The Technology Behind DK1

Date>: August 2002 Document Type >: Application Note AN 18

1. Introduction

This note describes *some* of the hardware constructs used in the transformation of Handel-C [1] programs to a netlist graph. A full description of the intricacies of the constructs produced by the compiler is out of the scope of this short note.

Handel-C is a synchronous programming language where the notion of time is fundamental. It builds synchronous circuits using global clocks that tick continuously (note that more than one global clock can be defined). The language is similar in abstraction to a *Register Transfer Language* and leans towards a behavioural style of programming rather than a structural *Hardware Description Language*, such as VHDL.

Handel-C captures programs at the level of simple transformations on data and movement of data between variables (registers). Parallelism can be explicitly defined. The semantics of the language are based on Occam/CSP [2,3] and the syntax is based on the C language [4].

2. The Data Path

All variables in the user program are mapped to hardware registers. These are constructed from sets of flip-flops. The exact flip-flop architecture is specific to the target device. A basic D-type flip flop is shown in **figure 1**. One flip-flop is built for each bit of a variable in the program. In the basic element the rising edge of the clock copies the value on the D-input into the flip-flop. The current state of the flip-flop is output continuously at Q. The global clocks are used to activate every flip-flop connected to that clock simultaneously as an *atomic* action.

The registers have input multiplexors if they have multiple sources, for example if they are the target of more than one assignment or communication in the Handel-C program. The control circuits for the statements in the program generate the multiplexor control signals and the clock enable signals for the destination registers. All expressions are implemented as combinational logic. The datapath generated by this strategy exactly matches the dataflow graph of the original user program.

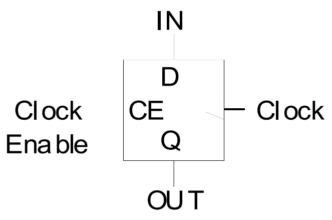

Figure 1.
Flip-Flop element with clock enable.

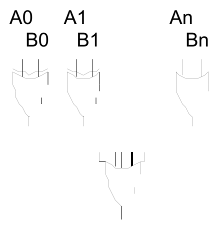

Figure 2.
Inequality construct.

3. Operators

3.1 Logical Operators

The Handel-C bitwise operators can be interpreted simply as combinational hardware circuits. The & operator is simply *n* AND gates, where *n* is the operand bit width. The | and ^ operators are similar. The ~ operator is simply *n* inverters.

3.2 Arithmetic Operators

Addition and subtraction map onto combinational logic. This may be a standard ripple carry adder but may also exploit other features of the target device. By default multiplication maps onto a combinational multiplier (a set of linked adders). A combinatorial multiplier has size of order n^2 but has to be used to ensure that the expression obeys the single cycle constraint. Other forms of operators (such as a pipeline multiplier) may be utilized to work over many clock cycles (with correspondingly reduced combinational delay on each clock cycle). The compiler also supports divide mod and shift operators.

3.2 Arithmetic Operators

Implementation of the inequality **A != B** is illustrated in **figure 2**. Each bit (0 to n) of registers **A** and **B** is compared. The relational test **A == B** is simply inequality negated. The other relational tests are implemented by subtraction and then testing the sign/carry bits with the inequality construct.

4. The Control Path

Each control construct in the program maps onto a control circuit in the hardware. A pair of control signals *(start and finish)* is used for each control circuit. An active control signal is high at the rising edge of the corresponding global clock.

In the control circuit diagrams that follow, a box with a triangle in the upper-left corner is used to denote a single instance of a control circuit. The control handshake is implemented by a single input and a single output to each control circuit.

4.1 The Delay Construct

The control circuitry for the delay construct is shown in **figure 3**. This construct has no effect on the state of the computation. It takes exactly 1 clock cycle to complete.

4.2 Assignment

The control circuit for the assignment statement

R = Exp;

is shown in **figure 4**. The start signal forms the clock enable signal for the destination register of the assignment. At the end of the cycle in which the assignment is scheduled the expression hardware has calculated the new value, which is thus loaded into the destination register. The start signal is delayed by a single clock cycle to provide the finish signal for this control procedure. This follows from the fact that assignment is always scheduled immediately and it always completes in exactly one clock cycle. Generally Handel-C assignments are loop free by design[1]. Sufficient combinational logic is built to perform the assignment in one clock cycle even if the expression is complex. The clock period is chosen to allow for the worst-case combinational delays in any complex logic such that the value of the expression hardware is guaranteed to be stable at the end of the clock period. One method for refining Handel-C programs is to split complex expression assignments to evaluate over multiple clock cycles by registering interim results.

