A Function Level Co-Design Tool: CoDeNios

ABSTRACT

The need of co-design systems, along with the FPGA complexity, is increasing dramatically, both in industrial and academic settings. New tools are necessary to ease the development of such systems. Altera supplies a development kit with a 200'000 equivalent gates FPGA; combined with its proprietary Nios configurable processor, it allows co-design and multi-processor architecture creation. In this paper, we present a new tool, CoDeNios, which lets a developer partition a C program at the function level, and automatically generates the whole system.

Categories and Subject Descriptors

B.6 [**Hardware**]: Logic Design; B.6.3 [**Logic Design**]: Design Aids—*Automatic synthesis*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.0 [**Computer Systems Organization**]: Hardware/software interfaces

General Terms

Design

Keywords

Co-design, FPGA, Nios processor

1. INTRODUCTION

Until recently, co-design[4] was limited to complex industrial projects. The high cost of such systems did not allow academic projects to use co-design. Now, with the development of Field Programable Gate Arrays (FPGAs), the conception of such systems is easier. The reprogrammable capability of FPGAs permits prototyping at a low cost, which is very important for universities and industries. The problem now is the lack of tools aiding development of these systems. With this aim in view, Altera supplies the Nios processor family. This soft IP core is a configurable RISC processor which can be used in any design.

In this paper we present CoDeNios (**CO-DE**sign with a **NIOS**), a new tool based on a Nios processor, which helps a developer make a hardware/software partition[3] of a C program. This partition is made at the function call level. For each function declared like `void fname(...)`, the user can force it to be calculated either by the main processor, by a slave processor, or by a hardware module. For the last case, the developer has to write a VHDL file to define the function behavior. Apart from this human intervention, the whole interface between hardware and software is automatically generated (C and VHDL files).

Contrarily to other systems like COSYMA [2], which automatically makes a partition, our software lets the user choose it. This particularity allows the developer to test any hardware module by automatically interfacing it to a processor. It is also useful for academic courses, where students can do the partition themselves, and evaluate their work. P. Chou, R. Ortega and G. Borriello [1] have created a system to synthesise a hardware/software interface for a micro-controller. Their work is made for peripherals present outside the chip which contains the controller. With our tool, the processor and its user-defined peripherals are implemented in the same chip. Thus, CoDeNios is better suited for system prototyping and hardware module evaluation.

This paper is structured as follows: Section 2 describes the APEX20K® FPGA family supplied by Altera^{TM} and the Nios processor used by CoDeNios. Section 3 focuses on CoDeNios itself, explaining its possibilities, while section 4 explores the performances of a design generated by our application. Finally section 5 concludes by discussing current and future work.

2. APEX20K FAMILY AND NIOS

Altera, with the APEX20K family, offers FPGAs with densities ranging from 30'000 to over 1.5 million gates. It is built for system-on-a-programmable-chip (SOPC) designs, with embedded system blocks used to implement memories as dual-port RAM, ROM, CAM, etc. This family is divided into 3 different types:

- APEX20K : Aluminium, 0.22- μm, 6-layer metal
- APEX20KE: Aluminium, 0.18- μm, 8-layer metal
- APEX20KC: All-layer cooper, 0.15- μm, 8-layer metal

