Towards a Tighter Integration of Generated and Custom-Made Hardware

Harald Devos, Wim Meeus, and Dirk Stroobandt

Hardware and Embedded Systems Group, ELIS Dept., Ghent University,
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{harald.devos|wim.meeus|dirk.stroobandt}@elis.UGent.be

Abstract. Most of today's high-level synthesis tools offer a fixed set of interfaces to communicate with the outer world. A direct integration of custom IP in the datapath would often be more beneficial than an integration using such communication interfaces. If a certain interface protocol is not offered by the tool, either translation blocks (wrappers) are needed or the code should be written at a lower level. The former solution may hurt the performance, while the latter one is often impossible using an untimed high-level description.

In this paper interface protocols or sets of IP core accesses are first described at a low level as sets of operations with scheduling information (macros). During the synthesis process, corresponding function calls are mapped to these macros. This facilitates the integration of custom-made hardware and hardware generated by high-level synthesis tools.

1 Introduction

The increasing number of transistors that can be put on a single chip, has led to a growing design gap. Therefore, hardware design has to be lifted to a higher level, as exemplified by the growing interest in high-level synthesis (HLS) techniques [5]. However, hardware designers like to have full control over critical design parts, which hinders the adoption of current HLS environments.

In the software world high-level languages have become mainstream since long. However, for critical pieces of code, manually written assembler code may still outperform compiled high-level code. *In-line assembler* allows to exploit the full power of certain processor instructions while still enjoying the benefits of high-level languages for the less critical program code. It would be very interesting to have a similar option for hardware design. The limitations of current HLS tools will be more tolerable if the designer can take over control for those parts of a design where the synthesis results are not satisfactory.

Many HLS tools offer a built-in library of macros to communicate with the *outer world*. The inclusion of a custom-made block should typically be specified as communication with an *external block*, that has to fit a certain interface. A translator block, called a wrapper, may be used to translate one interface into another (Fig. 1), but this may create an overhead (see Sect. 4). Describing the communication at a lower level, e.g., using multiple statements in the high-level

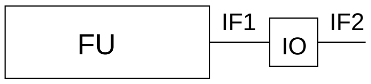
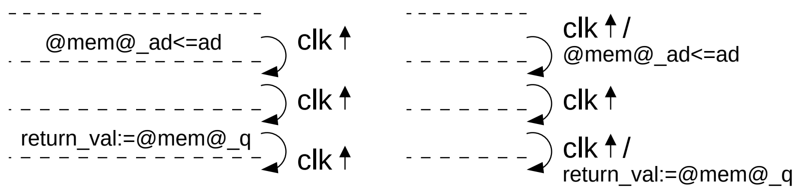


Fig. 1. An IO hardware block translates the interface of the functional unit (IF1) to the desired interface (IF2).



(a) (b) Alternative representation of (a)

```
defaults (
  @mem@_ad <= (others => '−');
  @mem@_d <= (others => '−');
  @mem@_we <= '0';
  )
```

(c) Default values of outputs

```
schedule:
  @mem@_ad <= ad;
  <nc>
  <nc>
  return_val := @mem@_q;
  <end>
```

(d) Textual schedule representation

Fig. 2. Protocol description for a single cycle memory read. The `@mem@` substring is a placeholder for the name of the memory instance. <nc>= new cycle.

input language to describe one external transaction, may often be impossible since the concurrency of signal assignments cannot be forced in untimed C code.

This paper presents a method to describe interface protocols as a sequence of schedule[1] steps with transition information (Sect. 2), such that it can be dealt with on equal terms with the built-in macros or interfaces of the HLS tool. Macros may also be used to interact with custom-made IP blocks in a more integrated way. Our scheduler is able to pipeline transactions and operations without specific directives (Sect. 3). Hereby, we still have the advantages of a modular design flow: portability and a clear separation for the designer of communication and functional behavior, but with less overhead thanks to the tighter integration of these two in the generated hardware.

2 Macro Schedule Description

A macro is a set of operations described at a cycle- and pin-accurate level that together with the external hardware it communicates with implements a functionality corresponding to one high-level construct, e.g., a memory access, a

[1] Scheduling in HLS is the task of assigning operations to clock cycles (or control steps) in a static way, i.e. in the hardware generation process.

