SoC Synthesis with Automatic Hardware Software Interface Generation

Amarjeet Singh* Amit Chhabra† Anup Gangwar‡ Basant K. Dwivedi‡

M. Balakrishnan‡ Anshul Kumar‡

‡Dept. of Computer Science & Engg.

Indian Institute of Technology Delhi, New Delhi, India.

{anup, basant, mbala, anshul}@cse.iitd.ernet.in

Abstract

Design of efficient System-on-Chips (SoCs) require thorough application analysis to identify various compute intensive parts. These compute intensive parts can be mapped to hardware in order to meet the cost as well as the performance constraints. However, faster time to market requires automation of synthesis of these code segments of the application from high level specification such as C alongwith its interfaces. Such synthesis system should be able to generate hardware which is easily plug-gable in various types of architectures, as well as augment the application code to automatically take advantage of this new hardware component.

In this paper, we address this problem and present an approach for complete SoC synthesis. We automatically generate synthesizable VHDL for the compute intensive part of the application alongwith necessary interfaces. Our approach is generic in the sense that it supports various processors and buses by keeping a generic hardware interface on one end and a dedicated one on the other. The generated hardware can be used in a tightly or loosely coupled manner in terms of memory and register communication. We present the effectiveness of this approach for some commonly used image processing spatial filter applications.

1. Introduction

An estimation driven hardware-software codesign methodology, *ASSET*[12], is shown in Figure 1. It takes C specification which offers more flexibility for codesign and simulation. It consists of estimation techniques for hardware and software cost as well as performance metrics. These estimates are fed to the partitioner, which decides hardware and software parts of the application in order to meet various constraints. Finally hardware, software and interface synthesis are carried out along with system integration and verification.

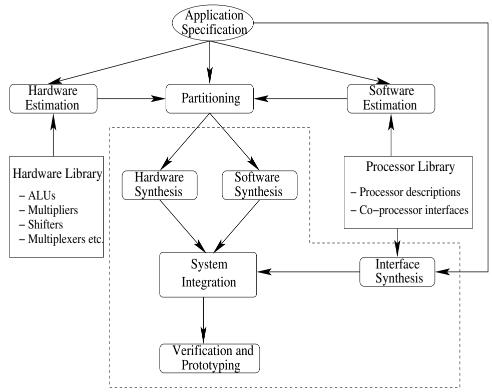


Figure 1. ASSET Codesign Methodology

Hardware synthesis plays very important role in the overall methodology described above. There have been various research efforts to come up with a good hardware compiler which can generate a synthesizable HDL from high level C specification of the application. In SiliconC[1], structural VHDL is generated for the C function. Prototype of the function becomes the entity. This system lacks support for pointer arithmetic as well as interface synthesis which is required when the generated hardware is integrated in the system. The Garp system[8] also uses a hardware compiler. Garp tightly couples a MIPS processor and a reconfigurable co-processor. Here VHDL is emitted for loops. It also takes care of interface of the hardware version of the loop. However, issues like quick reconfiguration of the FPGA etc. make their synthesis system different. SPARK[6] system uses various parallelizing compiler techniques such as speculation etc. and transforms C-specification into Register Transfer Level (RTL) VHDL. The focus is on control

*Currently with Tejas Networks, Bangalore, India. Email: amarjeet@tejasnetworks.com

†Currently with STMicroelectronics, NOIDA, India. Email: amit.chhabra@st.com

intensive applications. This system doesn't handle interface issues.

There has also been much interest in the Application Specific Instruction Set Processor (ASIP) synthesis [9]. Most of the ASIP synthesis frameworks are built around some architecture description language [5, 3, 7]. They allow to explore the architecture design space employing retargetable compilers and simulators. Except in LISA [4], HDL generation link is missing in other ADLs. Moreover, ADL based approaches do not address custom co-processor synthesis and automatic interface generation. The interface synthesis issue has also been addressed separately [2]. Most of interface synthesis approaches are library based and make certain assumptions about the nature of communicating processes and the input specification.

