Composing Data-driven Circuits Using Handshake in the Clock-Synchronous Domain

Jaroslav Sykora
Institute of Information Theory and Automation (UTIA) of the ASCR
Pod Vodarenskou vezi 4, Prague, Czech Republic
Email: sykora@utia.cz

***Abstract*—We present a technique for modelling and synthesis of fine-grained data-driven circuits in the clock-synchronous hardware, such as the field programmable gate arrays (FPGA), called the *Flow-Transfer Level* (FTL). The distinguishing property of the FTL technique is that it *does not* rely on FIFO queues to handle flow synchronization between the components (called operators). The communication channels, called pipes, employ conceptually a two-state handshake protocol. The handshake behaviour of each operator is defined logically using dependency subgraphs that are symmetrical for producers and consumers. The original data-flow netlist of operators is transformed into a global control dependency graph. Cycles in dependency graphs are allowed as long as they do not constitute real data dependencies but only dependencies in promises of handshake completions. A method is given that recursively eliminates these cycles. We demonstrate the feasibility of the approach in a prototype compiler that transforms an FTL netlist into a synthesizable VHDL code. A comparison to a manual RTL VHDL design shows that our technique is very light-weight, yet it has a potential of increasing the design abstraction level.**

I. Introduction

The contemporary digital design in the clock-synchronous technology is largely based on orchestrating the data movements between registers, with computation taking place in the combinatorial data paths, and the control implemented using cooperating finite state machines. The fragility of this approach–from the design, development, and debug viewpoints–stems from the formal disconnect of the data and control paths: each is specified in RTL HDLs (such as VHDL, Verilog) using different set of registers and wires.

In this work we propose a data-driven approach in which the computation is triggered by the availability (arrival) of data. We call the technique the *Flow-Transfer Level* (FTL) technique. The data-driven approach is by no means a new idea; it is natural in asynchronous circuits technology wherein the data arrival event serves to trigger computation in a physically self-timed block. In the clock-synchronous technology the advantages are less apparent. The initial aim of our research is to capture the communication dependencies among components (operators) in an explicit form, so that the cycle-by-cycle timing of the circuit is decoupled from its communication patterns. This by itself does not immediately improve the physical implementation results (resource costs, clock timing), but it could later uncover opportunities of higher-level optimisations. Our evaluation shows that the approach does not incur an overhead over manual RTL VHDL designs.

A couple of examples of data-driven circuits composed from primitive components, called *operators*, is in Figure 1. The patterns are: linear (sequential) pipeline, concurrent pipelines (using the Fork/Join operators), alternate pipelines (the *if-then-else* block, using the Switch/Select operators), feedback pipeline (the *while-do* block). Similar patterns have been used in asynchronous systems [1]. In the clock-synchronous technology the delay of the primitive operators is conceptually zero time (they are combinatorial), and the delay of the storage *cell* operator is 1 clock cycle.

A. Handshake

In asynchronous technology a channel handshake event starts processing in a component. In *bundled data* protocols the *request* and *acknowledge* wires are commonly bundled with the data wires to form a communication channel. The protocol provides the time-frame for the data transfer. Besides that, the individual components typically exploit the protocol to regulate the flow of data by the back-pressure mechanism: a busy component would not complete a handshake on the input channel until the output channel has accepted the outgoing data.

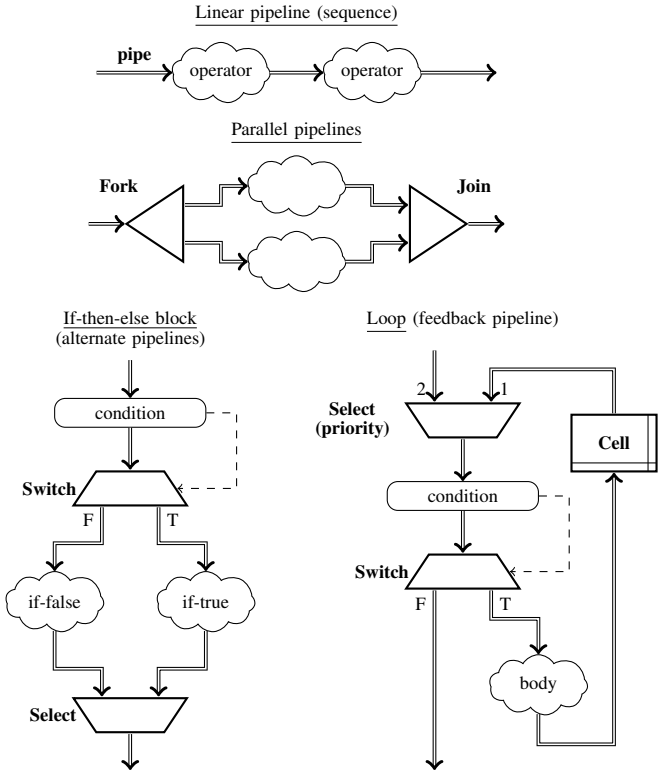


Fig. 1: Examples of data-driven pipeline patterns.