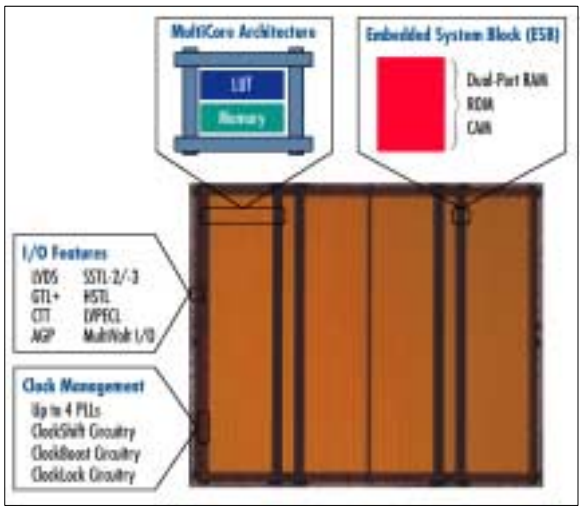

Figure 1: APEX Device Features

Table 1: Elements of the APEX20K family

Component	Typical gates	Maximum RAM bits
EP20K30E	30'000	24'576
EP20K60E	60'000	32'768
EP20K100E	100'000	53'248
EP20K200E	200'000	106'496
EP20K400E	400'000	212'992
EP20K1000E	1'000'000	327'680
EP20K1500E	1'500'000	442'368

For our application, we use a development board with an APEX20K200E, from the APEX20K family (cf. table 1 and figure 1). This FPGA contains 106'496 configurable memory bits, and 200'000 equivalent gates, which is enough to implement a 3-processor design.

Along with these new FPGAs which allow SOPC designs, Altera supplies a new processor. The Nios (cf. table 2) is a configurable RISC processor, working with 16 or 32 bits (instruction and data). A wizard helps create a Nios with all the necessary parameters.The size of instructions, as well as the number of registers, is decided by the user. A multiplication unit can be added to speed up multiplications, with a cost in term of gates. The most interesting possibility is the ability to add as many peripherals as needed. Many of them are already supplied by the wizard: memory interfaces for ROM or RAM, UART to manage a serial COM, IDE controller, timer, etc. All these peripherals are memory mapped for the processor. User-defined peripherals can also be added, by specifying the address range, an optional interrupt number and the number of clock cycles for write and read operation. When all the processor parameters are set, a VHDL entity is generated, which can be included in any design.

As CoDeNios supports a multi-processor architecture, we chose a 16 bit Nios, so as to allow a maximum of processors in a design. One single special peripheral was added, which contains all hardware and slave processor calculated functions. It has an address range of 2, used to access a counter (1 address for a 32 bit counter accessible in 2 read cycles) and to define a protocol for calling functions and pass parameters.

Table 2: Nios processor characteristics

Feature	Description
type	RISC
pipeline	4 levels (5 for load/store)
instructions and data size	16 or 32 bits
number of registers	128, 256 or 512
frequency	< 50 MHz
place	approximately 26'000 bits for the 16 bits version

3. CODENIOS

The hardware/software partitioning of a task aims to accelerate it, by taking advantage of hardware speed. An important issue is therefore to be able to find bottlenecks where hardware can speed up a system. Then the new solution needs to be evaluated in order to prove it is better than the original software execution. Currently there is no theory to calculate precisely the execution time of a co-design system, so many experiments and measures have to be run.

A second co-design problem is the interface between hardware and software. For each new hardware module connected to a processor a protocol has to be defined. The conception of this part of a system can be very time-consuming, so automating this task would be a great advantage for a developer.

CoDeNios proposes to solve both problems. This tool, based on the Nios processor described above, has a graphical user interface which enables a developer to make a partition of a C program, at the function level, simply by click, drag and drop operations. This partition allows a function to be calculated by the main processor, by a slave, or by a hardware module. Once the choices are validated, an interface between the different processors and the hardware modules is generated in the form of VHDL and C files. The original C code of the main processor is transformed to call slave modules, while for a slave Nios, the whole C code is generated. For the hardware, the whole system is generated, except the architecture of hardware modules. For them, a template is generated, letting the developer describe the function behavior.

3.1 Function Selection

At the beginning of a project, the developer writes a C program for a 16 bit Nios. The C file can be opened with CoDeNios. A graphical user interface (GUI), as shown in figure 3, lists all functions returning `void`[1] in a rectangle representing the main processor. It is then possible to drag and drop a function outside this rectangle to make it a hardware module. By clicking on it, a hardware module can be turned into a slave processor, and vice versa. For both entities, all input and output parameters are listed, connected by an arrow. For a parameter passed in C by reference (`int *a`),

[1]This limitation will be reduced, by also allowing functions returning an integer.

the direction (input, output, input-output) can be changed by the user, by clicking on the arrow. The value or the reference can be sent to the slave module, allows the use of pointers to access a shared memory.