It can be observed that work has been done on various aspects of SoC synthesis. However, co-processor synthesis and automatic software and hardware interface generation in an integrated manner for the complete SoC is not well investigated. In this paper, we address this issue and present an approach for the complete SoC synthesis. We automatically generate synthesizable VHDL for the function of the C specification identified for hardware, for a typical domain of image processing applications. We also generate software and hardware interfaces which are needed for proper system integration. The interface synthesis is library based where the selection depends on the processor and the bus being considered. We have validated our methodology on frequently used image processing filtering applications, on a testbed which employs LEON[10] as the processor core.

The rest of the paper describes the methodology in detail and is organized as follows: Section 2 gives the details about the architecture template for various models supported by our C-to-VHDL translator. Section 4 gives details of this translator. It illustrates the communication protocol adopted and and the conversion methodology. Section 5 describes the testbed and Section 6 gives details of experiments. Finally Section 7 concludes.

2. Architecture Templates

Following are the different architectures which could possibly exist for a processor-coprocessor system.

1. The processor has several co-processors, which access only registers of the processor possibly by sharing the ports.

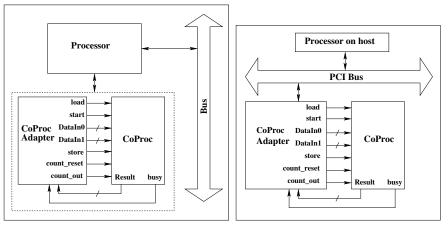
2. The processor has several co-processors and they can also access the memory. Since both the processor and co-processor use the same bus for communication with the memory, bus arbitration is required.

3. The co-processors can neither access registers nor memory. All the communication is through system bus employing handshake.

The first two architectures are closely coupled. In this configuration, the processor and co-processors are embedded in a single chip. If the processor allows only one custom co-processor, then rest of the co-processors can work together with the help of a co-processor adapter which ensures proper port sharing and communication of data. In the third architecture, the software part runs on the host machine. The custom co-processor sits on a board and works like a hardware accelerator. The software part communicates with the hardware part through the system bus which could be PCI for general purpose computing systems. This architecture is loosely coupled wherein the co-processor is not allowed to access either registers of the processor or the memory.

2.1. Closely Coupled Mode

Figure 2(a) shows the closely coupled configuration which we generate. We don't allow direct access to the memory mainly to keep things simple. Though we have only one co-processor in this mode, more co-processors can also be generated by appropriately customizing co-processor adapter (primarily for port sharing). The basic idea behind this configuration is to have an application specific functional unit (FU) which exploits certain features of the application in an effective manner. The support for this FU comes by either augmenting the instruction set of the processor or by using instructions provided to communication with the co-processor. This has further been described in the context of LEON[10] processor in Section 5.



(a) Closely coupled mode

(b) Loosely coupled mode

Figure 2. Processor Co-processor modes

2.2. Loosely Coupled Mode

Hardware accelerators are quite popular in the application domains such as Graphics, DSP, Multimedia etc.

This is a loosely coupled configuration (Figure 2(b)) where the co-processor (or hardware accelerator) cannot access the register files of the processor. The processor and co-processor communication takes place through the PCI bus. The *CoProc Adapter* is specific to the mode of communication and buffers all the values sent by the processor before making them available to the co-processor. Most of these are hand designed. Large design time puts the need for automation. To this end, we use the same interface of the hardware version of the function shown in the Figure 3 alongwith the *CoProc Adapter* which is selected from the library of co-processor interfaces.

