Hardware Compilation for Software Engineers: an ATM Example

M. Fleury, R. P. Self, and A. C. Downton

Department of Electronic Systems Engineering,

University of Essex, Wivenhoe Park,

Colchester, CO4 4SQ, U.K

tel: +44 - 1206 - 872817

fax: +44 - 1206 - 872900

e-mail fleum@essex.ac.uk

Short title: Hardware Compilation for Software Engineers

Abstract

Forthcoming technology such as single-chip RISC/FPGA combinations make hardware compilation, fast prototyping, and FPGA replacement of ASICs all more likely. FPGAs have made a software-oriented approach to digital design feasible. This paper reviews remaining obstacles to this approach. The trade-offs between use of an HDL and a 'C'-variant, Handel-C, for logic synthesis are considered particularly in regard to programmability and the overall design process. A simple example in a likely application area, simulation/emulation of telecommunications switches, serves to illustrate the analysis.

1 Introduction

A shortage of hardware chip designers alongside a reduction of product life-time to within six months makes it worth considering whether software engineers, with limited background knowledge of the underlying hardware, could still participate in the high-level design process. For general-purpose computers the answer of course is yes, but for the hardware family that originated in the programmable logic device (PLD) [1] and has now encompassed complex field programmable gate arrays (FPGAs) [2] the answer has until recently been no, principally because current software CAD tools do not sufficiently abstract from the hardware. FPGAs were originally thought of simply as weakly-programmable 'glue' logic, and CAD tools were designed bottom-up to support the hardware design process. Currently available devices have capacities of the order of a million system gates [3] however, and thus are potentially capable of directly implementing significant embedded applications. Therefore, research into hardware or silicon compilers which will seamlessly compile an algorithmic description directly into a layout [4] remains important, particularly if the compiler is compatible with existing software engineering practice, and can integrate into the high-level design processes which commence prior to the coding stage. Hardware compilation has the potential to remove an intermediate level of circuit design, represented by hardware languages such as VHDL and Verilog, in favour of abstracting away from low-level circuit implementation.

Preserving a vertical design process allows a seamless progression [5] from specification to implementation. Lower levels of FPGA design are largely automated so it is in the transition from specification to HDL program where the weakness lies and where it is possible to iterate between specification and circuit components description (netlist) if C were to be used. While traditional techniques of Boolean algebra optimizing combinational logic can still take place, there is the possibility of first optimizing the specification by compiler for reactive programs, by reduction to a normal form [6]. All the higher-level parts of an FPGA design prototype [7] might be accomplished by software engineers with the design constrained by cycle-accurate simulation before passing to the automated parts.

The term 'prototyping' is not yet fixed in the computer science community. Some understand the term to imply development of a 'quick and dirty' solution to gain insight or as a feasibility study, and thereafter start again from scratch for the 'real thing'. A newer interpretation is the initial development of a core (the prototype), which, thereafter, is extended with secondary features, but always remains part of the entire system. FPGAs embody useful aspects of the former hardware-oriented interpretation (as FPGAs can model circuitry which at a later stage may be transferred to a chip), while retaining scope for the latter interpretation, hence supporting reuse of the software prototype.

To test the aforementioned hypothesis, a prototype Asynchronous Transfer Mode (ATM) switch originally designed in VHDL has been recoded through a C-variant, Handel [8], which includes additions and subtractions from true C. Work reported in this paper is not connected with the originators of Handel or the later Handel-C and constitutes a user test from a software designer's perspective. Though a commercial product, Handel-C v. 2.1 [9], is close in time to its academic precursor. Later versions of Handel-C may be constrained by the exigencies of the marketplace.

2 Background

2.1 FPGAs

The dichotomy between the ready programmability of general-purpose processors and the relative inaccessibility of other programmable devices has been highlighted by the recent development on the same silicon die of RISC processor/FPGA combination chips, for example Chameleon Systems CS2112 [10], intended for protocol processing on mobile stations, or the NAPA100 from National Semiconductor [11], aimed at applications in aerospace. Current clock speeds for isolated FPGAs of 100-300 MHz limit their competitiveness compared with application specific integrated circuits (ASICs) in the high-volume consumer marketplace. However, for small volume applications and prototyping, a low-cost FPGA may be the preferred solution. For example, FPGAs are attractive in large-scale target-tracking radar systems. The FPGA employs fine-grained parallelism which complements the RISC and obviates the slower clock speed (rather as the SIMD DAP was used as an accelerator to ICL mainframes [12, pp. 290-301]) and also has the advantage of offering efficient, variable data widths [13], unlike word-oriented general-purpose processors.

There are two main types of FPGA architecture [7, 14]. One is based on look-up tables (LUTs), and the other is based on multiplexers. LUT devices implemented as static RAM (SRAM) [15] are particularly favourable for dynamic reconfigurability as a bitstream can be injected onto the FPGA from the host processor. Handel-C is compatible[1] with older, low-cost Xilinx devices and the newer, high-density devices,

[1]Typically, some settings are made in a header file for particular devices. While Handel-C outputs in Xilinx proprietary netlist format, it can also output in EDIF (Electronic Design Interchange Format), the industry-standard format.

including the Virtex family,[2] as well as devices from Altera.[3], which are also all SRAM-based devices.[4] Xilinx devices have a low-skew, global clock which matches the completely synchronous Handel-C model. (Xilinx products also include the multiplexer-oriented 6200.)

Xilinx LUT-based FPGAs contain an array of Configurable Logic Blocks (CLBs) connected to routing wires, which in turn are connected to other routing wires by 'switch boxes'. Each CLB (e.g. 400 on the Xilinx XC 4010, 6144 on the Virtex XCV1000) can perform any combinational logic function with K inputs, e.g. $K = 3, 4, 5$ for Xilinx 3000, 4000, 5000 series respectively, and 4 inputs on the XCV1000. An important determinant for the scaling performance of an FPGA is the number of I/O pins (e.g. 160 in the XC 4010, 512 on the XCV1000), as it is less easy to increase the number of I/O pads than it is to increase the number of transistors. Though Handel-C's current simulator assumes a single clock, two or more internal clocks occur in some FPGAs. It is important to note that many real-world systems have different external clocks, internal clock multipliers and so on. Therefore, Handel-C's current limitation to a single clock would make such systems difficult to design.

The output of hardware compilation only establishes whether there are enough components on the device to support the design. Xilinx supply a toolkit, which, after a number of transformations, performs a min-cut algorithm to partition the netlist (technology mapping) into CLBs, for which Handel-C outputs guidance instructions. Automatic place-and-route (APR) of the CLBs on the FPGA is by means of a simulated annealing algorithm, though again there are other algorithmic possibilities [15]. Finally, the routed array is converted into a bitstream to program the device.

2.2 Approaches to design for FPGAs

The early low-level approach to behavioural modeling of FPGAS was either to draw a schematic diagram with a package such as OrCAD, Daisy or Mentor, or to output a set of Boolean expressions or state machines via a language such as PLDesigner, ABEL or PALASM. Unfortunately, low-level designs cannot be easily changed or adapted whereas making changes to high-level designs requires far less effort, hence the popularity of algorithm simulation in software. Subsequently [16], four techniques more suitable for a software-oriented

[2]Specifically, Handel-C is compatible with the Xilinx LUT-based 3000 and 4000 series up to the 4000XV. There is also a version of Handel-C compatible with a Xilinx Virtex FPGA on the ADC-RC1000 board with a Pentium host.

[3]Specifically, Handel-C is compatible with Altera 6K, 8K, and 10K devices. The recent Altera 20K will also no doubt be compatible. Some Altera devices are eprom-based, non-volatile but of lower gate capacity.

[4]AT&T's (Lucent) ORCA series is another well-known LUT-based (SRAM) family of stand-alone FPGAs.

approach have been developed:

1. *Dataflow graphs* as in Sehwa [17], MAHA [18], HAL [19], and the Pangrle system [4].

2. *High-level languages*, for example Pascal [20], Ada [21].

3. *Enhanced versions of existing languages* such as HandelC, HardwareC [22] or HardwarePal [23].

4. *Hardware description languages* such as ISPS [24], CADDY/DSL [25], Verilog [26], or VHDL [27].