When the whole system is configured correctly, buttons on the GUI can launch VHDL and C file generation, hardware synthesis, placement and routing, C compilation, and finally start up the execution of the program on the board, assuming the FPGA is configured. This command sends all different executable codes for every processor on-chip. Then, with a terminal, CoDenios installs a communication between the main processor and the user, who can view `printf()` results and type characters which are sent to the FPGA.

3.2 Automatic Interface Generation

As explained above, CoDeNios generates VHDL files implementing a protocol between all processors and hardware modules. For a Nios-to-Nios communication, no intervention of the user is required, whereas he has to write VHDL for a Nios-to-hardware communication. In this last case, a template is generated, declaring the entity and implementing a small state machine. The state machine corresponds to the protocol the developer has to respect. First, every input and output parameter of the function is declared as ports. For an output parameter, an additional port, called `load_x` (where `x` is the name of the parameter), is used to load the result value in a register outside the entity. An input signal called `start` goes to '1' for one clock cycle, indicating that the input parameters are loaded, and that the entity can start the calculation. An output signal called `done` has to be put at '1' during one clock cycle to inform an external controller that all output parameters are loaded, and that the calculation is over.

As an example, the Greatest Common Divider (GCD) function is declared like this: `void gcd(int a,int b,int *c)`. Figure 2 shows the template generated, which implements a state machine waiting for the `start` signal to be '1'. When this event occurs, it loads the value 0 in the output register of `c` and sets `done` to '1' to signify the treatment is finished. From this template, the developer only needs to change the architecture, or to map an existing VHDL file into the architecture.

3.3 Parallelism

Regarding the C files, each original function which is chosen to be calculated by a slave (processor or hardware) is replaced by two new calls, one to start the function calculation, and one to wait for its termination. Continuing with the GCD example, `gcd(a,b,&c)` will be replaced by:

```
hcall_gcd(a,b,&c);hwait();
```

`hcall_gcd()` launches the new hardware function calculation, and `hwait()` waits for its termination and retrieves the output parameters. This call/termination splitting allows us to take advantage of the hardware parallelism. It is possible to call several independent[2] functions and then to

[2]Two functions are said to be independent if they are called consecutively, and no output parameters of the first are input of the second.

```
library ieee; use ieee.std_logic_1164.all;

entity gcd is port (
    a_in: in std_logic_vector(15 downto 0); -- input parameter
    b_in: in std_logic_vector(15 downto 0); -- input parameter
    c_out: out std_logic_vector(15 downto 0);-- output parameter
    load_c: out std_logic;              -- put it at '1' to
                                -- load the output
                                -- parameter

    clk:    in std_logic;  -- clock signal
    rst:    in std_logic;  -- reset, '0' active
    start:  in std_logic;  -- '1' during one clock cycle to
                           -- begin the treatment
    done:   out std_logic  -- put it at '1' during one clock
                           -- cycle when the treatment is
                           -- finished

);
end gcd;

architecture struct of gcd is

    type state_type is (s0,s1);
    signal state,n_state: state_type;

begin


    process(state,start)
    begin

        -- default output values
        done<='0';
        c_out<=(others=>'0');
        load_c<='0';
        n_state<=state;

        case state is
        when s0=>       -- wait for start
            if start='1' then
                n_state<=s1;
            end if;
        when s1=>       -- treatment finished
            done<='1';
            load_c<='1';
            n_state<=s0;
        end case;
    end process;

    process(rst,clk)
    begin
        if rst='0' then
            state<=s0;
        elsif clk'event and clk='1' then
            state<=n_state;
        end if;
    end process;

end struct;
```