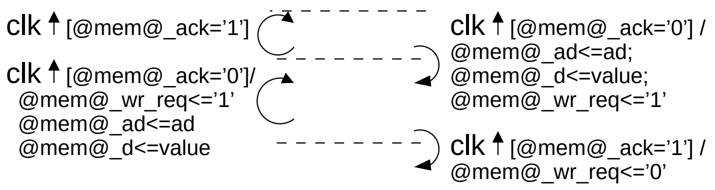


Fig. 3. Protocol description for a memory write with a full handshake.

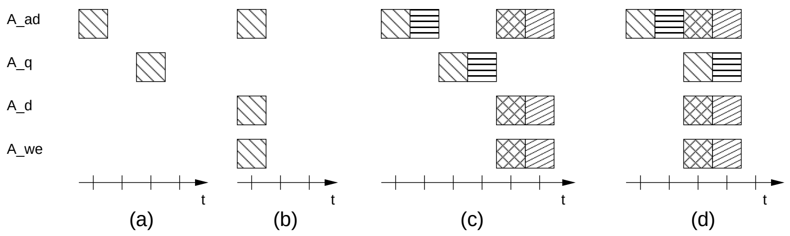
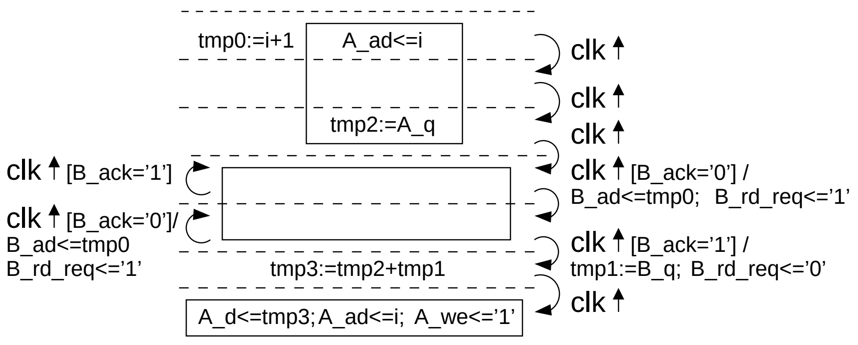


Fig. 4. Occupation of ports for single-cycle memory transactions (a) read, (b) write. Both reads and writes can be pipelined with initiation interval 1 (c), but also a mix of reads and writes (d).

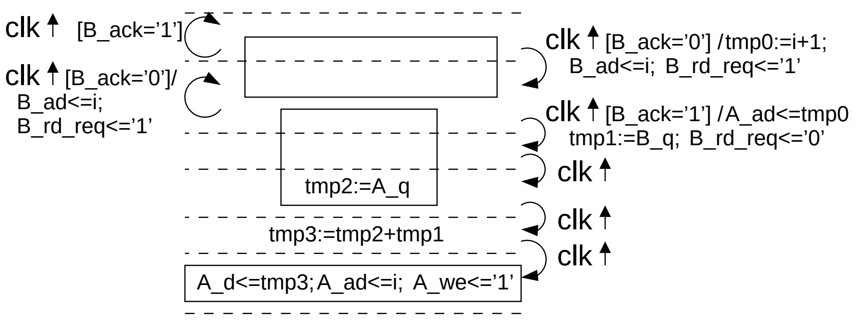
mathematical operation performed by an IP core, . . . (In this section we restrict ourselves to well known memory examples. Real IP-cores are used in Sect. 4.2). Such a macro can be described as a finite state machine (Fig. 2 and Fig. 3). The area between two dashed lines corresponds to one state. Transitions, indicated with arrows, are triggered by events (clk $\uparrow$), possibly guarded by conditions put between `[ ]`. Operations that are executed at a state transition are put behind a '`/`'. State charts only execute operations at state transitions. We use a more compact notation where operations inside a state represent operations that are executed at each outgoing transition as explicitly shown in Fig. 2(a and b). For each macro a corresponding C function is written. During synthesis, calls to these functions are mapped to the corresponding macros.