Dataflow diagrams may well be suitable for hardware-independent, conceptual descriptions of a design, and notably are now a feature of the Ptolemy open modeling environment [28]. It is unclear whether a dataflow diagram is in a form suitable for direct input to the synthesis stage. On the other hand, some languages like HardwareC remain strongly linked to the synthesis stage providing vertical integration through the design process. The same can be said of Handel-C except that unlike the independent set of synthesis tools, Hercule and Hebe, provided by HardwareC for circuit construction, Handel-C relies on the Xilinx tools for FPGA design.

Conventional procedural languages may be an insufficient solution for FPGAs through lack of any or all of the following features:

- parallelism (or concurrency emulating parallelism) which is the essential distinguishing feature of hardware. Parallelism may be at the statement level or at the function or procedure level;

- timing references to constrain the normal cycle-accurate simulation process;

- strong typing as conversion between hardware objects is not physically possible; and

- special data structures such as variable data-widths.

For example, HardwareC, which has similarities with Handel-C, assumes that all procedures run in parallel, with synchronization and communication by message passing. Tags constrain the sequencing of events in terms of clock cycles. Some existing languages have many of the desired features. For example, Ada has concurrency with rendezvous synchronization and communication; strong typing; some user-defined data-types; and some ability to add timing constraints. However, Ada does not have as large a pool of programmers as C.

3 Hardware Compilation from VHDL and C/C++

3.1 VHDL language issues

VHDL is a language, strongly influenced by Ada and Modula-2, originally intended for hardware documentation and modeling before circuit synthesis components were added, whereupon due to growing popularity it was standardized in 1987. When used at the requisite level of abstraction VHDL is indeed similar in structure to Ada, implying that, as with any software language, there should be no barrier to the competent software engineer. However, it remains the case that Ada is not C, and perhaps one should look to the origins of Ada [29] to explain the dichotomy in popularity. VHDL was not primarily intended for hardware compilation but dates back to very early HDLs such as PMS [30]. One consequence of this history is that VHDL encourages the designer to construct a hierarchy of structures or entities, strongly organizing the design. Though VHDL's hierarchical structure may suggest object-orientation, VHDL lacks inheritance, a key determinant of the object model. In fact, there is an IEEE working party considering two alternative object-oriented extensions to VHDL, SUAVE and Objective VHDL [31].

Like Ada, VHDL separates interface (`entity` in VHDL) from implementation (`architecture` in VHDL). Given that this is the same clear division of function as in Java interfaces and implementations, CORBA IDL mapped to C++, C++ abstract and concrete classes, or SML signatures and structures, there is no reason why the structure of VHDL should present a barrier to a software engineer.

A `process` within an `architecture` block is concurrent with execution stimulated or sensitive to change in a signal.[5] In fact, hardware is not just concurrent, i.e. there is a scheduled interleaving of threads of control, but parallel, which is the main conceptual leap from the stored-program model familiar from procedural languages. Java has been able to introduce multithreading (concurrency) within a C-like language. Again, the look-and-feel of VHDL is more akin to Modula-2 and thence Pascal, a language that whatever its merits has been taken up predominantly in academia (with an exception in Borland's Delphi).

[5]There is also, in VHDL, a statement-level form of concurrency called continuous assignment, again reactive to a change in circuit state.

3.2 VHDL implementation issues

VHDL does not completely succeed in abstracting from hardware and indeed requires a knowledge of circuitry. VHDL talks of components, sensitivity to signals, and ports. There is also a timing model which specifies the time allocated to statements when simulating code. However, VHDL is a generalized approach and therefore also has features that are inappropriate to a particular hardware type. For example, the standard logic type accepts nine different logic levels. VHDL also includes features such as integer division for which it is not normally possible to synthesize a circuit, and floating-point numbers and multi-dimensional arrays which would result in inefficient implementations. Conversely, it can be argued that these features of VHDL are present to allow high-level architectures to be created in abstract terms, and to simplify the development of testbenches.

Normally, more detailed descriptions in the register-transfer level (RTL) of VHDL are instantiated as components at the structural level. One approach to VHDL's over-rich expressiveness is to write only in structured VHDL whereby components from a pre-written library are selected from within the code. In particular, at RTL a restricted set of VHDL constructs can be employed to suit a particular synthesizer toolkit. Thus, in practice, VHDL introduces a multi-stepped programming environment.

Figure 1 illustrates the VHDL development process. The first step is to define a suitable architecture from the system specification. This results in a set of functionally independent sub-systems, capable of being coded and tested as separate units.

Code development consists of writing two modules; the first implements system functionality, while the other defines a test bench with which to test the system functions. Test bench code exercises system inputs during simulation and thus defines a baseline for validation of both behavioural (structural) and RTL implementations.

While VHDL development follows the code, compile, execute cycle common with software development, the runtime environment is a hardware simulator instead of a host microprocessor. VHDL code once checked for syntax and type integrity is compiled into a repository of hardware entities. Simulation consists of executing the top-level hardware entity (the test bench) and saving the time ordered sequence of events and outputs generated by test bench stimuli. Subsequent analysis and verification is facilitated by a variety of waveform viewers and output capture tools. However, it is essential to check correct working by means of runtime capture and verification routines coded into test bench modules, as visual checking of waveform displays is insufficient.

Once simulation has verified the functional correctness of VHDL source, the next phase is Logic Synthesis. Historically, synthesis was the step during development concerned with translating the top-level design into a schematic diagram of gates and flip-flops. Software synthesis tools now automate this process by compiling source code into a netlist file of gates and gate interconnections. The final stage involves passing the netlist into a layout tool, which is responsible for mapping logic onto device resources, as is discussed in Section 2.1.

3.3 C/C++

Some hardware designers have recognized that high-level behavioural analysis may be better served by means of the 'C' or C++ programming languages than HDLs (hardware description languages) such as VHDL and Verilog. The design of the Neon 256-bit 3D Graphics Accelerator [32] employed C++ templates for bit-width specification, 'C' mathematical libraries, and a fast cycle-accurate simulator written in 'C'. A C-to-Verilog translator enabled synthesis (the process of checking the timing of circuitry by mapping the design to the intended technology) to take place. However, this presupposes that the designer is hardware-aware so that features of 'C' with no obvious counterpart in hardware, such as recursion, are avoided.

At a more subtle level, synchronous variable assignments require a delay of one clock cycle for a value to be latched in. Therefore, to synthesis software, 'C' statements such as $a = x++$ imply a two-cycle assignment to x of $x+1$ and then assignment to a, whereas expressions such as $a = x+1$ imply a single step assignment without latching. The former is not permitted in Handel-C because it does not have the form *variable = expression*.[6] Note also that assignment of complex expressions, though legal, have the effect of extending the clock width, which is always the time to execute the most complex expression in a program.

A principal advantage of hardware compilation is the flexibility it offers to run different operations in parallel, but this necessitates thread libraries and support for concurrency in the high-level algorithmic specification, and some way of extracting synchronous behaviour when subsequent cycle-level simulation takes place. To enable a software-oriented approach, some features of 'C' must therefore be removed, but there is also a need to provide additional structure. However, modification of C/C++ results in lack of portability, and means that existing auxiliary tools such as debuggers become unavailable until there has been time to construct replacements. VHDL and Handel-C both rely on the programmer to explicitly

[6]Handel does not permit the ++ construct. Handel-C permits statements but not expressions of the form $x++$.

identify parallelism, whereas automatic compiler identification is possible within software, particularly in regular, data-parallel algorithms [33]. A scheme is presented in [34] to extend the Handel-C compiler by means of the SUIF compiler to extract loop and pipelined parallelism. An intermediate approach is to make use of program annotations, or compiler hints. For example, the GARP compiler [35] embeds compiler `pragmas` into 'C'.

4 Hardware compilation from Handel

4.1 Handel and occam

While Handel has the 'look-and-feel' of 'C', Handel [36] owes its origin to the parallel programming language occam. Put another way, Handel is a strongly-typed variant of 'C' with additional parallel constructs. Unlike occam [37], where synchronization is solely at `rendezvous` points through channels, Handel instructions execute in lock-step synchronization, as befits hardware emulation. The idiosyncratic features of occam such as closed format and strict precedence enforcement have been removed, and sequential statements are the default. Earlier academic versions of Handel constructed programs through a metaprogram written in SML (Standard Meta Language) [38], a functional language, while Handel-C has been implemented to industry strength [9].[7]

Handel gains from the occam maxim of 'more in simplicity'. Like occam and the transputer, Handel achieves some of its simplicity by being targeted towards one device, in the way that other hardware compilers have a restricted explicit [39] or implicit target. Handel follows a policy of restricting the options available to match what is feasible on an FPGA. For example, Handel has integer and byte (`char`) types but no floating point or complex types. There is also no integer divide operation. The width of integer types is assigned or inferred but cannot be left open. Incompatible types (because the data width does not match) are resolved by a concatenation operator which can extend widths.