3. Co-Processor and Co-processor Adapter

Our C-To-VHDL translator generates the co-processor from a C function. This opens several possibilities of co-processor interface. At one end, there could be as many input ports for the operands as there are input parameters in the function. However, in a closely coupled mode, there might be smaller number of register ports in the register files of the processor than required by the co-processor interface. In that case, the co-processor adapter will have to buffer all the operands before passing them onto the co-processor. Apart from the case when number of operands of the co-processor exactly match number of source operands in the processor instruction which invokes the co-processor, buffering of operands can be offloaded to the co-processor and adapter can be made simple.

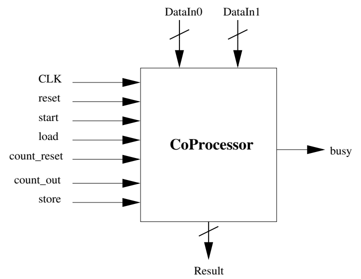


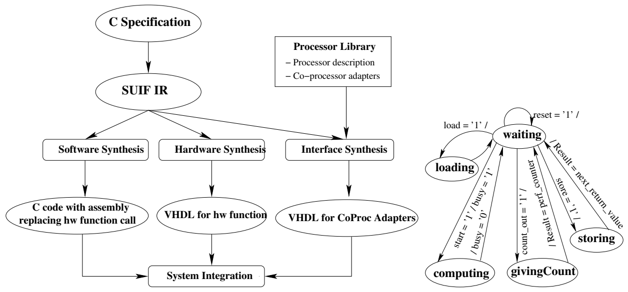
Figure 3. Generic Co-Processor Interface

Figure 3 shows the interface of automatically generated VHDL of the hardware function call. The choice of only two input ports for operands is motivated by the fact that most of the instructions of the processor will have two source operands and one destination operand. Since only two operands come at a time, we maintain a parameter array inside the co-processor and a load_counter inside it ensures loading of the operands at correct location. *load* signal enables the loading of the operands. A high on *reset* signal triggers the reset operations such as making the counters zero etc. *start* signal triggers the actual computation. *busy* goes high as soon as *start* comes, to denote that the co-processor is busy executing at the moment. It goes low once the computation is over.

A function could be multiple valued when a *structure* is returned from the function. By default, once the computation is over, first return value is available. High on *store* signal allows to get next return values of the function on Result lines. Internally the co-processor maintains a store_counter which ensures availability of the correct return value.

Additionally co-processor has a performance counter inside it. The basic idea behind this counter is to know the utilization of the co-processor during execution of the whole application. *count_reset* signal is used to reset the performance counter and high on *count_out* signal allows to get performance counter value from the unit. This information can be used for profiling and debugging purposes.

4. C-to-VHDL Translation



(a) Translation Methodology

(b) FSM for HW function

Figure 4. C-to-VHDL Conversion

Figure 4(a) shows the overall translation methodology The C specification of the application is converted to SUIF[11] intermediate representation. The conversion process assumes that the part of the application to be mapped onto the hardware, has already been identified. The identified function of the application is converted to the synthesizable VHDL which confirms to the interface shown in the Figure 5. The translator also performs software synthesis. The co-processor adapter is generated by selection from the processor library and customized for appropriate buffer size and bitwidth as previously discussed. The co-processor and its adapter is synthesized and C code of the software part is compiled. All these are put together and the system is integrated. One can also apply some optimization passes available in the SUIF compiler, however our C-to-VHDL translation doesn't depend on these.

4.1. Software Synthesis

As we have said the granularity of the C code segment which is converted to the VHDL, is at function level. A software function call proceeds as follows: **1)** pass the input parameters, **2)** start the function execution and **3)** store the results at appropriate place.

When the function gets converted into the hardware, following steps are taken which have a one-to-one correspondence with the software call: **1)** load the operands in the co-processor register file, **2)** assert start signal to the co-processor and **3)** store the values back.