From this information, our scheduler derives which signals are used by a macro instance in which cycles (Fig. 4(a and b)). It is able to overlap several instances of one macro (Fig. 4(c)), but also instances of different macros sharing ports (Fig. 4(d)). If in a clock cycle a macro does not apply a value to an output, that output can be used by another macro execution. Else, it receives a default safe value (Fig. 2(c)). That these default values are not forced by macros that do not use a certain output is essential to the ability of automatically pipelining macro instances.

Conditional state transitions (Fig. 3) result in an indeterministic execution time. Our scheduler is able to combine macros (and built-in constructs) with both deterministic and non-deterministic execution time (Sect. 3).



(a) Schedule for `A[i]=A[i]+B[i+1]`



(b) Schedule for `A[i]=A[i+1]+B[i]`

Fig. 5. Behavioral templates (boxes) take care of the relative schedule constraints (offsets) of different operations of a macro during the scheduling process.

3 The Scheduling Algorithm

Scheduling operations with fixed time intervals can be done using behavioral templates [11]. A behavioral template is a set of nodes of a control data flow graph (CDFG) coupled with cycle offsets. If one of the nodes of the template is scheduled, the schedules of all the other nodes are defined by the relative offsets. Behavioral templates for a single cycle read and write are recognized in the first and last cycle of the schedule in Fig. 5(a). We extended the concept of behavioral templates with conditional state transitions that introduce an indeterministic execution time (E.g., handshake read of array `B` in Fig. 5.). A conditional state transition cannot be scheduled in parallel with a template that has a fixed cycle offset, unless some kind of micro-threading is used (which is kept as future work). This is regarded as a *conflicting resource constraint.* However, single cycle operations can be made on the conditional state transitions that are taken exactly once, e.g., `tmp0:=i+1` in Fig. 5(b).

Our scheduling algorithm (Fig. 6) schedules operations as soon as possible. If two operations are ready to be scheduled but have conflicting constraints the one that comes first in the input C code is selected. In future work we will explore

```
while nodes remain to be scheduled {
    ready list := all nodes that do not depend on unscheduled nodes
    for each node in the ready list {
        find earliest execution cycle of this node based on dependencies
        while conflicting resource constraints
            increase execution cycle of this node
        schedule this node at the found execution cycle
        update consumed resources
    }
}
```

Fig. 6. The scheduling algorithm. A *node* is a behavioral template consisting of several CDFG operations, or a single operation that is not included in any behavioral template (could be regarded as a singleton behavioral template).

more advanced scheduling heuristics. The outer loop does not iterate over cycles, but over the nodes to be scheduled. This differs from classical algorithms. The nodes in the algorithm are behavioral templates that may consist out of multiple operations scheduled together or operations not being part of a behavioral template, regarded as singleton behavioral templates (that can be combined with conditional transitions).

4 Experiments

Our high-level synthesis environment, JCCI [6] (Java C to CLooG [3] Interface), reads in macro descriptions (in a textual representation, cf. Fig. 2(d)). Function calls in the C input code that correspond to macros are recognized and replaced with the scheduling information of the corresponding macros.

4.1 Memory and FIFO Interfaces for a Sobel Edge Detector

The kernel of the Sobel edge detector [8] algorithm contains a 3×3 sliding window operation. Five conceptually different designs were described in C.

1: The standard textbook algorithm. 2: Variables store the input values that are reused in the next two column iterations. This reduces the number of memory reads with a factor three. 3: Vertical reuses are exploited by storing two lines of the image in FIFO buffers. This reduces the memory reads with roughly another factor three. The FIFO buffers are created as IP cores (Altera MegaWizard) and accessed using macros. Designs 2 and 3 are similar to the designs listed in [7, 12]. 4: similar to 3, but the FIFO is made using single port memory blocks, with a macro that implements the address increments with wrap around. 5: no FIFO macros used. A C description of FIFOs is used to buffer the two lines of pixels.