A summary of the relationship between occam and Handel-C is contained in Table 1. Parallel statements (fine-grained) or functions (medium-grained) are grouped by a `par` construct with synchronization at an implicit barrier (represented by an AND logic gate). Handel also has a variant of alternation, whereby guards

[7] We used Handel and Handel-C for compilation and simulation, and Handel for synthesis. We did not use the earliest version of Handel, which is closer to occam than it is to 'C'.

[40] are activated by communication events (including, unlike occam, output communication[8]). Communication and synchronization is by channels through `rendezvous`. A delay statement, somewhat equivalent to the occam `skip`, causes a one cycle delay to prevent access to a shared resource or to break combinational logic cycles. ROM and RAM types can also only be accessed sequentially. It must be added that the Handel-C version of Handel has found it necessary to allow hardware-specific tags to be added which add timing characteristics to some objects, particularly when external to the FPGA.

Handel-C does not separate the declaration of an interface and the definition of an associated implementation. If Handel-C code could be treated in this way, as a black-box with an external interface, then a Handel implementation could be treated as an object. The advantage would be that a Handel-C object could be treated as any other object within a software/hardware codesign framework. It will frequently be the case that a Handel-C FPGA implementation will form one part of a system (for an example see the MP3 port[42]). It will be the role of codesign to arrange compatible interfaces between the modules within the design, ensuring that performance goals are met. The details of the implementation will be less relevant for the higher-level design. Finding a way to extend Handel-C in this direction is probably a significant area for investigation.

Handel-C also provides a macro facility which allows composite circuit elements to be reused within the program text. As is normal with macros, parameters are not typed as they would be if procedures had been used, but otherwise reuse is now possible.[9] However, the software engineer should be aware that the same physical component cannot be expected to run in parallel. There is also a finite limit to the available FPGA hardware implied by repetition of circuitry.[10] It is however specifically because an FPGA can have (say) multiple arithmetic logic units (ALUs) that dedicated FPGA implementations at lower clock-speeds can out-perform conventional general-purpose microprocessors and DSPs.

Though macros have been introduced to support reuse, which is important to avoid tedious repetition

[8]Output guards were found to be to difficult to implement in the transputer microcode implementation of occam [41], though appearing to be necessary for reasons of consistency. This restriction does not extend to a language implemented by logic gates as a ready line can be selected and tested from either end of the channel circuitry. To avoid connecting an input guarded statement to an output guarded statement, Handel requires that a channel can only appear once in a set of guards, which is not the case in occam. Handel's guards are examined in priority order.

[9]In the Handel version, macros were not parameterized.

[10]However, the Virtex-E (1.8V) XCV32000 FPGA from Xilinx [43] is to have 4M system gates in comparison to (say) the earlier Xilinx 3090 that has 9000 gates.

of code, Handel-C lacks the ability to group objects in a hierarchy. It also currently lacks the equivalent of VHDL's `generate` construct; hence, it is necessary to cut-and-paste code when building up the switch layout of the example given in Section 5. VHDL's `generate` construct is similar to occam replicators, which allow a parallel process to be defined once and then repeatedly instantiated, each process indexed by a loop counter. Differences between VHDL and Handel-C have been summarized in Table 2.

Handel-C's resemblance to 'C' enables semi-automatic porting from 'C'. Table 3 [9] identifies some of the changes required. Handel-C has additional occam-like constructs and statements specific to FPGAs, while 'C' has additional complex data types. Large (*circa* 10,000 lines of code) 'C' applications such as IP telephony and MP3 have been ported to Handel-C.[11]

4.2 Handel and CSP

Though Handel resembles occam in a 'C' wrapper, in fact it is better described as an implementation of the specification language Communicating Sequential Processes (CSP)[44]. A potential value of a process algebra is that simplifications can be made by virtue of known algebraic laws. As such, the Handel model is more closely defined than some competitors. However, CSP lacks reference to a global clock and in fact there has been some interest in translating CSP into delay-insensitive VLSI circuits e.g. [45]. Timed-CSP [46] is continuous in time, implying that still some other variant of CSP is needed for modeling clock synchronous processes. Another weakness of CSP is that the notation for modeling data state is rudimentary, which timed communicating object-Z (TCOZ) [47] may remedy. Both timed-C and the related TCOZ can be represented in graphical form, which would be the equivalent of dataflow diagrams for Handel. Another possibility, which illustrates the need to consider representation above the language level, is to convert directly from a process algebra [48] to a netlist. The practical consideration with all such attempts is that process algebras are not well-known in the wider programming community.

4.3 Handel development process

The Handel hardware compiler itself produces a netlist so does not require (say) translation from C to VHDL. The netlist represents pre-defined combinational circuit elements onto which elements of the language

[11]MP3 is the standard audio codec. It is reported [42] that the core code was ported on a line-by-line basis to Handel-C after conversion from floating point form. Numerous LUTs were extracted and placed in various ROMs and RAMs attached to the FPGAs.

are mapped [49]. Were the output of the Handel compiler to be translated to VHDL then the resulting circuit elements after synthesis would not be predictable, breaking the 1:1 mapping from specification to implementation.[12] Because Handel is carefully structured, all elements of the language map directly onto corresponding circuit elements, unlike accelerating compilers [35] which only map designated parts of a program.

Figure 2 identifies the major parts of the development process, and shows the flow of data between system components. Development involves three distinct phases:(1) hardware compilation,(2)logic synthesis, and (3)host compilation. During the first phase, Handel source is compiled into logic elements. The compiler output consists of netlist symbols, and XACT editor macros. The second phase is the place-and-route stage of logic synthesis; it is responsible for converting netlist symbols into ASCII FPGA configuration data. Communication with the FPGA is via an Application Programming Interface (API) from the Pentium PC host. The configuration bitstream is loaded into the FPGA from a file via the API. In the final phase, the host source code is compiled and linked. For a Pentium host, the process is simplified as the source code once compiled is executed, not downloaded as a bootable file. For embedded work as herein, the host acts as an interface to the FPGA but equally the FPGA could act as a coprocessor in software acceleration mode. As a whole the organization of communication with the FPGA can be likened to Remote Procedure Call (RPC), in which the channel interface code on the host takes the place of a communication stub.

Simulation is enabled by default during hardware compilation. The simulator reports the quantity of logic generated, and redisplays the values of all program variables at the end of each clock cycle. Special channel components temporarily replace standard FPGA channel logic in order to generate user prompts and inputs. The simulation environment is not as extensive as VHDL, and in particular, Handel-C lacks the equivalent of a testbench. While Handel-C remains targeted at Xilinx and Altera FPGAs support for other devices may not be needed. Automatic translation from Handel-C to VHDL would make the simulation facilities of VHDL available at a cost in design integrity.

5 ATM switch prototyping

FPGAs can potentially provide high throughput for ATM switches. A large variety of multi-stage interconnection networks (MIN) for ATM switches exist [50]. A non-blocking Benes switch, which is formed from

[12]This presupposes that the compiler's integrity is also checked, as has taken place when the compiler was commissioned.

two Baseline networks, Fig. 3, has been prototyped in VHDL and Handel-C. The Benes network is by no means the only MIN one might want to prototype. For example, by means of a sort network, internal switch conflicts within a Banyan switch can be resolved, provided output destinations are unique.

Hardware prototyping/simulation can model the potential throughput and latency experienced by data cells passing through networks of switches under a variety of traffic sources, including long-tailed distributions. Accelerated simulations or parallel simulations in software or both have generally been applied to single switches with limited inputs. As ATM protocols do not employ flow control, cell loss is possible, though this must be kept very low (typically 1 cell in 10^9) to match the low losses on optical fibres, and to avoid retransmission of video frames, implying lengthy software simulations. The effect of signaling to reduce congestion under ATM best-effort policies is also of interest.

The FAST testbed [51] is a hardware simulator based on 13 Altera FPGAs per board. The FAST design is intended for modeling other designs apart from multi-stage such as crossbar. However, ATM switch prototypes for the FAST system are designed using VHDL, which restricts the usage of such a testbed, as by definition it should be user extensible. The Atlas 1 is an example ASIC 16×16 switch [52] that can be combined in modular manner to form a larger switch, of the type that could be modeled or even replaced by an FPGA.