The extra code required to facilitate hardware function call depends on the processor. For example, LEON (described in the next section) offers its floating point unit interface to connect custom co-processors. Another situation where extra code needs to be generated is when the co-processor has more than two input parameters. Extra code will be required to load the parameters in the co-processor as there are only two data input ports in the co-processor interface. Currently we perform this additional assembly code generation for LEON, but the method is general and can be applied to other processors as well depending on their availability in the processor library. We place this assembly in the SUIF IR by replacing the hardware function call. This IR is converted back to C using available SUIF-to-C converter. The generated C code is compiled and run on the processor augmented with our co-processor.

4.2. Hardware Synthesis

We support a restricted subset of C for hardware generation. **1)** The function should be either be a Multiple Input Single Output (MISO) or Multiple Input Multiple Output (MISO). **2)** Pointers and nested function calls are not supported. **3)** Floating point variables are not supported because of large FPU cost. **4)** Memory communication is not permitted.

Apart from these there is one-to-one correspondence between C and VHDL. Only function parameters, return values and structures are treated differently.

Since the co-processor interface only accepts two inputs at a time, these need to be buffered inside in case more number of source operands are present. A set of registers, which is internal to the co-processor is used. Another register, *param_counter*, keeps track of how many operands have been loaded till now. Once all the input parameters arrive, the generated FSM will not allow any more load to be done. The register, *param_counter* is reset to zero once the computation starts. Similar treatment is given to the return values. Only difference is that every time *store* goes high which corresponds to the request for the next return value, *store_counter* increments. As soon as all the returns values are read, *store_counter* resets. Structures defined in the function are taken care by defining separate variable for different fields of the *struct* variable type.

Generated FSM of the function is shown in the Figure 4(b). There are five states in this FSM. While in *waiting* state, based on value of start, load, count_out or store, it decides next state. Most of the time the co-processor will remain in this state waiting for one of the 4 signals to arrive to start the computation. It goes back to *waiting* state once loading of both the operands is complete. Actual computation is performed in the *computing* state. Once computation is over, busy signal goes low and results are available. By default, first return value is available. Rest of the values can be availed by asserting store signal high in the coming cycles. Just like load counter, a store counter is appropriately updated in the *storing* state and correct return value is availed.

For profiling and debugging purposes, the FSM also maintains a performance counter. This counter gets incremented every cycle during the period when busy is high. The value of the performance counter can be obtained at some point during execution by asserting count_out signal. Here there will be a state transition from *waiting* to *givingCount* state.

4.3. Interface Synthesis

The interface synthesis is library based, wherein co-processor adapter is selected from the processor library, as shown in Figure 4(a). There are two main parameters to customize. The first parameter is amount of buffer required for the source operands, which is decided by the hardware function prototype. The second parameter is interface to the co-processor. Here we fine tune the adapter co-processor interface as per the bitwidth of the computation being done within the co-processor.

5. Experimental Setup

We obtain our results by simulating the compiler generated binary code for each benchmark application, over the LEON RTL-VHDL model in Modelsim VHDL simulator. The synthesis results have been obtained using FPGA Express targeted for XCV-800 FPGA.

5.1. LEON Processor Co-processor Interface

We have used the LEON processor alongwith the generated VHDL for the hardware function call as our testbed. The LEON VHDL model implements a 32-bit processor conforming to the SPARC V8 architecture. It is designed for embedded applications with many features. These include separate data and instruction cache, two UARTs, flexible memory controller and the provision to add additional modules such as the Floating Point Unit (FPU) and some application specific functional units acting as co-processors.