For each of these designs we generated three variants with a different memory interface (Fig. 7). A first variant uses a single-cycle memory (cf. Fig. 2). A second one uses a memory with a full handshake protocol (cf. Fig. 3). A third variant is

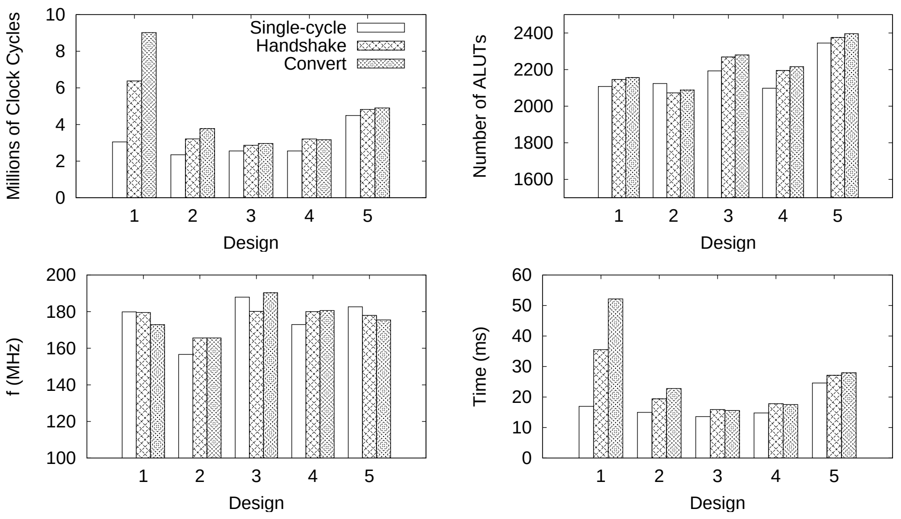


Fig. 7. Simulation and synthesis results for 5 implementations of a Sobel edge detector on an Altera Stratix III (EP3SL150F1152C2) for different interfaces. ALUT = Adaptive Look-Up-Table. Execution time and number of clock cycles for a 320×320 input image. The number of DSP and memory blocks used are not plotted since they do not depend on the IO interface.

identical to the second extended with a block that converts the full handshake protocol into a single-cycle protocol, similar to Fig. 1 (with IF1 the handshaking and IF2 the single-cycle protocol). This mimics the situation where a protocol is not part of the HLS tool's library and a wrapper is inserted.

Results and Discussion The interface conversion blocks introduce (on the average) an overhead in area and clock cycles (*Convert* vs. *Single-cycle* in Fig. 7). In some cases a wrapper augments the clock frequency, but this is counterbalanced by the increased number of cycles. It is thus beneficial to include the proper protocol directly. The usage of macros to describe the FIFOs gives better results than the equivalent description in C code. Probably, other HLS tools may generate line buffers in a much more efficient way. However, the point we want to make here is that the user can take control over the synthesis process when the HLS techniques do not give satisfactory results.

4.2 Integration of Floating-Point Megacores

The Altera Floating Point Megafunctions [1] implement IEEE 754-compliant floating-point operations. The cores are pipelined and have a known latency (add/sub: 7 cycles, mul: 5 cycles, div: 6 cycles). The addition and subtraction use the same block. A select signal is used to choose between the two. After writing a macro description (applying the input values in the first cycle and reading the output a fixed number of cycles later), a C function that performs one floating-point operation can be synthesized using the corresponding IP core. Our scheduler is able to pipeline all operations and it knows that a sub and an add cannot be started at the same time. We tested the macros by implementing the Newton-Raphson algorithm to find a zero of a fifth degree polynomial:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{a_5x_n^5 + a_4x_n^4 + a_3x_n^3 + a_2x_n^2 + a_1x_n + a_0}{5a_5x_n^4 + 4a_4x_n^3 + 3a_3x_n^2 + 2a_2x_n + a_1} \ . \quad (1)$$

The time needed to set up this design was less than two hours from C-code to synthesisable VHDL: 15' to create the Altera floating point megacores with the Altera QuartusII MegaWizard Plug-In Manager; 15' to convert all floating point operations in the C code into function calls, such as fp_add, fp_sub, ... (E.g. `x2=x*x;` becomes `x2=fp_mul(x,x);`); 30' to describe the macro schedules for each of these floating point operations; some additional time for constructing the top-level and testbench, and testing. On an Altera Stratix III (EP3SL150F1152C2), the design can clock at 150.60 MHz, and does one evaluation of (1) in 69 cycles, i.e. 458 ns. The design uses 20 DSP blocks (5% of the FPGA), 4608 Memory bits (< 1%), 2147 combinational ALUTs (2%), 26 memory ALUTs (< 1%) and 3079 dedicated logic registers (3%). On an Intel Core2 Quad CPU Q6600 at 2.40 GHz the same computation takes 43 ns or roughly 90 cycles. The focus of this experiment, however, is not the comparison of the FPGA and processor implementations, but the ease of integration of the floating-point cores.

