RAISING THE ABSTRACTION LEVEL OF HDL FOR CONTROL-DOMINANT APPLICATIONS

Marc-Andre Daigneault and Jean Pierre David

Department of Electrical Engineering,
Ecole Polytechnique de Montreal
Montreal, Canada
email: marc-andre.daigneault@polymtl.ca, jpdavid@polymtl.ca

ABSTRACT

As the complexity of modern digital systems continues to increase exponentially, the need for beyond RTL design methodologies is growing as well. In this paper, we propose a high-level hardware description language that allows the user to dynamically modify and constrain the connections between data token sources and sinks. Actual transfers occur when both sources and sinks are ready to proceed, according to different predefined synchronization protocols. At this level of abstraction, both FSM programming and constraint programming paradigms are combined to enhance the user's ability to describe and exploit fine-grain parallelism in control-intensive hardware designs. The proposed hardware description methodology is applied to the description of two hardware implementations of the *QuickSort* algorithm, using pipelined memory and comparator components.

1. INTRODUCTION

It has been nearly three decades since the first commercially available FPGA was released in 1985, delivering 64 configurable logic blocks and 58 input/output blocks from its 85,000 transistors [1]. In modern time, FPGAs have become multi-billion transistor chips, delivering multi-million bits of on-chip memory, tens of thousands of registers, and hundreds of DSP blocks. Meanwhile, it has become increasingly hard to fully exploit the potential offered by these programmable silicon estates. As this so called productivity gap widens, the need for a beyond-RTL hardware description era is clear, but it isn't new [2]. Despite tremendous efforts, raising the level of abstraction has proven to be harder in the context of concurrent hardware descriptions than it has been for single-threaded sequential software description languages.

In this work, we present a HDL that raises the level of abstraction at which the designer manages data token producers and consumers. A set of concurrent high level state machines dynamically control the activation and deactivation of connections between data sources and sinks. The proposed language actually builds on the CASM language [3]. Nevertheless, in order to express behaviors involving interdependent data connexions and transfers (such as synchronization, exclusivity, arbitration, constrained scheduling and others), we propose to use logical implication rules governing the authorization of data transfers over these connections. All the control and synchronization logic required is generated automatically by a compiler.

The rest of this paper is organized as follows: Section II presents a brief overview of related work. Section III is dedicated to the presentation of the main features and concepts of the proposed hardware description language. To illustrate the benefits of the proposed abstraction level, Section IV presents a simple hardware description of the QuickSort algorithm, which is further refined to produce more efficient, functional-pipeline design. Both implementations rely on 2-cycle latency pipelined comparator and on-chip memories. Section V presents the compilation and synthesis results obtained for both implementations targeting Altera's Stratix-III FPGA. Section VI concludes this work.

2. RELATED WORK

Most of the past effort over the last three decades in the field of high-level synthesis has been oriented towards behavior synthesis, which most often starts from sequential C/C++ descriptions, and more recently SystemC [4, 5]. As such languages are already frequently used in modern Electronic System Level design methodologies, which promote the utilization of executable specifications, one can understand the strong incentives for an automated path to silicon. But still, while this research has resulted in a mature industry [6, 7], there are many challenges to the synthesis of sequential descriptions, and behavioral synthesis is still not widely accepted as a replacement for RTL design methodology [8, 2].

Nevertheless, the spectrum of modern high-level synthesis goes beyond C-like synthesis. BlueSpec SystemVerilog

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

978-1-4673-2256-0/12/$31.00

(BSV) is of interest in our context because it makes use of guarded actions, a concept close to our constraint programming paradigm. BSV allows the designer to specify atomic guarded actions that will occur only if some predicate is true. Rules can be composed and their associated actions scheduled to form new behaviors while preserving atomicity [9].