[1]An exception to this is in the use of signals.

4.3 Assignment Multiplexing

The circuit in **figure 5** shows the portion of the datapath generated by two assignment statements

R = Exp1; R = Exp2;

which target the same register. An active start signal steers the appropriate expression value through the multiplexor to the input of the destination register. The multiplexor here is one-hot with one enable for each data line. Whichever start signal is active also enables the destination register via the OR gate.

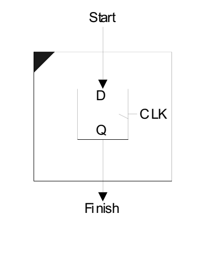

Figure 3.
The Delay construct.

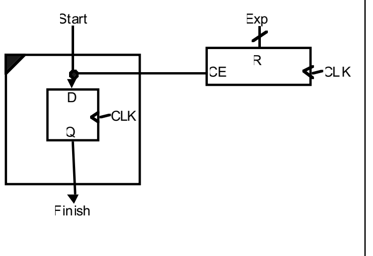

Figure 4.
Implementation of Assignment

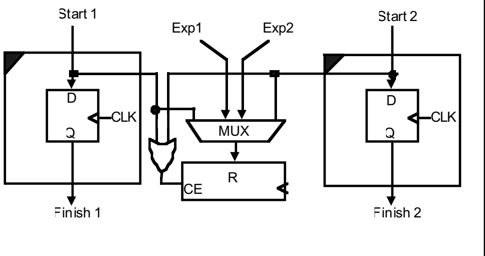

Figure 5.
Assignment Multiplexing.

4.4 Sequential Composition

The control circuit for the sequential execution of statements is trivially simple. The start and finish signals of the component processes are connected together in a "daisy-chain" as shown in **figure 6**.

The control strategy is basically that of a modified "one-hot" control state encoding. This particular scheme is well suited to FPGA implementations since it requires little in the way of wiring and logic resources when compared to encoded representations of control state. Indeed, the compiler can take advantage of architecture specifics such as shift registers to further optimize implementation. Any program control circuits that obey the handshake assumptions can be connected in this way such that the entire construct also obeys these assumptions.

4.5 Parallel Composition

Figure 7 shows the control construct used for parallel execution of statements using

par{S_1;S_2;... S_n };

This circuit passes control simultaneously to all of the parallel statements **S_1,S_2,... S_n** to initiate parallel execution. The **par{}**construct terminates only when all of the constituent components have terminated. Thus the parallel control circuit collects together the separate finish signals in a set of flip-flops and produces its own finish signal as soon as the last finish signal is generated. In addition, this signal resets the synchronizer flip-flops ready for the next time the circuit is used. Note that wherever a textual analysis of the program can demonstrate a partial ordering on the termination times of the individual processes the compiler will optimize the circuit to remove such synchronizers.

Parallel and sequential compositions may be arbitrarily nested.

5. Flow Control

The control circuitry for the binary choice

if{BE}S_1; else S_2;

is given in **figure 8**. The start signal is steered to trigger just one of the control processes S_1 or S_2 depending on the value of the Boolean guard expression (BE). Since only one command can be active, the finish signals from S_1 and S_2 are simply ORed together to derive the completion handshake for the entire construct.
The switch construct builds an equality test **(figure 2)** for each of the test cases. Since the tests are restricted to compile time constants the optimization process significantly reduces the amount of hardware required f
or these comparisons.

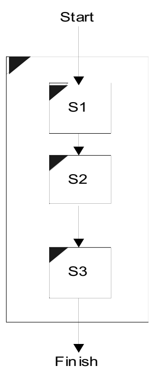

Figure 6.
Sequential composition.

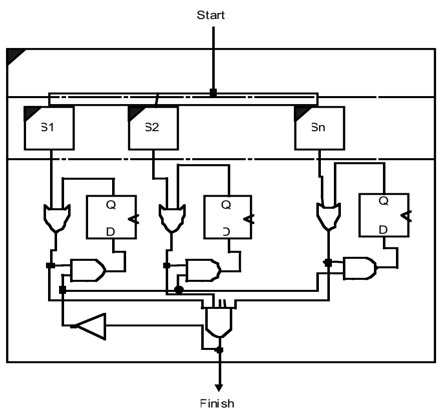

Figure 7.
Parallel composition.

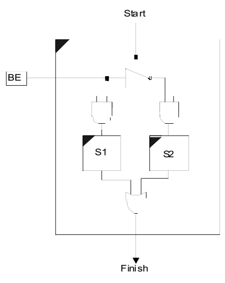

Figure 8.
The if construct.