In the clock-synchronous technology the timing is provided implicitly by the clock signal. A handshake protocol is used exclusively to signal the data availability in the producer and the readiness in the consumer. Hence, just two *high-level* (logical) states, denoted ϕ here, must be recognized: either both the producer and consumer are ready simultaneously and the data *flows* between them ($\phi = 1$), or any one of them is not ready and the data flow stops ($\phi = 0$). The roles of producers and consumers are symmetrical and equal.

In FTL the communication channels are called *pipes*. Pipes are unidirectional for data, state-less (without internal buffers), and point-to-point (no branching). For data-transfer modelling in netlist graphs (e.g. in Figure 1), pipes are the edges, and operators are the nodes. However, for handshaking purposes, pipes are modelled as *nodes* in dependency graphs. The edges in dependency graphs are inferred

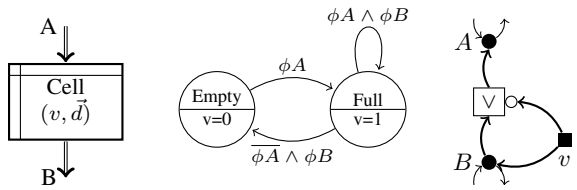


Fig. 2: The *cell* operator (left), its state transition diagram (middle), and the handshake control dependencies among the pipes (right).

based on the handshaking (control) behaviour in FTL operators.

B. Storage

A *cell* is a primitive operator that contains a storage place. An internal register stores a data vector $\vec{d} \in \mathbb{B}^n$ (where $\mathbb{B}$ is the boolean set $\{0, 1\}$) and a single validity bit $v \in \mathbb{B}$. The schematic symbol and a state diagram are in Figure 2 on the left and in the middle. A *cell* is basically a 1-deep FIFO queue, with a 1 clock-cycle latency. When it is empty ($v = 0$) it may store incoming data and become full. When it is full ($v = 1$) it will not accept incoming data *unless* the stored data is concurrently being off-loaded (accepted) in the consumer at the output. The semantics provides the necessary elasticity for data-driven circuits: data is preserved when not immediately required, and the stored data is not overwritten until consumed.

C. Problem Description and Motivation

1) Data-flow functional composition: Consider a trivial feedback pipeline constructed out of two *cells* depicted in Figure 3. The intuitive semantics is that the *cells* should periodically exchange the data values at each clock-cycle. This is consistent with the semantics and the state diagram given above. The technical obstacle is, however, that signalling dependencies in handshake controllers could close combinatorial loops in physical hardware implementation when pipelines converge. This problem exists e.g. in the SELF [2] protocol discussed below.

2) Property specification and optimization opportunities: Consider the example of an integer multiplier block, such as the one in DSP macros in FPGAs, depicted in Figure 4. The multiplier is combinatorial, the output data is valid only if both the inputs are valid at the same time. Data producers on both inputs must keep the values valid until the consumer at the output is ready to take the result. The functionality of the two-input multiplier block can be expanded into a *Join* operator and a single-input single-output compute operator. The *join* operator manages solely the handshake interplay of the three ports (A, B, C): it makes the producers on A, B and the consumer on C wait on each other, i.e. synchronizes all of them. It does *not* buffer data, nor it contains any storage element. The *join* operator is transparent for data: the data output vector $C.\vec{d}$ simply combines the data vectors $A.\vec{d}$ and $B.\vec{d}$.

Consider now a requirement to compute x^2 using the multiplier block. It is natural to use a *Fork* operator to duplicate X into A, B

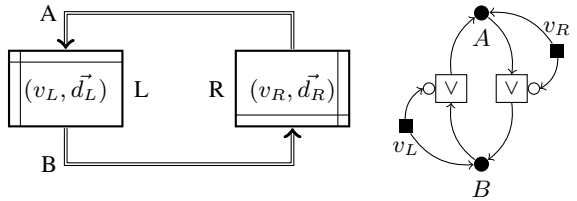


Fig. 3: Two-*cell* feedback pipeline (left), and the dependency graph derived from it (right).

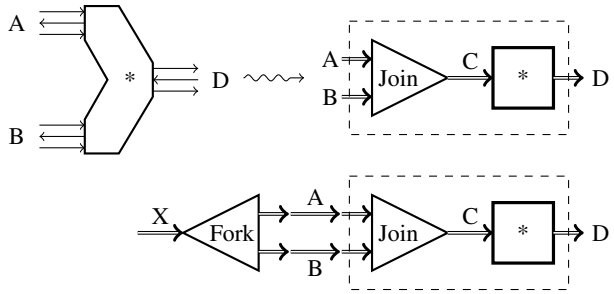


Fig. 4: A two-input multiplier requires both inputs being ready simultaneously to begin the processing. Below: To compute x^2 the pipe X is *forked*.

and connect them to the two-input multiplier block from a library, as shown in Figure 4 at bottom. The *fork-join* combination is a null operation (an identity) that could be later automatically optimized out in a tool once the design hierarchy is flattened. The *join* operator specifies a *property* that both inputs must be valid at the same time.