3. PROPOSED HDL AND IMPLEMENTATION

3.1. Data-synchronized connections

Three synchronization protocols are presently supported. The *full-synchronization (FS)* protocol ensures that no data token is lost. A transfer occurs each time the ready-to-send (RTS, source driven) and ready-to-receive (RTR, sink driven) signals are both asserted on the rising edge of the clock signal. The *half-synchronization (HS)* protocol assumes that the sink is always ready (no RTR signal). The *no-synchronization (NS)* protocol assumes that both source and sink are always ready (no synchronization signal at all). The NS protocol is the equivalent of *wires* in Verilog or *signals* in VHDL. Full-synchronization and half-synchronization protocols are illustrated in Figure 1. Each source and sink of the design must comply to one of these protocols and provide the adequate synchronization signals.

Two types of connections are presently supported by the compiler. The '=' operator instantiates a connection that will last until one (and only one) data token is actually transferred. Moreover, this connection will block the ASM's state flow until the transfer is complete. The '*=' operator allows multiple transfers and does not block the ASM's state flow.

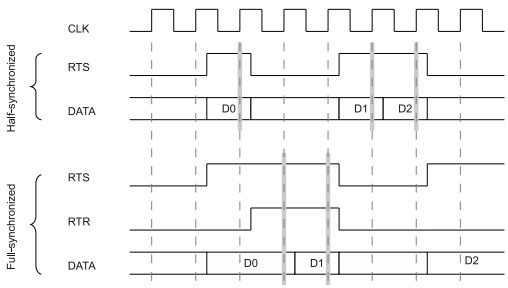

Fig. 1. Illustration of the two basic synchronization protocols present in the CASM language.

3.2. High-level Algorithmic State Machines

The proposed language offers a high-level ASM-programming model. To each ASM's state we associate one *activation state* to each branch combination of its conditional statements. Each activation state may contain connection and rule statements, and can last several clock cycles in the presence of blocking connections. To each state must be assigned a *next* state using either **goto** or **call/return** statements. Each call specifies a state to branch to, as well as a state to return to since no default successor exists in an ASM. A call stack is automatically instantiated in the design. The single state ASM presented in Figure 2 contains four states. In state *N0*, the ASM reads a command from a FIFO and then branches to *N1*, which initiates the corresponding operation. This is performed via a call/return operations to state *SCATTER* or *GATTER*, which both return to state *N0*. The ASM's flow, as well as the authorization of data transfers over the blocking connections, will take into account that each source and sink may not be ready to produce or accept a data token at each clock cycle, depending on their synchronization protocols.

```
 1: asm simpleScatterGather {
 2:     switch( state )
 3:     case N0:
 4:         cmdReg = cmdFifo;
 5:         goto N1;
 6:     case N1:
 7:         if( cmdReg == 0 );
 8:             call SCATTER; return N0;
 9:         else;
10:             call GATHER; return N0;
11:         end;
12:     case SCATTER:
13:         A1 ={ta1} chA;
14:         A2 ={ta2} chA;
15:         ta1 < ta2;
16:         return;
17:     case GATHER:
18:         chB ={tb1} B1;
19:         chB ={tb2} B2;
20:         tb2 => !tb1.rtf;
21:         return;
22:     end;
23: }
```

Fig. 2. ASM description example featuring blocking and non-blocking connections, data transfer authorization rules, *goto* statements, as well as state *call* and *return*.

3.3. Transfer authorization rules

Each connection has several attributes and can be labeled, to be further referenced in rules. For instance, the attributes *rtf* (ready to fire) can be used to determine if both source and sink are ready. The attribute *fire* is true when the transfer is actually authorized at the current clock cycle. The attribute *done* is used to determine if a transfer has already occurred during the current activation state. Rules are based on the logical implication operator '=>'. Nevertheless, some shortcuts are supported. Figure 2 illustrates a few uses of this semantics.

Statements 13-15 describe an in-order 2-data scatter operation from source *chA* to sinks *A1* and *A2*. Sequencing is enforced by the rule at statement 16, restricting that the con-

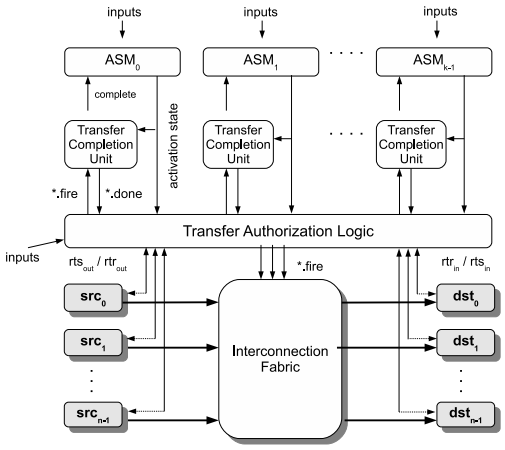

Fig. 3. Target implementation architecture for a CASM device description featuring 1 to k concurrent ASMs.