The topologies associated with ATM switches recur in dense wave-length division multiplexing (WDM) optical switches. Internet protocol (IP) routing has recently turned to Multi-Protocol Label Switching (MPLS) to circumvent slow software-based searches of routing tables, again implying ATM-like behaviour at the data link level. Though some researchers will be interested in replacing ASICs with FPGAs in ATM switch designs, the direction of our research is to construct a flexible simulator for a variety of ATM switch configurations. To this end, Handel-C seems ideally suited as the design can be altered almost at will.

5.1 VHDL version

An outline VHDL version of a Banyan switch has already been published [53]. The 16×16 switch implementation utilised an Actel A54SX08-3 (8,000 logic gates[13]) device and required a third of the resources; it operated at 250 MHz, giving a 4 Gbps throughput. A more detailed VHDL design of an ATM switch

[13]Logic gates are distinguished from system gates as these are the gates that can be directly used for an application. Unlike ASICs, the gate count for an FPGA is notional, depending on the efficiency of the mapping for a particular application to a particular technology.

emulator, which includes modeling of routing control, can be found in [54].

Figure 3 shows the intended layout for the switch, and it is perhaps clear how this might map onto an FPGA. However, it is important to realize that this layout is in fact a behavioural model, so that in the case of a more complex behavioural design the mapping would not be at all obvious and indeed would not be expected to correspond directly to the behavioural model.

The following RTL code (reproduced from [53]) defines a single switching element. A switch element, Fig. 4, has two settings, straight-through or cross-switched. The switching-element process is sensitive to the rising edge of the circuit clock signal. The value of the input `SW` determines whether inputs from `A` and `B` are passed through or crossed over.

```
entity Element is
    port(SW: in STD_LOGIC;
    A: in STD_LOGIC;
    B: in STD_LOGIC;
    CLK:in STD_LOGIC;
    Y0:out STD_LOGIC;
    Y1:out STD_LOGIC);
end Element;

architecture Behaviour of Element is begin
 process(CLK)
 begin
 if CLK'event and CLK='1' then
    case SW is
     when '0' =>
       Y0 <= A;
         Y1 <= B;
     when '1' =>
       Y0 <= B;
         Y1 <= A;
```

```
        when others =>
          null;
      end case;
  end if;
end process;
end;
```

The switch state is set up in the following RTL code which converts a 64-bit serial input from `shiftin` to parallel form. The intention is that once all the bits are established in a register then the output will be latched into the 64 switch elements. It is clear that the design of the shift register strongly depends on knowledge of digital circuitry rather than simply understanding a more abstract software shift operation. Thus, the `&` operator concatenates an input to an existing vector before the shift.

```
entity Shiftreg is
   port(ShiftEnable: in STD_LOGIC;
        Shiftin: in STD_LOGIC;
        Aclr   : in STD_LOGIC;
        Clock  : in STD_LOGIC;
        Q      : out STD_LOGIC_VECTOR(63 downto 0));
end Shiftreg;

architecture Behaviour of Shiftreg is
  signal Qaux: STD_LOGIC_VECTOR(63 downto 0);
begin
 process (Clock, Aclr)
 begin
   if (Aclr='0') then
      Qaux <= CONV_STD_LOGIC(0, 64);
   elsif(Clock'event and Clock='1') then
      if (ShiftEnable='1') then
         Qaux <= Qaux(62 downto 0)& Shiftin;
      end if;
   end if;
 end process;
 Q <= Qaux;
end;
```

The following structural level code sets up one column of eight switch elements by repeatedly generating switch elements. The elements of the stage then operate in parallel lock-step.

```
entity Stage is
  port(SW: in STD_LOGIC_VECTOR(7 downto 0);
  A   : in STD_LOGIC_VECTOR(7 downto 0);
  B   : in STD_LOGIC_VECTOR(7 downto 0);
  CLK: in STD_LOGIC;
  Y0 : in STD_LOGIC_VECTOR(7 downto 0);
  Y1 : out STD_LOGIC_VECTOR(7 downto 0);
end;


architecture Behaviour of Stage is
  component Element
    port(SW: in STD_LOGIC;
    A   : in STD_LOGIC;
    B   : in STD_LOGIC;
    CLK: in STD_LOGIC;
    Y0 : out STD_LOGIC;
    Y1 : out STD_LOGIC;
  end component;
begin
  G1: for i in 0 to 7 generate
    C1: Element port map (SW(i), A(i), B(i), CLK, Y0(i), Y1(i);
  end generate G1;
```

```
end;
```

5.2 Handel-C version

Our Handel-C implementation of the same ATM switch provides an input buffer stage, seven 16-port switching stages, and an output buffer stage, which is the equivalent of the VHDL version. Fig. 5 shows a simplified version (to avoid over-complicating the diagram) with just five of the stages and eight inputs. New data samples were fed to the switching fabric at the beginning of each clock cycle. As in the VHDL version the switching configuration is a bitstream, which is intended to be latched into each switching element. The initial implementation required two clock cycles to process each data sample at a switching element because switching elements worked by processing input on the first cycle and writing output on the second cycle. Input processing is suspended through the blocking semantics of Handel-C channels.

In a second design, the input and output ports were modeled as separate processes with shared access to data buffers, updating their buffers and output ports on each cycle. Although parallel access to shared data violates normal concurrency rules, in practice the synchronous discrete-time behaviour of the underlying hardware ensures that switch element I/O operations are sequential over the duration of a single clock cycle. Under the synchronous model, input data is processed via combinational logic from the beginning of a clock cycle and is written to registers (switch element data buffers in the ATM switch) at the beginning of the next clock cycle. This design for the full version, when implemented on a Xilinx XC3195A FPGA, took up 1103 gates, corresponding to 15% of the capacity. The design ran at 37 MHz (implemented by a programmable phase-locked loop).

Handel-C code for a switch element with two input channels, `a` and `b`, and two output channels, `y0` and `y1` is shown below. The variable `switched` represents the switch element configuration. As the code represents a macro of type procedure, the 'parameters' are untyped. The figure '1' before the buffer declarations indicates the data width of 1 bit. `show` instructs the simulator not to include this object in its output (to avoid display clutter). It is left to the reader to assess whether (say) the presence of `par` statements and the `?, !` symbols are a significant impediment to the C programmer:

```
macro proc sw_element(a, b, y0, y1, switched)
    {
    unsigned 1 buf_a, buf_b with {show = 0};
    par{
        while(TRUE){
            par{a ? buf_a; b ? buf_b; }
        }
        while(TRUE){
            if(switched == 0){
                par{ y0 ! buf_a; y1 ! buf_b;}
            }
            else{
                par{ y1 ! buf_a; y0 ! buf_b;}
            }
        }
    }
}
```

While in VHDL a single cycle transfer might reasonably be expected to take place, in Handel-C transfer is implemented and governed by the semantics of channels as illustrated in Fig. 6. Before transfer takes place across the datapath, both processes must be ready, which is signaled through handshaking in the previous cycle, resulting in a register enable. `start` and `finish` signals are required to ensure the silicon program's correct start and termination. If sequentiality had been introduced into the switch element design, as in the following code, then though the latency (7 cycles) would remain the same, throughput would be halved (for equivalent clock speed) as there is a two-cycle delay at the onset while channels are made ready.

```
void p5_se00() {
     while(true){
         par{
             ca0 ? a0; cb0 ? b0;
             if(swc.1 == 0){
                 par{ ca1 ! a0 ; cb1 ! b0;}
             }
             else{
                    par{ cb1 ! a0 ; ca1 ! b0;}
                }
            }
        }
    }
```

By way of comparison with the VHDL design, the Handel-C switch was built up on a stage-by-stage basis, and it is easy to see how this is convenient given the VHDL `generate` statement. However, it is also possible to introduce a different modularity in Handel-C, Fig. 7, which might be significant when building up component switch libraries. Making connections directly in the source code for a repetitive design is error prone, so that even for this small switch it was worthwhile to write a script in Visual Basic to automate generation of the source code, when creating the stage-by-stage version.