5.1 Guarded Iteration

The while loop

while (BE) S*;

is similar to the if circuit except that here **(figure 9a)**, under the control of the Boolean guard expression **(BE)**, the start signal is steered either to the feedback loop or to form the finish signal. The locus of control is trapped in the feedback loop until the Boolean expression evaluates to false. Here we must be careful that S* does not have any combinational path through it from start to finish that executes in "zero time". If it does then this circuit will violate the control assumptions and the circuit will fail. In fact, the compiler tests for such conditions and "fixes" the circuit by inserting a delay.

A very similar circuit exists for the

do S* while (BE);

loop **(figure 9b)**. Again S* is repeatedly activated until BE is false, except that the start signal is guaranteed to execute S* at least once.

A for loop will be compiled to the equivalent while construct unless there is a continue statement present, in which case a modified implementation is generated to produce the correct behaviour.

5.2 Inter-process Communication

In Handel-C a channel (chan) is an abstract mechanism for communication, which can be used by only one pair of processes at any one time. It is used to pass a single-word message, which is the width of the particular channel. By default the communication is directed, point-to-point, unbuffered and synchronized. The compiler also supports asynchronous channels. The simplest form of channel communication is discussed here.

An input (or output) process naming a particular channel waits for a corresponding output (or input) process to be ready to communicate on that channel. When both processes are ready, the (single-word) message is passed across and both processes are then allowed to continue execution. **Figure 10** shows the control (synchronization) circuitry for channel input and output. The circuit is the same for the channel input

c ? var ;

and output

c ! Expr

commands.

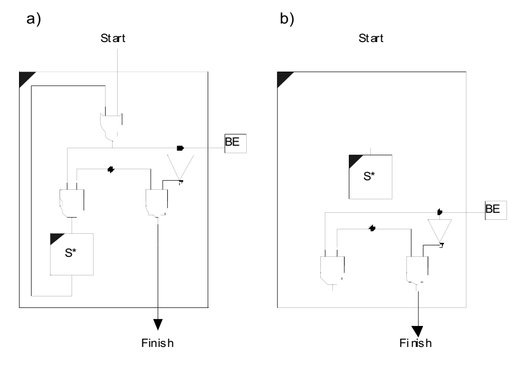

Figure 9.
The (a) while and (b) do while.

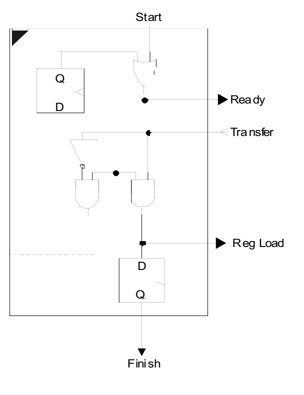

Figure 10.
Channel communication synchronization construct.

Synchronization is achieved by looping back the start signal through a flip-flop and a multiplexor controlled by the transfer signal. The control signal is trapped in this feedback loop until activation of the transfer signal.

The ready signal goes to the arbitration circuit **(figure 11)** and returns as the transfer signal to indicate when the partner to this communication is also ready to run. When both circuits are ready to communicate, the circuit below the dashed line in **figure 10** is activated. This is simply the assignment circuit seen earlier. These diagrams illustrate that here channel communication is just distributed, synchronized assignment.

As with assignment certain scope and usage rules guarantee that there can be no more than one input and one output command scheduled for the same channel at the same time, hence the very simple arbitration circuit in **figure 11.**

6. The Compilation Process

The compiler parses the user's program by recursive descent of the abstract syntax tree representation. At this time it performs width inference on variables and operators, and various checks to ensure that design violations are not present in the circuit.

There are back-end netlist transformation routines that massage the netlist so that it is optimized and suitable for different FPGAs. These transformations might impose a ceiling on the number of inputs to any gate, for example, or might implement one of the compilers abstract flip-flops with one of a small number of physical circuits depending on the target technology.

The resulting netlist could in principle be implemented by any available means, ASICs or FPGAs. However because of our interest in fast system development and dynamic systems we tend to use FPGA implementations. The FPGA vendor's place and route software is used to produce a configuration bitmap from our netlist for download to the target architecture at system initiation time. The output of the place and route software also gives an estimate for worst-case delay through the combinational circuitry. This is used to set the FPGA clock speed.

6.1 Optimizations

Since we are compiling from software to hardware we are able to benefit from both traditional software compiler techniques, and logic optimization techniques. In this section we describe the general framework for compilation from the abstract syntax tree to target hardware, and detail some of the optimization techniques used.

