# CBG CTOV Compiler - Two Page Overview.

DJ Greaves*

## 1 Summary

The CBG CTOV compiler converts almost arbitrary C/C++ code to a hardware netlist. It operates by building the datapath for a VLIW-like processor and a sequencer to control the datapath that may or may not be microprogrammed. The datapath may consist of any number of RAMs and ALUs and also holding registers for data that cannot be transferred to/from RAM at the current time owing to RAM port bandwidth constraints. There are many possible datapaths that will serve to execute a given program, trading time for space. The compiler will generate a datapath using an internal heuristic that places each array present in the source code in a different RAM and converts all other state-bearing variables to flip-flops. However, the user can override the default action to control the datapath shape: he may specify the location of every variable stating whether it should be placed in a RAM or not, and if so, which RAM. The number of RAMs used and the style and number of ports on each RAM can also be fully specified by the user. This allows complete flexibility within a design space that encompasses a pair of opposite points: one point where all state is held in flip-flops and another point that places all programming-model state in a single, single-ported RAM.

The compiler partitions the behaviour of the input program into macrocycles. A macrocycle corresponds to a number of successive steps in the input program by one thread. The number of steps implemented by a macrocycle is controlled by a combination of three things: a size heuristic, decidability of name aliases and user 'barrier' statements inserted in the source code. Each macrocycle may take a number of clock cycles to execute on the hardware, depending on the availability of ALU's and RAM ports to compute, source and sink the data. Either run-time or compile-time arbitration can be used to resolve such structural hazards. Again, a default heuristic is provided: the compiler performs compile-time arbitration for competition between events caused by a common thread and run-time arbitration for inter-thread competition. Undecidability causes termination of the macrocycle generation when array subscript comparison and/or loop exit conditions cannot be calculated at compile time.

## 2 Operation

The default entry point to the user's program is the function '`main`' but various entry points can be specified on the command line for concurrent designs. The compiler operates by breaking the control flow graph of the program into loop-free sequences of instructions (known as 'twiglets') and then performing a symbolic evaluation of each twiglet to determine the state changes it would make when executed. A separate strand of processing maps each variable in the user's program to a virtual storage resource: either one or more locations in a RAM or one or more flip-flops. The union of the datapaths between these resources required by each twiglet is then formed. Each twiglet is then allocated a static schedule of operations that can executed by the datapath as a macrocycle. Many temporary variables (such as a loop counter for an unwound loop) are no longer required and so discarded, thereby giving the final concrete storage allocations for components of the programming model. Additional holding registers are needed to overcome structural hazarsm such as when both operands to an addition are sourced from the same, single-ported RAM.

The sequencer is generated from the control flow graph of the user program. Each twiglet is like a VLIW instruction. The data driven part of the control flow is implemented by generating and compiling predicate functions that tap the appropriate part of the datapath. It accepts a clock and reset as external inputs. A one-hot sequencer is generated by default, but a custom microcontroller and accompanying microcode PROM are just as easily created. The one-hot sequencer allows dynamic thread fork and join.

Generation of RTL output as a structured Verilog netlist is straightforward. Each RAM and ALU appears as an instance in the netlist, except for RAMs that have been declared 'offchip' which are instead accessed via formal connections.

Input and output to the hardware is implemented by interpreting the formal parameters given to the entry function(s). Any call by value parameter or undriven call-by-

*David.Greaves@cl.cam.ac.uk

reference parameter is compiled as an input, whereas updated call-by-reference parameters are compiled as outputs.

As is commonly required in hardware designs, registers of various widths are supported using macros in an imported C header file. These can follow the SystemC style of coding or our own. The macros are compiled to hidden C code that is interpreted by the tool, or else can be compiled to the nearest natural C variable when the source code is being used in a software environment.

Multi-threading of the input code is often useful and is supported by the tool. Each thread has its own sequencer and datapath but shares the same RAMs and programming-model flip-flops. This allows true concurrency. Runtime arbitration occurs when two sequencers require to put different addresses on the same address bus to a RAM. Mutex variables are provided to support atomic test and set operations on a variable for thread synchronisation. For all other variables, if two threads write at once, the results are random.

Initial values of variables are loaded when the hardware's reset input is asserted, except for variables placed in RAMs.

## 3 Source Code Restrictions

The source code must have a non-recursive flow graph. The other requirement is that all pointer arithmetic must be resolvable at compile time to narrow down the range of access to an individually declared array. Dynamic calls (index branches) are not implemented but can be added using the normal source-to-source preprocessing.

For debugging use, a version of printf can be added to the source code and compiled to hardware, where the '`write`' output routine is trapped to the Verilog '`$display`' PLI function.

## 4 Current Status

The tool is implemented and has been tested on some small designs. It consists of a C/C++ parser written in C and a main body written in SML. As a future work item, the parser could be replaced with llvm.