Code for an ATM stage simply consists of a number of switching elements within a `par` statement. Code for a switch consists of parallel 'calls' (in fact code substitution through macros) to a set of suitably parameterized stages. Finally, a 'testbed' can be coded, the equivalent of the VHDL shift-register but with no reference to digital techniques (with all switches set to straight through for verification purposes in this simplified version):

```
macro proc tb_out(traf_out) {
    while(1){
        par{traf_out ! 0x01; swc = 0x0000;}
        par{traf_out ! 0x02; swc = 0x0000;}
        par{traf_out ! 0x04; swc = 0x0000;}
        par{traf_out ! 0x08; swc = 0x0000;}
        par{traf_out ! 0x10; swc = 0x0000;}
        par{traf_out ! 0x20; swc = 0x0000;}
        par{traf_out ! 0x40; swc = 0x0000;}
        par{traf_out ! 0x80; swc = 0x0000;}
    }
}
```

5.3 Comparison of implementations

The VHDL implementation was on a member of Actel's recent SX FPGA family [55], while the Handel implementation was on a member of the established Xilinx 3000 series FPGA. In Table 4, the results are extrapolated to a more recent Xilinx FPGA, and a larger Actel device, both lower-power devices. Extrapolation is by reference to detailed manufacturer's data sheets. As there are many other points of comparison than contained in Table 4 reference should always be made to device data sheets. Actel's FPGAs are programme-once, anti-fuse devices, which cannot be reverse-engineered. Actel's logic blocks are smaller than Xilinx CLBs which may increase network latency on the Actel devices [56]. What is clear from Table 4 is that internal clock speeds on Actel devices are faster, whereas the number of logic gates on Xilinx devices is larger. The number of paired I/O lines indicates the size of switch that can be accommodated, about 256×256 on either of the two larger members of the two FPGA types.

An interesting feature of the comparison is that Handel appears to have used fewer gates than the raw VHDL design. As signaling functionality along with routing table size also needs to be gauged in any design, the number of gates available after the basic switch is implemented needs to be considered. Xilinx have designated, though not at the time of writing made available, two of the Virtex-E (1.8 V) lower-power FPGA family for network switching based on extended RAM provision. Notice that when running the existing Virtex XCV1000 (2.5V, 0.25 micron feature distance), when fully-loaded at about 100 MHz, independent tests record 25 W of power consumption; not low-power and unsuitable for mobile devices.[14]

Clock speed obviously affects an ATM cell's (53 octets) switch traversal latency. Therefore, it would appear, from Table 4, that it is the clock effect on throughput that is more important.

Table 4 represents the fully-parallel version of the code and it might be thought that this would be optimal. However, in experiments with a reduced-size switch, it was found that a design with the two-step switching elements (Section 5.2) allowed a faster clock speed (86 MHz) than a fully-parallel design (46 MHz). The gate count was also reduced from 435 gates (parallel) to 360 gates ('sequential'), which is not surprising as a `par` barrier results in extra logic. This suggests that a technology mapping can result in longer routes, and, therefore, longer settling down time. Unfortunately, this information is not available from the present Handel-C tools as there is no simulation level below the cycle level, which raises a question mark over usage

[14]A Pentium II has 7.5 M transistors, while the Virtex ranges from 25 M to 50 M transistors, all adding to the power consumption.

on programme-once devices. As it happened for this design, the reduction in clock-speed is almost exactly balanced by the decrease in throughput caused by the two-step design, though there is still a disparity in gate usage. In general, this type of tradeoff causes problems when decoupling behavioural analysis through software from technology mapping.

An unstated point of comparison is the time taken to implement a design as this depends on the skill-level and experience of those performing the design. Once the Handel-C development process was understood, the raw ATM switch design could be coded and tested in less than a day. The MP3 port to Handel-C (Section 4) is reported to have taken four man-months to produce working silicon, but presumably by skilled personnel. It is not difficult to convert from existing C code, whereas converting to VHDL is time-consuming without translation software. As we move to larger designs, a saving in coding time is anticipated as VHDL is well-known to be a verbose language, stemming from its origins as a documentation language. For example in [57], the high-level design of a cable modem chip took 5,348 lines of code (loc) while 21,798 loc were used in a C++ version. It could be argued from the two code examples, that *neither* VHDL nor Handel-C are terse enough if prototyping is the goal.

The complexity of the design could significantly impact Handel-C compile time on the older C-version of the compiler. A six-stage 16-input design with connections straight through took 19 minutes to compile on a 233 MHz Pentium II, whereas a full seven-stage equivalent design took 30 minutes. When the connections were transferred to make the Benes switch pattern, the compile time increased to 3 hours 14 minutes. However, the newer version, 2.1, of the compiler, which we only used for compilation and simulation on a trial basis, has reduced these compile times to 2 s. Synthesis time reduced from 41 minutes to 31 minutes. Improvements to the Handel compiler, therefore, appear to have addressed compile time concerns which we originally experienced, but the time required to place-and-route a complex design would still restrict the interactivity of the high-level design process.

5.4 Discussion

Handel-C is best-suited to rapid prototyping and proof of concept engineering rather than high-performance, optimal solutions. However the definition of what an optimal solution is may shift just as it has done for general-purpose programming where sub-optimal hardware usage is accepted to achieve efficiency in software development. Handel-C allows hardware to be implemented without a digital background, although awareness of the underlying synchronous finite machine model and digital design trade-offs is beneficial. The `par` statement provides a flexible and code-efficient mechanism for specifying parallelism of arbitrary granularity. In fact, a Handel-C program might equally translate to a software-only design [58], whereas that is not so likely for VHDL. By being able to target code to software and hardware implementations, a true codesign is possible.

Specification at behavioural-level minimizes the amount of code compared to HDLs such as VHDL and insulates the software engineer from low-level hardware detail. The trade-off is that the Handel-C compiler controls implementation detail so there is no opportunity to perform the gate-level optimization that is available to VHDL-trained engineers. Consequently, gate counts may be higher than a VHDL solution (though not necessarily as the example demonstrates), and the inability to dictate logic to the FPGA technology mapping can result in lower clock speeds.

Handel-C is a product which has only just made the transition from a research phase to commercial exploitation. There has probably simply not been time to develop extensive support for modularity and debugging facilities, as is present in HDL toolkits. As larger-scale projects are tackled these two issues will become significant. Basic function and procedure mechanisms are adequate when FPGA capacity is limited to less than 10 kgates but will become unacceptable as designs with 1 Mgates occur. Equally, long place-and-route times will make debugging aids imperative as programs become larger. In short, though hardware compilation has come a long way, scalability has now become an issue which was not present when the journey was commenced.

6 Conclusion

A convergence of opinion now favours high-level languages as a means of creating high-level, behavioural models for hardware. System-on-a-chip designers prefer C/C++ because partitioning between hardware and software can remain uncommitted for a longer time. Software acceleration of general-purpose programs by compiler is now feasible as RISC processors share a chip with reconfigurable hardware. The vertical integrity of a design is preserved by relating the initial design outline to the code outline and hence to circuit synthesis. This paper has examined the current feasibility of achieving a transparent design process all the way from high-level language software specification to silicon implementation using Handel-C, and compared it with an alternative hardware-oriented approach based upon VHDL. Language modifications, as exemplified in

Handel-C, enable guidance to digital design novitiates to be built in or in the long run to be abstracted away. Therefore, it may become feasible to solve the skills gap by means of software-oriented hardware programming. Two weaknesses of Handel-C as it stands are its current limitation to a single clock, and its weak support for modularity. Abstracting from its origin as a language targeted only at FPGAs, Handel-C would now appear to be too rigid in its treatment of clocks whereas VHDL may be considered to be too general.

While much work has been done on the FPGA development processes little attention appears to have been given to the question as to how related hardware and software implementations might be best integrated within existing software component models. Increasingly, software development uses objects as the basis for integrating diverse components within applications. Not only does this increase the opportunity for reuse, but it also insulates programmers from implementation detail. The need to abstract away the detail of how families of processing algorithms are implemented would seem to be beneficial, since application code would be independent of a hardware/software partitioning or specific algorithmic implementations. This would facilitate the integration of hardware/software libraries (based on different application performance profiles) into application code. VHDL is already object-oriented, with readily reusable entities and architectures. However, VHDL lacks inheritance, therefore, one direction to go would be to strengthen VHDL's object-oriented offering with inheritance, perhaps also incorporating dynamic component structure onto the passive structure already available in the object model. Handel-C will also need to be reinforced to provide higher-levels of encapsulation.

Acknowledgement

This work is being carried out with assistance from an EPSRC/DERA CASE studentship no. 9930329X. We thank R. W. M. Smith for guidance on aspects of VHDL and for a helpful review of this paper prior to submission.