5 Related Work

Some HLS tools focus on the datapath and have little attention for the communication interface. SPARK [9], e.g., generates 1 port for each array element. Impulse-C [10] offers a set of macros for the communication with hardware blocks, e.g., FIFOs and channels, but no user-specified interfaces can be defined.

The approach of Mentor Graphics' Catapult C [4] is similar to ours. Catapult C accepts a pure untimed C++ description as its input. The user can define its own interfaces with a library builder. The interface resource consists of an *I/O hardware block* connected to a standard *interface* (similar to the architecture in Fig. 1). Property mappings are used to describe the delay (for combinational logic) or initiation delay (needed for pipelining) of a transaction. This allows to pipeline different instantiations of the same transaction as in Fig. 4(c). We doubt, however, if it would be possible to detect a pipeline option as in Fig. 4(d).

Transaction Level Modeling (TLM) using SystemC is used for system-level design and architecture exploration. TLM-RTL transactors connect blocks with interfaces at different abstraction levels: a TLM interface, which is a function call, and an RTL interface. For simulation, this is similar to the concept of macros. However, our focus is on the hardware generation and not on the modeling.

SpecC has modeling capabilities similar to SystemC but has a designer-assisted tool flow to RTL [13]. Communication is implemented with protocol adapters, similar to [2], where an IO block is inserted for interface translation.

6 Conclusions

The growing complexity of digital system design has raised the need for more flexible HLS tools. This paper presents a method to extend such a tool with the ability to generate custom interface protocols or IP core integration, without the overhead that would be caused by wrappers. This enables a tighter integration of generated and custom-made hardware and may create a design flow in which the strengths of HLS tools and manual hardware design can be combined.

Acknowledgements This research is supported by the I.W.T. grant 060068. Our research group at Ghent University is a member of the HiPEAC Network of Excellence. We would like to thank Altera for donating a Stratix III board.

References

1. Altera. *Floating-Point Megafunctions User Guide*, 1.0 edition, March 2009.
2. Felice Balarin and Roberto Passerone. Functional verification methodology based on formal interface specification and transactor generation. In *Design, Automation and Test in Europe (DATE)*, pages 1013–1018, 2006.
3. Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Juan-les-Pins, September 2004.
4. Bryan Darrell Bowyer, Peter Pius Gutberlet, and Simon Joshua Waters. Interactive interface resource allocation in a behavioral synthesis tool, US Patent 0077906, March 2008.
5. Philippe Coussy and Adam Morawiec, editors. *High-Level Synthesis: From Algorithm to Digital Circuit.* Springer, 2008.
6. Harald Devos, Wim Meeus, and Dirk Stroobandt. CLooGVHDL and JCCI. In *DATE University Booth*, pages 1–2 (CD–ROM), Nice, France, April 2009.
7. Harald Devos, Jan Van Campenhout, Ingrid Verbauwhede, and Dirk Stroobandt. Constructing application-specific memory hierarchies on FPGAs. *Transactions on High-Performance Embedded Architectures and Compilers*, 3(3), 2008, To appear in LNCS.
8. Bill Green. Edge detection tutorial. `http://www.pages.drexel.edu/~weg22/edge.html`, 2002. Last Accessed 2009.02.24.
9. Sumit Gupta, Rajesh Gupta, Nikil Dutt, and Alexandru Nicolau. *SPARK, A Parallelizing Approach to the High-Level Synthesis of Digital Circuits.* Kluwer Academic Publishers, 2004.
10. Impulse Accelerated Technologies. Impulse C. `http://www.impulsec.com/`, Last Accessed 01.2009.
11. Tai Ly, David Knapp, Ron Miller, and Don MacMillen. Scheduling using behavioral templates. In *Proceedings of the 32nd Design Automation Conference (DAC)*, pages 101–106. ACM, 1995.
12. Craig Moore, Harald Devos, and Dirk Stroobandt. Optimizing on-chip memory systems for FPGAs. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2009.
13. Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. An interactive design environment for C-based high-level synthesis of RTL processors. *IEEE Trans. on VLSI Systems*, 16(4):466–475, April 2008.