/Write.

SDRAM: The data master of CPU 0 is connected to a SDRAM with a memory size of 64M bits and data width of 32 bytes.

Pheriperals: Every CPU has inernal timer pheriperals and only CPU 0 is connected to external pheriperals like buttons, leds, switches and seven segment display.

If any of the processor has to communicate to external harware pheriperals, it can be achieved through CPU 0 because only CPU 0 can acess pheriperals.Each CPU has a data master and the data master port is an Avalon-MM master port that connects to data memory and peripherals via system interconnect fabric.

Avalon-MM interconnection fabric:

SOPC Builder generates system interconnect fabric with slave side arbitration, every master in your system can issue transfers concurrently. As long as two or more masters are not posting transfers to a single slave, no master stalls. The system interconnect fabric contains the arbitration logic that determines wait states and drives the waitrequest signal to the master when a slave must stall. Figure 3 illustrates a system with three masters. The bold wires in this figure indicate connections that can be active simultaneously.[4]

C. Safe mechanism for shared memory and message passing communication

THe shared memory is accessed by all five CPUs and therefore there is an urge to secure the data written in by any of the CPU and prevent overwriting by another. This is achieved through mutex. The data in the shared memory is accessed by CPUs by locking the mutex when it is reading or writing so that simultaneously another CPU won't access it. It will have to wait till it receives the lock. After accessing the

memory CPU will make the mutex available for other CPUs by unlocking it.

For message passing communication also mutex is a safer way to read and write from the queue. Another possible option is the use of existing fifo buffers. The read and write buffer pointers will prevent data corruption to an extend.

III. Code optimization

Code size is always a concern for embedded systems developers, because there is a cost associated with the memory device that stores code. The ability to control and reduce code size is important in controlling this cost.Code size plays an important role in controlling the cost of memory device that stores the code. The HAL environment is designed to include only those features that we request, minimizing the code footprint. From the given scripts, we studies and analysed the optimizations used. We rechecked if they are really required, and also, what other optimizations can possibly be added.

Existing optimizations in given script:

hal.enable_reduced_device_drivers: The HAL defines a C preprocessor macro named ALT_USE_SMALL_DRIVERS that it can be used in driver source code to provide alternate behavior for systems that require a minimal code footprint. You can enable ALT_USE_SMALL_DRIVERS in a BSP with the hal.enable_reduced_device_drivers BSP setting.

hal.enable_lightweight_device_driver_api: With lightweight drivers turned on, printf() calls the HAL write() function, which directly calls your drivers _write() function, bypassing file descriptors.

hal.enable_small_c_library: Causes the small newlib (C library) to be used. This reduces code and data footprint at the expense of reduced functionality. Several newlib features are removed such as floating-point support in printf(), stdin input routines, and buffered I/O.

hal.enable_sopc_sysid_check: The system ID check is enabled in the creation of command-line arguments to download an .elf file to the target. With the system ID check disabled, the Nios II EDS tools do not automatically ensure that the application ..elf file (and BSP it is linked against) corresponds to the hardware design on the target.

hal.max_file_descriptors: Determines the number of file descriptors statically allocated.

hal.make.bsp_cflags_optimization: C/C++ compiler optimization level. -O0 = no optimization, -O2 = normal optimization, etc. -O0 is recommended for code that you want to debug since compiler optimization can remove variables and produce nonsequential execution of code while debugging.

hal.enable_exit: This option increases code footprint if your main() routine returns or calls exit(). If set to false, reduces footprint.

hal.enable_c_plus_plus: Enable support for a subset of the C++ language. This option increases code footprint by adding support for C++ constructors. If set to false, reduces code footprint.

New or changed optimizations:

With original source code					
Filename	**text**	**data**	**bss**	**dec**	**hex**
hello_mpsoc_0.elf	3872	580	424	4876	130C
hello_mpsoc_1.elf	1816	328	16	2160	870
hello_mpsoc_2.elf	1816	328	16	2160	870
hello_mpsoc_3.elf	1816	328	16	2160	870
hello_mpsoc_4.elf	1240	96	16	1352	548

With Optimization: Replace printf() with alt_printf()					
Filename	**text**	**data**	**bss**	**dec**	**hex**
hello_mpsoc_0.elf	3680	580	424	4684	124c
hello_mpsoc_1.elf	1624	96	16	1736	6c8
hello_mpsoc_2.elf	1624	96	16	1736	6c8
hello_mpsoc_3.elf	1624	96	16	1736	6c8
hello_mpsoc_4.elf	1240	96	16	1352	548

Fig. 4. Memory footprint for compiled source codes - hello_mpsoc

1. Use alt_printf() instead of printf() : alt_printf() This function is similar to printf(), which takes up substantially less code space than printf(), regardless whether you select the lightweight device driver API. alt_printf() occupies less than 1 KBKB with compiler optimization level -O2

2. hal.enable_lightweight_device_driver_api is true, hence, there are no file descriptors, so the setting hal.max_file_descriptors will be ignored and hence, can be removed.[5]

Other than these, the focus should be initially on code optimization, by applying techniques such as loop unrolling, inline functions. While using HAL optimizations, elimination of unused drivers should be done, such that, if the hardware includes a device that the respective program never uses, then the hardware can be removed from the device. This reduces both code footprint and FPGA resource usage.

IV. Memory footprint

We analysed the code size of compiled codes using command 'nios2-elf-size'. The observations are given in figure .

V. Demo Application

The application that we have developed does the basic functionality of a calculator which performs addition, subtraction, multiplication and division. The input is given via switches or push buttons from the DE2 board which is read by CPU 0 and store the respective inputs in the shared onchip memory. According to the application chosen by the user the other CPUs will perform the task. For example: if the user has requested for addition, CPU 0 locks the mutex to store the input in shared memory. This mutex can be accessed only by CPU 0 and CPU 1. After CPU 1 performs addition,it writes back to the shared memory and unlocks the mutex. CPU 0 then will display the result using seven segment displays or leds.

Using shared memory				
Section	%	Time(usec)	Time(clocks)	Occurences
Communication through shared memory	75	6710950	335547513	1
alt_putstr(frequently used)	0	313	15688	1
Total time: 8908117 usec (445405864 clock-cycles)				

Fig. 5. Communication using shared memory

Using FIFO				
Section	%	Time(usec)	Time(clocks)	Occurences
Communication through fifo	59	17969647	898482347	1
alt_putstr(freqently used)	0	313	15688	1
Total time taken for execution: 30197713 usec (1509885663 clock-cycles)				

Fig. 6. Communication using fifo

The performance comparison of reading and writing data using shared memory and fifo is compared. The code is divided into subsection for this comparison and the compared result is shown in Figure 5 and Figure 6. For the entire communication starting from writing, performing the desired task and then reading back, use of shared memory shows better performance than using fifo.

VI. Conclusion

The architecture and interconnection of Altera Nios II processor was analysed and various optimization techniques were studied. Considering all these features, demo application will be developed.

References

[1] Nios II Processor features, https://www.altera.com/products/processors/features.html

[2] Nios II Architecture, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf

[3] Nios II classic software developer's handbook, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2sw_nii5v2.pdf

[4] Avalon Memory Mapped design optimizations, https://www.altera.com/zh_CN/pdfs/literature/hb/nios2/edh_ed51007.pdf

[5] Developing programs using the Hardware Abstraction Layers, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2sw_nii52004.pdf