References

[1] J. F. Wakerly. *Digital Design: Principles and Practices.* Prentice Hall, Upper Saddle River, NJ, $3^{\underline{rd}}$ edition, 2000.

[2] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentielli. Architecture of Field-Programmable Gate Arrays. *Proceedings of IEEE*, pages 1013–1028, 1993.

[3] Xilinx Inc., 2100 Logic Drive, San Jose, CA. *Virtex 2.5V Field Programmable Gate Arrays Datasheet*, 2000. Available as `http://www.xilinx.com/partinfo/d2003.pdf`.

[4] B. M. Pangrle and D. D. Gajski. Design tools for intelligent silicon compilation. *IEEE Transactions on Computer-Aided Design*, 6(6):1098–1112, 1987.

[5] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12:87–107, 1996.

[6] C. A. R. Hoare, J. He, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–793, 1993.

[7] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vransic. *Field-Programmable Gate Arrays.* Kluwer, Boston, 1992.

[8] M. Spivey, I. Page, and W. Luk. *How to program in Handel.* Oxford University Hardware Compilation Unit, 1995.

[9] M. Bowen. *Handel-C Language Reference Manual, 2.1.* Embedded Solutions Ltd, Abingdon, UK, 1998.

[10] D. Bursky. Scalable, reconfigurable processor adjusts logic for top performance. *Electronic Design*, 48(10):66–70, 2000.

[11] M. B. Gokhale and J. M. Stone. NAPA C: Compiling for a hybrid RISC/FPGA architecture. In *IEEE Symposium on FPGAs for Custom Computing*, pages 578–587, 1998.

[12] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2.* Adam Hilger, Bristol, UK, 1988.

[13] A. DeHon. The density advantage of configurable computing. *IEEE Computer*, 33(4):41–49, 2000.

[14] P. W. Foulk. User configurable logic. *IEE Computing and Control Engineering*, 3(5):205–213, 1992.

[15] S. Trimberger. A reprogrammable gate array and applications. *Proceedings of the IEEE*, 81(7):1030–1041, 1993.

[16] J. E. Istiyanto. *The Application of Architectural Synthesis to the Reconfiguration of FPGA-based Special-Purpose Hardware.* PhD thesis, University of Essex, UK, 1995.

[17] N. Park and A. C. Parker. Sehwa: A software package for synthesis of pipelines from behavioral descriptions. *IEEE Transactions on Computer-Aided Design*, 3(7):356–370, 1988.

[18] A. C. Parker. Maha: A program for data path synthesis. In *ACM/IEEE Design Automation Conference, DAC'86*, pages 461–466, 1986.

[19] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, 8(6):661–679, 1989.

[20] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Transactions on Computer-Aided Design*, 6(3):259–269, 1987.

[21] E. Girzyc, R. J. A. Buhr, and J. P. Knight. Applicability of a subset of Ada as an algorithmic hardware description language for graph-based hardware compilation. *IEEE Transactions on Computer-Aided Design*, 4(2):134–142, 1985.

[22] G. de Micheli, D. Ku, F. Mailhot, and T. Truong. The Olympus synthesis system. *IEEE Design & Test of Computers*, pages 37–53, October 1990.

[23] M. Leeser, R. Chapman, M. Aagaard, M. Linderman, and S. Meier. High level synthesis and generating FPGAs with the Bedroc system. *Journal of VLSI Signal Processing*, 6(2):191–214, 1993.

[24] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Academic, Norwell, MA, 1990.

[25] R. Camposano and R. Rosentiel. Synthesizing circuits from behavioral descriptions. *IEEE Transactions on Computer-Aided Design*, 8(2):171–180, 1989.

[26] D. Thomas and P. Moorby. *The Verilog Hardware Description Language.* Kluwer, Boston, MA, 1991.

[27] D. C. Blight and R. D. McLeod. VHDL for FPGA design. In W. R. Moore and W. Luk, editors, *FPGAs*, pages 246–254. EE & CS, Abingdon, UK, 1991.

[28] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmidt. Ptolemy: A framework for simulation and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 1994.

[29] A. Burns and A. Wellings. *Real-time Systems and their Programming Languages.* Addison-Wesley, Wokingham, UK, 1989.

[30] C. G. Bell and A. Newell. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1971.

[31] P. J. Ashenden and M. Radetzki. Comparison of SUAVE and Objective VHDL language features. In $2^{\underline{nd}}$ *Forum on Design Languages, Lyon, France*, pages 292–297, 1999.

[32] J. McCormack, R. McNamara, C. Gianos, N. P. Jouppi, T. Dutton, J. Zurawski, L. Seiler, and K. Correll. Implementing the Neon: A 256-bit graphics accelerator. *IEEE Micro*, 19(2):58–69, 1999.

[33] V. N. Muchnick and A. V. Shafarenko. *Data-Parallel Computing: the Language Dimension.* Thomson, London, 1996.

[34] M. Weinhardt and W. Luk. Memory access optimization and RAM inference for pipeline vectorization. In P. Lysaght, J. Irvine, and R. Hartenstein, editors, *Field-Programmable Logic and Applications. Proceedings of the* $9^{\underline{th}}$ *International Workshop, FPL '99*, volume 1673 of *Lecture Notes in Computer Science*, pages 61–70. Springer, Berlin, 1999.

[35] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In K. L. Pocek and J. M. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, 1997.

[36] I. Page and W. Luk. Compiling occam into FPGAs. In W. R. Moore and W. Luk, editors, *FPGAs*, pages 271–283. EE & CS, Abingdon, UK, 1991.

[37] *occam 2 Reference Model.* Prentice Hall, New York, 1988.

[38] L. Paulson. *ML for the Working Programmer.* Cambridge University Press, UK, 1991.

[39] P. Hilfinger. A high-level language and silicon compiler for digital signal processing. In *IEEE Custom Integrated Circuit Conference*, pages 213–216, 1985.

[40] E. W. Dijkstra. Guarded commands, nondeterminancy, and formal derivation of programs. *Communications of ACM*, 18(8):453–457, 1975.

[41] D. A. P. Mitchell, J. A. Tompson, G. A. Manson, and G. R. Brookes. *Inside the Transputer.* Blackwell Scientific, Oxford, 1990.

[42] Converting mp3 software to hardware, 2000. Application note available at `http://www.embeddedSol.com/`.

[43] Xilinx Inc., 2100 Logic Drive, San Jose, CA. *Virtex-E 1.8V Field Programmable Gate Arrays Datasheet*, 2000. Advanced specification available as `http://www.xilinx.com/partinfo/d2022.pdf`.

[44] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[45] C. R. Jesshope, I. M. Nedelchev, and C. G. Huang. Compilation of process algebra expressions into delay-insensitive circuits. *IEE Proceedings Part E (Computers and Digital Techniques)*, 140(5):261–268, 1993.

[46] G. M. Reed and A. W. Roscoe. *A Timed Model for Communicating Sequential Processes.* Springer, Berlin, 1986. Lecture Notes in Computer Science # 226.

[47] B. Mahony and J. S. Dong. Timed communicating object Z. *IEEE Transactions on Software Engineering*, 26(2):150–117, 2000.

[48] O. Diesel and G. Milne. Compiling process algebraic descriptions into reconfigurable logic. In *IPDPS 2000 Workshops*, pages 916–923, 2000. Lecture Notes in Computer Science No. 1800.

[49] N. Wirth. Hardware compilation: Translating programs into circuits. *IEEE Computer*, 31(4):25–30, 1998.

[50] A. Pattavina. *Switching Theory.* Wiley, Chichester, UK, 1998.

[51] D. Stiliadis and A. Varma. A reconfigurable hardware approach to network simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(1):131–156, 1997.

[52] G. Kornaros, D. Pnevmatikatos, P. Vatsolaki, G. Kalokerinos, C. Xanthaki, D. Mavroidis, and M. Katevenis. ATLAS 1: Implementing a single-chip ATM switch with backpressure. *IEEE Micro*, 19(1):30–41, 1999.

[53] C. Roberts. High-speed ATM switch fabrics. *Electronic Engineering*, 71(870):9–10, 1999.

[54] M. T-C. Lee, Y-C. Hsu, Vhen B., and M. Fujita. Domain-specific high-level modeling and synthesis for ATM switch design using VHDL. In $33^{\underline{rd}}$ *Design Automation Conference*, 1996.