Figure 2: Generated VHDL file for GCD function

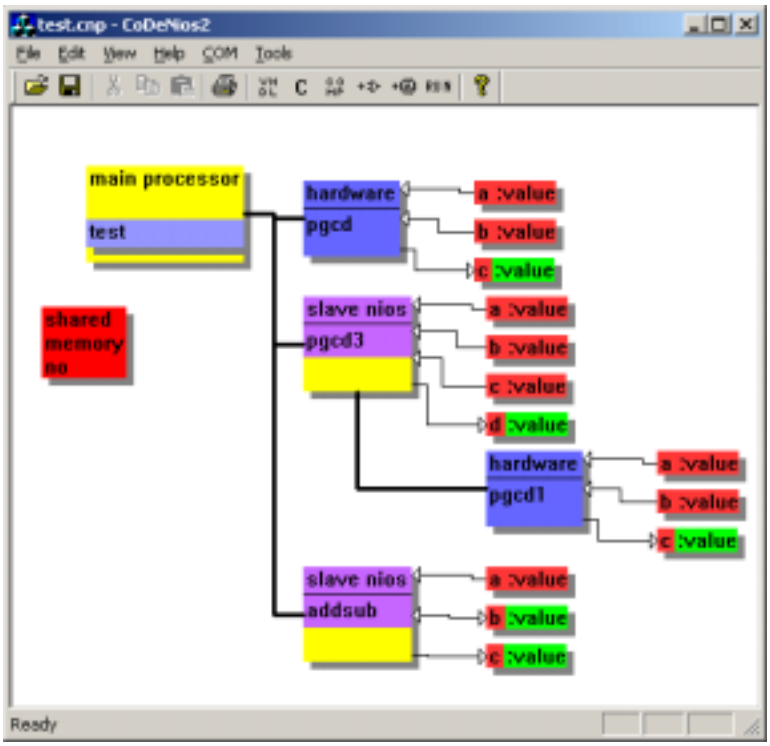

Figure 3: CoDeNios graphical user interface

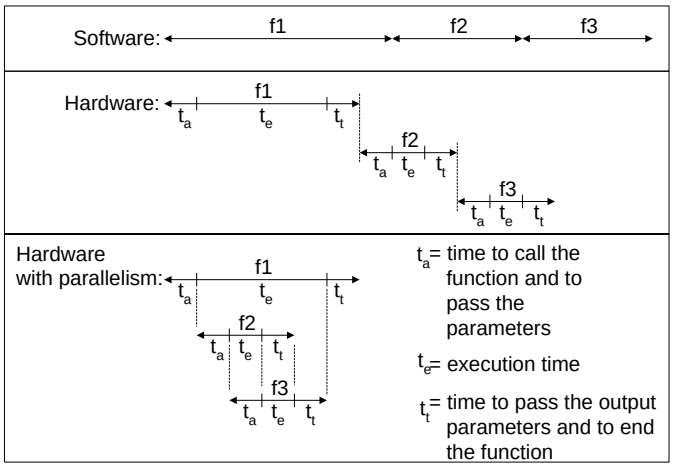

Figure 4: 3 types of executions

wait until they are all finished. By calling the most time-consuming functions first, the total execution time can be dramatically reduced (cf. figure 4).

3.4 Execution Time Evaluation

As presented above, one important aspect of CoDeNios is its capacity to evaluate the execution time of a hardware orsoftware function. With this aim in view, some counters are automatically placed in the system. One global 32bit counter is directly accessible by the main processor. It is set to 0 with a soft reset of the FPGA, and counts the clock cycles. It makes it possible to evaluate the total time of different (parallel or not) function calls. A counter is attached to each co-design module, in order to evaluate the real number of clock cycles of a function execution. It does not take into account the time to pass parameters and to call the function. Its value is retrieved by the master after the output parameters.

The global counter value is accessible via a function `void GetTime(time_t *t)` and the module counters are accessible by `void GetFuncTime(int FUNCID,time_t *t)`. They are declared in an automatically generated file which contains all the procedures responsible for the co-design function calls.