nection $ta1$ is done before $ta2$. It is equivalent to *ta2=>ta1 .done*. The statements 18-20 describe a prioritized gather operation. Statement 20 means that transfer *tb2* is allowed only if transfer *tb1* is not ready to fire, which means that its priority is lower than transfer *tb1*.

The main motivation behind the use of such constraint programming paradigm is that it lets the designer specify precisely and concisely the required behavior in a declarative fashion. A compiler can then automatically generate a control circuit that will conform to the requirements for any possible combination of i/o synchronization signals.

3.4. Hardware implementation

The target implementation for devices described at the proposed level of abstraction is presented in Figure 3. It contains optimized transfer authorization logic (TAL), an interconnection fabric, as well as one transfer completion logic (TCL) unit per ASM. The transfer completion unit handles the completion of single data transfer blocking connections, and contains a one bit-register *done* register for each of these connections. The interconnection fabric is responsible for establishing the point-to-point connections, between sources and sinks, over which data transfers are authorized at each clock cycle by the TAL.

4. HARDWARE DESCRIPTION OF THE QUICKSORT ALGORITHM

4.1. Simple hardware implementation and description

We start with a single ASM description of the quicksort algorithm using the proposed hardware description language, exposing how it facilitates the manipulation of data tokens transfers involving pipelined devices. Figure 4 shows the hardware description, contained within two states, of the

Algorithm 1 partition(mem, left, right, pivotIndex)

```
1:  storeIndex = left;
2:  pivotValue = mem[pivotIndex];
3:  swap mem[pivotIndex] and mem[right];
4:  for i = left → right − 1 do
5:      r1 = mem[i];
6:      if r1 < pivotValue then
7:          mem[i] = mem[storeIndex];
8:          mem[storeIndex] = r1;
9:          storeIndex++;
10:     end if
11: end for
12: swap mem[pivotIndex] and mem[right];
13: return storeIndex;
```

partition algorithm's *for*-loop. The in-place partition algorithm is illustrated in Algorithm 1, where the inner-loop has been detailed for a more direct correspondence to the hardware description of this region of interest. The state $QSPART0$ realises the test required by the for-loop, the memory read operation, and then compares the read value with the pivot's value, corresponding to statements 4-6 of Algorithm 1. Depending on the result of the comparison with the pivot, the state $QSPART1$ may perform the memory swap and increment the $storeIndex$ register. Both of these states are multi-cycle instructions. In this case, $QSPART0$ requires 3 cycles to complete, one cycle to initiate the read, and then two cycles wait are required before the comparison can be initiated with the read data. The state $QSPART1$ will require 3-cycles to complete as well, two cycles being required for reading $mem[storeIndex]$ and one cycle to perform the write operation at $mem[i]$. The write operation at $mem[storeIndex]$ is performed at the second clock cycle.

4.2. Functional pipeline hardware implementation and description

The proposed functional pipeline hardware implementation decomposes quicksort's partition algorithm into two concurrent ASMs: one responsible of reading all $mem[i]$ of a partition and of initiating all the comparisons, while the other is responsible of performing the required swaps. This is achieved in hardware by sharing a dual-port RAM memory between two concurrent ASMs using arbitration rules. Due to a lack of space, the detail of this implementation is not presented.