[55] Actel's SX family of FPGAs: A new architecture for high-performance designs, 1998. White paper available as `http://www.actel.com/products/devices/SX/SXbkgmdr.pdf`.

[56] V. Betz and J. Rose. How much logic should go in an FPGA block? *IEEE Design and Test*, pages 10–15, January-March 1998.

[57] S. Vernalde, P. Schaumont, and I. Bolsens. An object oriented programming approach for hardware design. In *IEEE Conference on VLSI*, pages 68–73, 1999.

[58] B. Lin. Efficient compilation of process-based concurrent programs without run-time scheduling. In *Design Automation and Test, DATE'98*, 1998.

List of Tables

List of Figures

Table 1: Comparison of occam and Handel-C

Language Feature	occam 2	Handel-C v. 2.1
format	indented	open
variables	various	arbitrary-width integers
constants	supported	supported
type checking	rigid	strict
scope	nested	within basic blocks
channels	`rendez-vous` sync.	as in occam
channel protocols	simple, variant, 'anarchic'	data widths
sequential section	explicit sequences	currently implicit
barrier parallelism	explicit	explicit
alternation	non-deterministic choice	prioritized choice
do nothing	SKIP process	delay statement
halt	STOP process	dropped from Handel
subroutines	procedures & functions	macros
conditional choice	guarded, halts if no guard is ready	'C'-like
replication	supported for all process types	not supported
iteration,selection	'C'-like	'C'-like
abbreviation	supported	'C' pre-processor
timers	supported	lock-step sync.
operators	includes range of modulo operators	includes bit selection & concatenation

Table 2: Differences between Handel-C and VHDL

Language Feature	Handel-C v. 2.1	VHDL
Inter-process communication	Processes linked by channels. Operators '!' and '?' performs input/output.	Entities linked via signals (wires) over entity ports. Continuous assignment operator '<=' performs i/o.
Data types	Integer/char only.	Integer,float,bit,bit-vector, boolean,string,time, user-defined types & operator overloading.
Modularity	Single function source code and parameterized macros.	Libraries, packages, entities/architectures procedures, & functions.
Interfaces	Not supported	Entities/architectures.
Replication	`generate` construct	Not currently supported
Timing model	Assignment & ready-to-run channel comms. take 1 cycle. All other constructs take zero time. `delay` statement introduces 1 clock cycle delay.	All constructs execute in zero time, but `after` keyword allows delays & propagation to be modeled.
Parallelism	Default: statements execute sequentially Hardware parallelism by `par` statement blocks.	`process` block and continuous assignment.
Alternation	Priority choice	not present

Table 3: Reserved Word Comparison Between 'C' and Handel-C

Types, Type Operators, Objects		
In Both	In 'C' Only	In Handel-C
int	double	chan
unsigned	float	ram
char	enum	rom
long	register	chanin
short	static	chanout
	extern	undefined
	struct	interface
	volatile	
	void	
	const	
	union	
Statements		
In Both	In 'C' Only	In Handel-C
{;}	continue	par
switch	return	delay
do ... while	goto	?
while	typedef	!
if ... else		prialt
for(;;)		
break		

Table 4: Implementation Comparison

Device:	Actel A54SX08	Actel A54SX72	Xilinx XC3195A	Xilinx XCV3200E
Logic Gates (k)	2.6	2.6	1	1
Usage (%)	33	3	15	1
Clock (MHz)	250	250	37	130
Max. Internal Clock	320	320	85	130
Thruput (Gbps)	4	4	1.2	2.1
Per-cell Latency (μs)	12	12	80	23
I/O Pairs	125	340	176	344

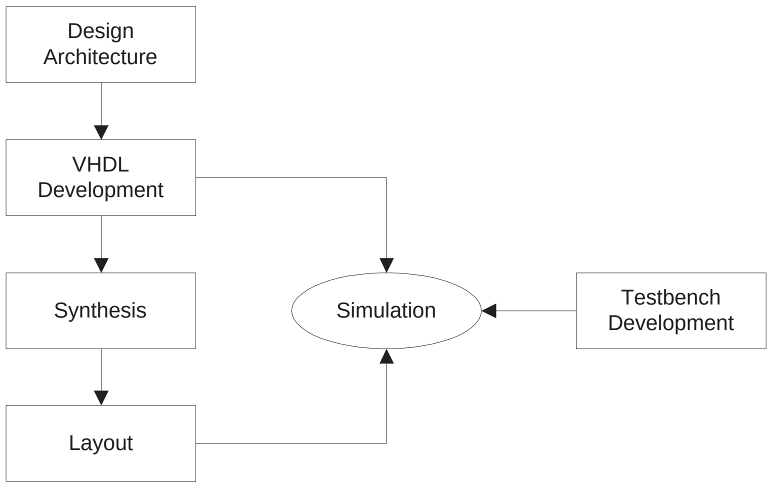

Figure 1: VHDL development process.

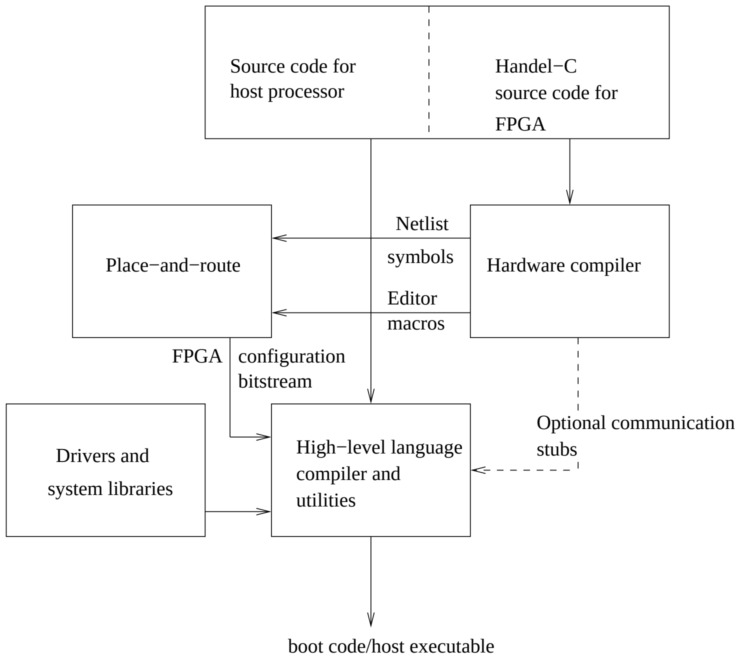
Source code for host processor
Handel−C source code for FPGA
Place−and−route
Hardware compiler
Netlist
symbols
Editor
macros
FPGA
configuration bitstream
Drivers and system libraries
High−level language compiler and utilities
Optional communication stubs
boot code/host executable

Figure 2: Handel-C development process.

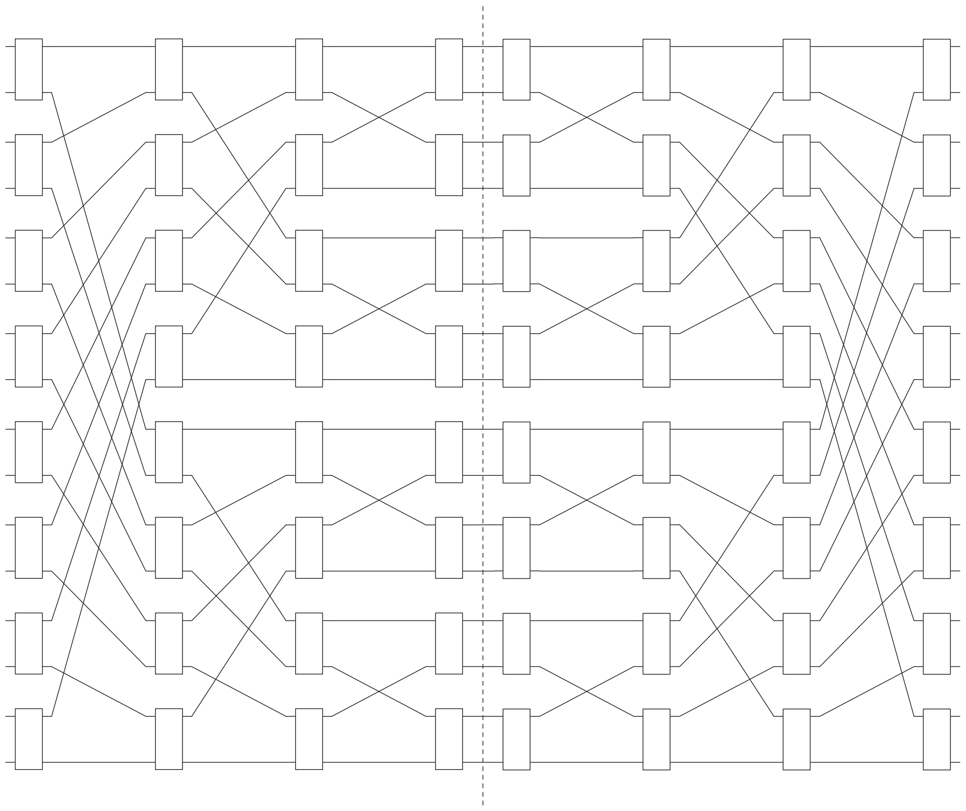

Figure 3: Horizontally-extended Baseline switch fabric topology.