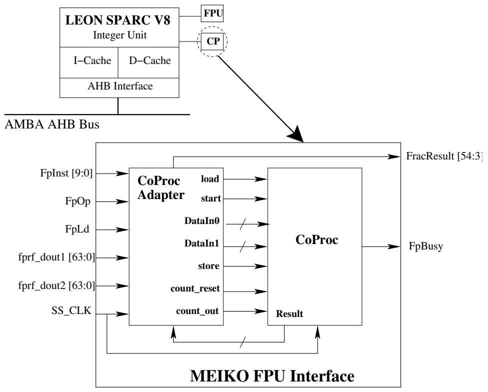


Figure 5. Generated Co-Processor with LEON Co-Processor Interface

The LEON model does not include a co-processor, but provides an interface to *Meiko Floating-Point Unit* core. Currently Sun is the licensee of this core. Its interface is provided to allow the interfacing of floating point units or other custom co-processors, with all the memory accesses being made through the integer unit. It also allows the execution unit of LEON to operate in parallel to increase performance. However, we are using the serial co-processor interface which halts the integer unit while the co-processor is executing. When finished, the result is written back to the co-processor register file. Figure 5 shows this configuration. *CoProc Adapter* interprets the opcodes and drives the *CoProc* which performs the actual computation. *CoProc* is generic and independent of LEON co-processor interface, but *CoProc Adapter* is specific to this interface as it has to decode the opcodes accordingly. This adapter is selected from a set of co-processor interfaces available in the processor library shown in the Figure 1.

The co-processor is started by asserting the signal *FpOp* together with a valid opcode *FpInst*. The operands(*fpu_dout0, fpu_dout1*) are driven on the following cycle together with the *FpLd* signal. If the instruction takes more than one cycle to complete, the co-processor must drive *FpBusy* from the cycle after the *FpOp* signal was asserted, until the cycle before the result is valid. The result *FracResult* is valid from the cycle after the de-assertion of *FpBusy*, and until the next assertion of *FpOp*.

5.2. Benchmark Applications

We have chosen three image filtering applications to demonstrate the results. The first application is *smoothing filtering* which is a common operation to smoothen the image and reduce the noise contents. Mask for this filter is shown in the Figure 6(e). Essentially this application performs following operation on every pixel of the image.

$$G = (1/8) * (z1 + z2 + z3 + z4 + z5 + z6 + z7 + z8 + z9) \quad (1)$$

Here zi is the value of the pixel at ith location in the image where the mask is positioned. Significant speedup can be achieved provided this operation goes to the hardware.

The second application is *gradient filtering*. This application is the core of many edge detection processes. Following operation takes place for every pixel:

$$Gx = (z7 + 2z8 + z9) - (z1 + 2z2 + z3) \quad (2)$$

$$Gy = (z3 + 2z6 + z9) - (z1 + 2z4 + z7) \quad (3)$$

$$G = abs(Gx) + abs(Gy) \quad (4)$$

Mask of this application is shown in the Figure 6(b) and 6(c). The third application which we have chosen is *Laplacian of Gaussian - LoG*. This is also used in various edge detection algorithms. It performs following operation:

$$G = 4z5 - (z2 + z4 + z6 + z8) \quad (5)$$

The mask for LoG is shown in the Figure 6(d). All these applications take the pixel values in the range 0-255. Hence only 8 bit datapath is required. As a consequence, it is possible to pack upto four values in a single precision co-processor operand. In order to allow efficient mapping to hardware, we perform division and multiplication by powers of 2 (in this case 8 instead of 9). This allows to perform the required operation by just a re-assignment of signal lines. For example, division by 8 is performed by mapping lines 4 to 0 of the result to lines 7 to 3 of the sum in case of smoothing filter.

6. Results and Analysis

	Smoothing	Gradient	LoG
Without Coproc	334868	367324	363251
With Coproc	305119	305119	213831
% Gain	8	16	41

Table 1. Cycle count on 64x64 image

The speedup obtained is heavily influenced by two factors: **a)** the amount of computation being performed inside the co-processor and, **b)** the amount of data being transferred to the co-processor. As mentioned in the last section, we pack 4 pixel values inside a single 32 bit operand. This packing is carried out using the normal *IALU* operations.