The input code is first transformed to an abstract syntax tree. During this transformation the compiler performs width inferencing, constant folding and other syntactic expansions (such as macro expressions, loop unfolding etc.). The abstract syntax tree is then compiled to a high-level netlist containing coarse blocks of functionality. Optimizations are applied to this high-level Netlist before it is expanded to a technology specific lower-level netlist. This format is able to take advantage of specific architectural features of the target device, such as fast carry chains, on-chip RAM, dedicated multiplier circuitry and so forth. Further optimization is then performed on the low-level netlist. Three of the simplest optimization techniques used are now described in more detail.

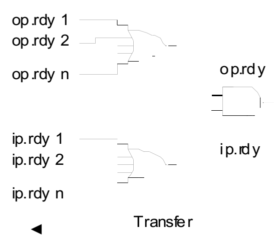

Figure 11.
Channel communication arbitration construct.

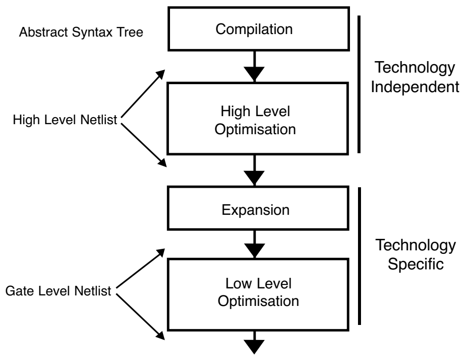

Figure 12.
Compilation flow after parsing.

6.2 Re-writing

Re-writing involves transformations to the gate-level netlist that leverage logical equivalences. For example, a constant '1' input to an AND gate is removed (figure 13(a)). This sort of situation arises when constants are used in expressions such as

x = y + 7;

Chains of inverters can be optimized to reduce gate count, and flip-flops with constant outputs can be optimized to a logical '1' or '0'.

If the output of a gate is not used, then that gate can be removed since it will have no effect on the circuit operation **(13(b))**. Additional checks are performed to remove blocks of circuitry that form a 'loop' but are not connected to an external pin **(13(c))**. This optimization is similar to dead-code elimination in a traditional compiler, except that we can remove hardware that is not connected to an output pin even if it is 'executed' at some point.

6.3 Conditional Re-writing

An extension of re-writing is to apply test patterns to gates to infer impossible conditions (such as a gate output being both '0' and '1' at the same time) – since these conditions can never exist the compiler can optimize away any circuitry that contributes to such cases – see **figure 14** for an example.

6.4 Common Sub-Expression Elimination

The classic CSE optimization from software compilation techniques is also applicable at the hardware level. The example given in figure 15 shows how two identical gates with identical inputs are optimized.

7. Conclusions

The DK1 design suite provides for the rapid synthesis, optimization and device specific targeting of C based software to hardware.

By using a simple, yet powerful, model of timing and parallelism we are able to produce robust, deterministic hardware implementations of C based algorithms. A series of optimization stages and technology specific mapping phases ensures that the output hardware is both speed and area efficient.

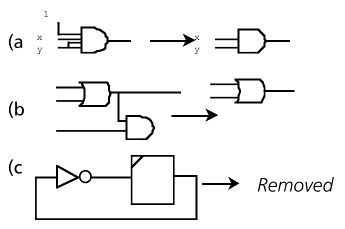

Figure 13.
Some re-writing optimizations.

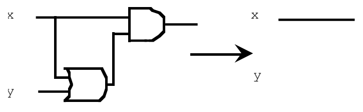

Figure 14.
Conditional re-writing.

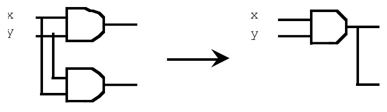

Figure 15.
Common sub-expression elimination.

8. References

[1] Handel-C Language Reference Manual.

[2] C.A.R. Hoare, Communicating Sequential Processes, International Series in Computer Science, Prentice-Hall, 1985.

[3] Inmos, The occam2 Programming Manual, Prentice-Hall,1988.

[4] B.W. Kernigham and D.M. Ritchie, The C Programming Language, Prentice-Hall,1988.

9. Further Information

For information on Celoxica products contact **Sales@celoxica.com.**
Regional office locations are given below.

Revision History

Date	Version	Revision
August 2002	1.1	New Layout And Contact Details

Celoxica in Europe
T: +44 (0) 1235 863 656
E: sales.emea@celoxica.com

Celoxica in the Americas
T: +1 800 570 7004
E: sales.america@celoxica.com

Celoxica in Japan
T: +81 (0) 45 331 0218
E: sales.japan@celoxica.com

Celoxica in Asia Pacific
T: +65 6896 4838
E: sales.apac@celoxica.com

www.celoxica.com