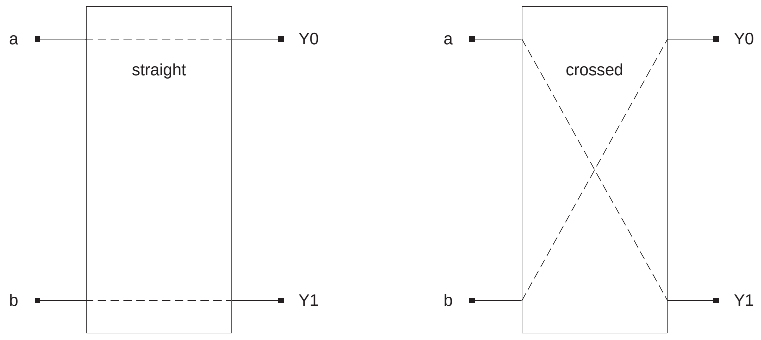

Figure 4: Switch element.

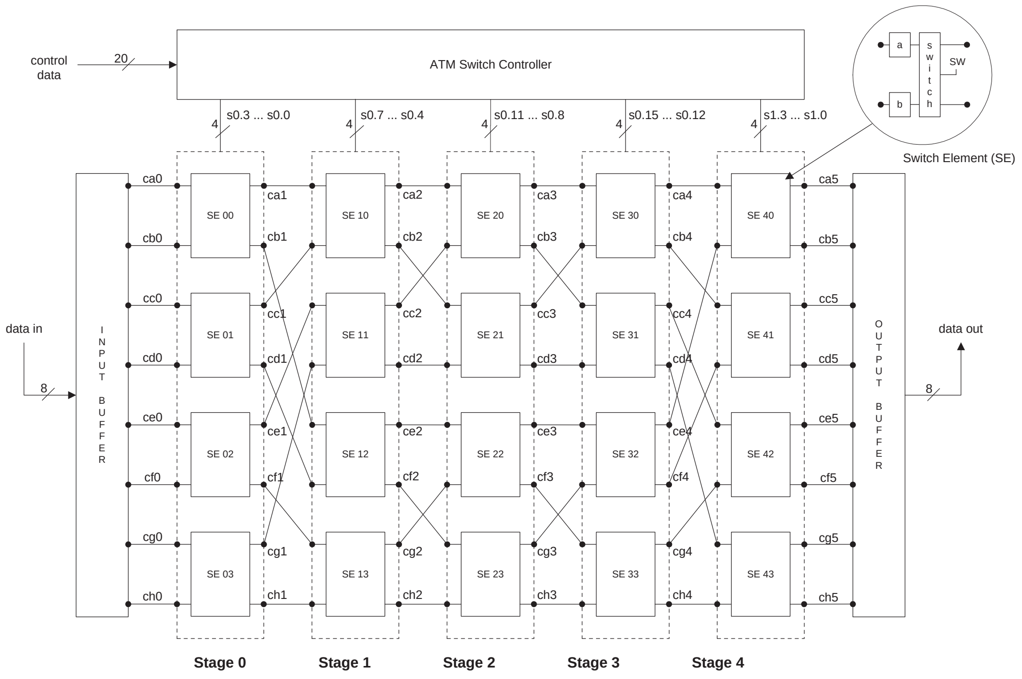

Figure 5: Handel-C version (simplified) of the ATM switch design.

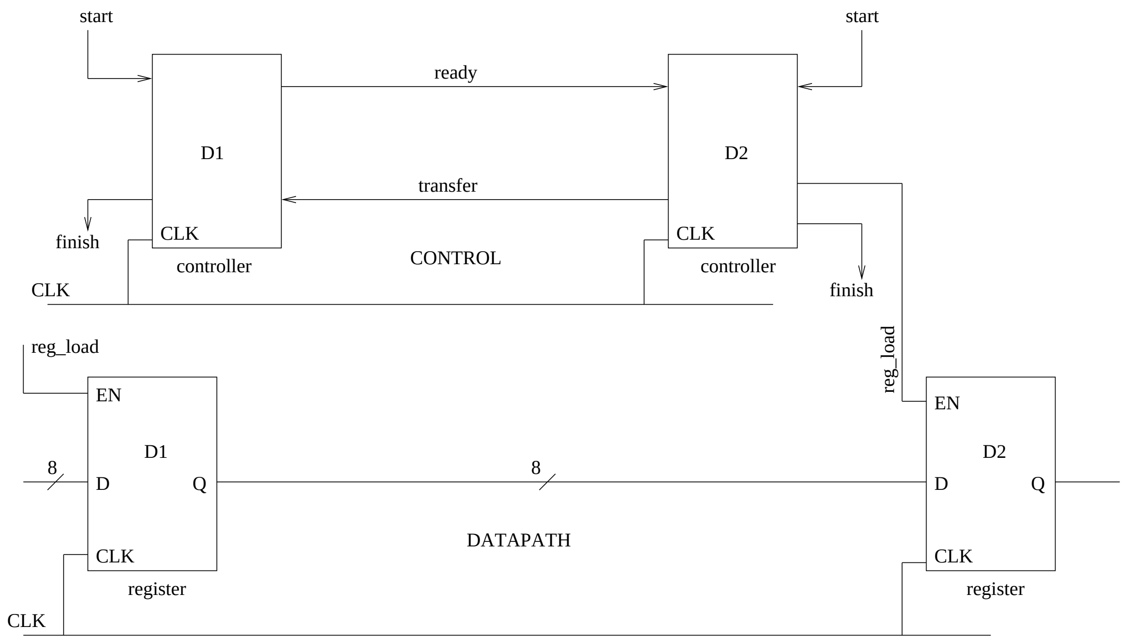

Figure 6: Channel logic.

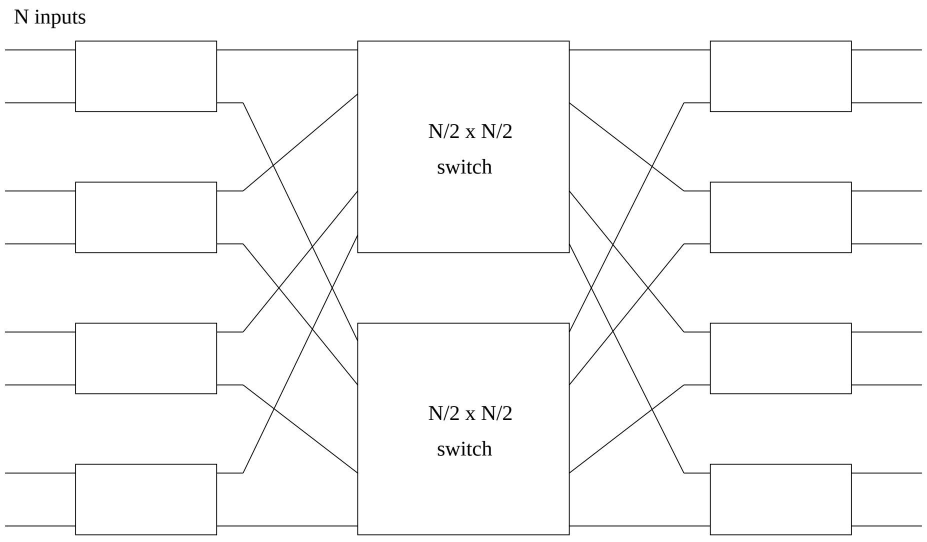

Figure 7: Recursive construction of a Benes switch.