z1	z2	z3
z4	z5	z6
z7	z8	z9

(a) 3x3 region

−1	−2	−1
0	0	0
1	2	1

(b) Gx

−1	0	1
−2	0	2
−1	0	1

(c) Gy

0	−1	0
−1	4	−1
0	−1	0

(d) LoG

$1/9$ x

1	1	1
1	1	1
1	1	1

(e) Smoothing

Figure 6. Masks for image filtering in spatial domain

Packing is required to efficiently move data between integer register file and the co-processor register file. Since, the SPARC architecture does not permit direct data movement between these two register files any operand residing in the integer register file first needs to be stored in the memory. This incurs an overhead of a memory load/store for the each operand of the co-processor. So, by efficiently packing pixel values inside a 32 bit integer, we are able to transfer 4 pixel values by incurring an overhead of just a single memory load/store.

For different benchmarks, the co-processor is able to generate result within a single cycle. This is justified as there are no complex operations to be performed between any two pixel values. Multiplication and division are performed by signal re-assignment and additions are performed in parallel. However, the single *IALU* would take much more than this even when it performs all the operations in a pipelined fashion.

As shown in Table 1, the least gain is obtained in the case of smoothing filter, since this is very data intensive, but not so compute intensive. The gradient filter is quite compute intensive, but is also data intensive. As a consequence, lot of gain obtained is lost in packing and transferring of operands. LoG is the least data intensive and moderately compute intensive. That is why a huge gain of 41% is obtained.

	No CoProc	Smoothing	Gradient	LoG
Slices	38	50	50	48
LUTs	36	46	46	43
FFs	8	10	10	10

Table 2. % resource utilization on FPGA

We plug our co-processor as the execution unit inside the co-processor pipeline. The complexity of this pipeline is of the order of integer unit. As a result, a large increase in device utilization is observed between LEON without a co-processor and one which has a co-processor attached. However, not much variation in device utilization is obtained by slightly changing the execution unit. This is clearly shown in Table 2.

7. Conclusion and Future Work

We have presented an approach for Complete SoC synthesis. This allows to generate synthesizable VHDL for compute intensive application function and its associated software and hardware interfaces. We have also shown how the approach allows to build a system in tightly or loosely coupled manner based on processor and communication media being considered.

Currently we don't allow generated co-processor to communicate with the memory. We are working towards removing this limitation. This will also open possibility of generating co-processor for several other applications. We are also exploring possibility of incorporating pipelining etc. to further enhance performance.

References

[1] C. Scott Ananian. SiliconC: A Hardware Backend for SUIF. http://flex-compiler.lcs.mit.edu/SiliconC.

[2] Arvind Rajawat et al. Interface Synthesis: Issues and Approaches. In *Proc. Int. Conf. on VLSI Design*, 2000.

[3] J. Gyllenhaal et al. HMDES version 2.0 specification, IMPACT, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-96-03, 1996.

[4] Oliver Schliebusch et al. Architecture Implementation Using the Machine Description Language LISA. *ASP-DAC/VLSI Design 2002, Bangalore, India*, pages 239–244, Jan. 2002.

[5] Peter Grun et al. EXPRESSION: An ADL for System Level Design Exploration. Technical Report 98-29, University of California at Irvine, September 1998.

[6] S. Gupta et al. Conditional Speculation and its Effects on Performance and Area for High-Level Synthesis. In *Proc. ISSS*, 2001.

[7] Stefan Pees et al. LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *Proc. DAC*, 1999.

[8] Timothy J. Callahan et al. The Garp Architecture and C Compiler. *IEEE Computer*, April 2000.

[9] M. K. Jain, M. Balakrishnan, and A. Kumar. ASIP Design Methodologies : Survey and Issues. In *Proc. Int. Conf. on VLSI Design*, 2001.

[10] http://www.gaisler.com/leon.html.

[11] http://suif.stanford.edu.

[12] Synthesis Methodology for Real-Time Embedded Systems for Vision and Image Processing. http://www.cse.iitd.ernet.in/esproject.