5. COMPILATION AND SYNTHESIS RESULTS

5.1. Simulation and synthesis

Both simple and pipelined implementations of the quicksort hardware descriptions have been compiled to RTL using our compiler, and then synthesized with Altera's Quartus II tool targeting a Stratix-III FPGA. Full-functionality

```
 1: asm quicksort
 2:     switch( state )
 3:     //...
 4:     case QSPART0:
 5:       if(i < imax)
 6:           mem0.addr ={t1a} i;
 7:           r1 ={t2} mem0.dout;
 8:           compareL.a ={t2a} mem0.dout;
 9:           compareL.b ={t2b} pivotValue;
10:           goto QSPART1;
11:       else
12:           goto N9;
13:     case QSPART1:
14:       if(compareL.lt.dout.main == 1
15:        and compareL.lt.dout.sync.rts == 1)
16:           mem0.addr ={t1ar} storeIndex;
17:           mem0.addr ={t2aw} storeIndex;
18:           mem0.din ={t2} r1;
19:           storeIndex ={t2s} storeIndex + 1;
20:           mem0.addr ={t3aw} i;
21:           mem0.din ={t3} mem0.dout;
22:           i ={t3i} i+1;
23:           t2 <=> t2aw; t2aw <=> t2s;
24:           t3 <=> t3aw; t3aw <=> t3i;
25:           t3 > t2 > t1ar;
26:           goto QSPART0;
27:       else
28:           if(compareL.lt.dout.sync.rts == 0)
29:               goto QSPART1;
30:           else
31:               i ={t1} i+1;
32:               goto QSPART0;
33:           end;
34:       end;
35:     //...
```

Fig. 4. CASM description of the for-loop performing the in-place partition within 2 states.

has been validated through RTL simulation. Performance results are obtained by sorting (in ascending order) an array of 256 32-bit words which is initialized with a descending order sequence. Both designs, simple and functional pipeline, can operate at maximal frequencies of 325 MHz and 318 MHz, and execute in 16.2k and 12.8k cycles respectively. The simple implementation requires 538 ALUTs and 356 registers, while the functional pipeline implementation requires 948 ALUTs and 1046 registers.

5.2. Discussion

Both quicksort architectures are fully-functional, demonstrating how the proposed HDL facilitates the description of control-dominant applications. From the synthesis results obtained, it is observed that a 1.3× speed-up factor is obtained with a 2.2× hardware cost increase. Nevertheless, despite some significant differences in both architectures, they both enable solid maximal operating frequencies above 300 MHz, while the control logic hardware cost increase is 1.1×.

6. CONCLUSION

Proposing new hardware abstractions for high-level synthesis of digital designs is a real challenge. While many previous works have focused on C/C++ like languages and the automated synthesis of optimized datapaths, this work has focused on control-dominant applications. Combining a high-level FSM programming paradigm with the presented constraint programming paradigm, one can describe digital designs as concurrent dynamically reconfigurable data-flow graphs that can implement complex multi-cycle instructions, involving multiple concurrent operations. A control dominant algorithm (QuickSort) has been described and compiled with our compiler. Results demonstrate full functionality at rather high frequencies.

7. REFERENCES

[1] P. Alfke, I. Bolsens, B. Carter, M. Santarini, and S. Trimberger, "It's an FPGA!" *Solid-State Circuits Magazine, IEEE*, vol. 3, no. 4, pp. 15 –20, fall 2011.

[2] (2009) International Technology Roadmap for Semiconductors (ITRS) Report. [Online]. Available: http://www.itrs.net/Links/2009ITRS/Home2009.htm

[3] E. Ogoubi and J. David, "Automatic synthesis from high level asm to vhdl: a case study," in *Circuits and Systems, 2004. NEWCAS 2004. The 2nd Annual IEEE Northeast Workshop on*, june 2004, pp. 81 – 84.

[4] (2010) SystemC Modeling, Synthesis, and Verification in Catapult C. [Online]. Available: http://www.mentor.com

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473 – 491, april 2011.

[6] S. Sarkar, S. Dabral, P. Tiwari, and R. Mitra, "Lessons and experiences with high-level synthesis," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 34 –45, july-aug. 2009.

[7] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 18 –25, july-aug. 2009.

[8] S. Edwards, "The challenges of synthesizing hardware from c-like languages," *Design Test of Computers, IEEE*, vol. 23, no. 5, pp. 375 –386, may 2006.

[9] D. L. Rosenband, "Hardware synthesis from guarded atomic actions with performance specifications," in *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, ser. ICCAD '05, 2005, pp. 784–791.