3.5 Memories

As multi-processor architectures are possible with CoDeNios, several memories are used. The main processor places its executable code in the onboard SRAM of 1MB. The slaves each use only one on-chip RAM of 1KB. This limitation is due to the number of embedded system blocks[3] of the APEX20KE200 (52 blocks of 2048 bits). A larger RAM for each would have prevented having 3 processors on-chip. A shared memory of 1KB can be added automatically in order to pass arrays to co-design functions (by passing a pointer). It is shared between the main processor and all co-design modules. To manage this RAM, a simple arbitration is implemented, giving a different priority to each module.

4. PERFORMANCE

[3]The embedded system blocks are used to implement memories.

The performance of a design made with CoDeNios depends on the hardware implementation written by the user for the hardware functions. The total execution time depends on the parameter passing time , the calling time, and the hardware calculation time. The parameter passing time is very small; a write instruction for an input parameter, and a read one for an output. On the other hand, to call and then to wait for a function costs 113 clock cycles. Because of this, the efficiency of the hardware user-defined modules is very important. One single addition would be slower by hardware, the latency of 113 clock cycles being too long, whereas a mathematical series calculation could be more efficient in hardware. Note that for an industrial purpose these 113 clock cycles could be reduced, by changing the generated C code. Currently, this code is split into different functions (one to call, and one to wait). As a software function call costs time, by putting all operations inline we could gain a lot of time. This has not been done yet, because of the C code clarity, which is important for student projects. Another way to save time would be not to allow exact calculation of hardware function execution time. In the current version, this value is retrieved after the function termination. By deleting it, 4 clock cycles could be spared, but, because they allow the developer to evaluate the software solution as well as the hardware one, this deletion was not done.

Finally, the performance of a system depends on the parallelism imposed by the developer. If more than one function can be launched at the same time, the execution time can be dramatically reduced.

5. CONCLUSION

In this paper we presented a co-design tool called CoDeNios. This pedagogic tool helps a developer make a hardware/software partition of a C program, and generates the interface between the hardware and the software. A multiprocessor architecture is also possible, sparing the user the task of interfacing the different processors.

CoDeNios, in its present state, can be used as a teaching tool. The students can rapidly test hardware modules by integrating them in a co-design system, without having to develop a protocol to synchronize the hardware and the software. To evaluate the efficiency of their hardware modules, C functions permit to retrieve counters values. Therefore it is possible to compare a software solution with a hardware one.

In addition to the educational function of CoDeNios, an industrial use is possible. Having the possibility to create a complete system mixing hardware and software implies a small development time. To make this even easier, a tool to generate VHDL from C functions is currently being developed in our lab. It will be able to transform a subset of C (if, for, while, +, -, *, /) calculating with 16 bit integers into a hardware pipeline. Integrated with CoDeNios, it will complete the automation of the system generation. The development process will also be totally automated based on the user choices.

Finally, besides the C to VHDL translation, we will add new possibilities to CoDeNios. First, functions which return an integer will be potential slave calculated functions. For instance, a developer will be allowed to use a co-design function in a conditional statement, or in an expression. Second, the function parameter size is currently fixed to 16 bits. This limitation will be removed, allowing different types of data to be sent to a co-design module.

6. REFERENCES

[1] P. Chou, R. Ortega, and G. Boriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *Proceedings of the International Conference on Computer Aided Design*, pages 488–495, Los Alamitos, California, 1992. IEEE Computer Society Press.

[2] J. Henkel, T. Benner, and R. Ernst. Hardware generation and partitioning effects in the COSYMA system. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1993.

[3] A. Kalavade and E. A. Lee. The extended partitioning problem: Hardare/software mapping, scheduling, and implementation-bin selection. In G. D. Micheli, R. Ernst, and W. Wolf, editors, *Readings in hardware/software co-design*, Series in Systems on Silicon, pages 293–313. Morgan Kaufmann, June 2001.

[4] G. D. Micheli and R. K. Gupta. Hardware-software co-design. In G. D. Micheli, R. Ernst, and W. Wolf, editors, *Readings in hardware/software co-design*, Series in Systems on Silicon, pages 30–44. Morgan Kaufmann, June 2001.