II. Related Work

Elasticity in circuits [3] provides tolerance to variations in computation and communication latencies. Although it is usually associated with an asynchronous design, it can be also implemented in fully clock-synchronous systems [4]. For example, an elastic adder can have data-dependent latency: it could take one clock-cycle for short operands and two cycles for operands requiring long carry-chain propagation. The SELF protocol [2] (*Synchronous ELastic Flow*) can be used to transform a synchronous circuit into the elastic one. This requires replacing original flip-flop registers with *Elastic Buffers*, implemented as special pairs of latch-based registers (master and slave). In the contemporary FPGA technology at least two flip-flop registers are needed to implement one elastic buffer.

Dataflow is a well-known design paradigm in which computation is triggered by the arrival of data. In *dynamic* data-flow the dependency graph evolves in time, possibly depending on actual data values, while in *static* data-flow the graph is predetermined. In *synchronous* data-flow (SDF) [5] the number of data tokens consumed and produced by each node on each invocation is specified a-priori. The static dataflow model is extensively used in the digital signal processing (DSP) domain to specify the behaviour of both the software (e.g. StreamIt [6]) and hardware components (e.g. CAPH [7]). These systems usually model each channel as a FIFO queue, and employ domain-specific knowledge in the compiler to optimize the granularity of tasks.

Term rewriting systems (TRS) can be used to describe the operational semantics of hardware circuits (TRAC [8], Bluespec [9]). The goal is parallel to ours–by increasing the abstraction level the designs are easier to comprehend and maintain. A TRS consists of a set of terms and a set of rewriting rules. A rule consists of a pattern, an optional predicate, and an action (a rewrite) that is carried out on the system state if the rule fires. The effects of actions are atomic; several terms may fire simultaneously only if their actions do not conflict. Term rewriting systems are defined as non-deterministic. During hardware synthesis the compiler extracts parallelism and creates a deterministic schedule of actions.

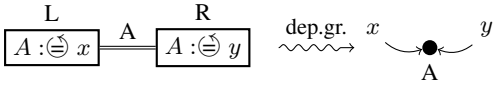
III. Handshake Formalization

This section describes the formal apparatus for specifying the *handshake* (as opposed to the data-processing) behaviour of the FTL operators. Capital letter symbols $A, B, C, ...$ denote pipes or ports. Their values in $\mathbb{B} \equiv \{0, 1\}$ indicate the handshake state: 1 means the handshake is successful (data is transferred), 0 means the opposite.

A. Dependency Graphs

As the *pipes* (⟹) propagate handshake dependencies among the endpoints simultaneously in both directions, we need a mechanism to resolve control cycles. Original FTL graph, composed of pipes and operators, is transformed into a *dependency graph*. In dependency graphs each pipe (an edge in the FTL graph) is replaced by a *wand* node (●), and each operator (a node in FTL graph) is replaced by edges. The dependency graph is typically cyclic at first. Using two basic transformations described below the cycles are gradually removed, and new *wand* nodes may be introduced during the process. The final acyclic dependency graph is mapped directly to a control state machine in the circuit.

The *wand* nodes in dependency graphs behave like *wired and*: all incoming edges are conceptually concatenated using the *and* (∧) boolean operator; all outgoing edges are driven by the result. In picture below there are two operators L, R, and a pipe A. The operators 'drive' the pipe handshake state to $x, y \in \{0, 1\}$. The corresponding dependency graph is shown on the right:



Dependency graphs may be given graphically, or textually with the help of an auxiliary operator :⃝. An expression '$A :⃝ x$', where A is a dependency node representing a pipe and x is a boolean expression, says that the handshake on the pipe A can successfully complete *unless* x is *false*. Operators may drive (and/or sense the state of) any pipe they connect to; multiple writes into the pipe handshake state are *unified*:

$$\{A :⃝ x; A :⃝ y\} \xrightarrow{\text{unify}} \{A :⃝ (x \wedge y)\} \tag{1}$$

Note how the symbol A in the equation above corresponds with a pipe in the FTL graph and with a *wand* node in the dependency graph. The symbols x, y, originally generated in distinct operators L, R, are brought together to form incoming edges in the *wand* node.

The simple diagram above could be given a higher-level meaning as follows: It shows a producer and a consumer operators in FTL (we do not say which is which) connected by the pipe A. Say $x = 1$, $y = 0$. If the producer is on the left, the diagram could be interpreted that the producer is ready to send data (1) while the consumer on the right is not ready (0). Or vice-versa: the consumer on the left is ready (1) but the producer on the right is not (0). The handshake is symmetrical. The result in any case is 0, i.e. data is not transferred because the handshake is not successful.

B. Eliminating Cycles in Dependency Graphs

Handshake dependencies can be cyclic; it is in fact the main feature of our approach. The key insight is that the handshake dependency is *not* necessarily a data dependency (which cannot be cyclic for obvious reasons), but rather a dependency in the *promises* to transfer *some* data. Hence, a cyclic handshake dependency among a group of operators is viewed *positively*, not as the obstacle to the flow of data.

Consider the example in Figure 5 with three operators (boxes) in a cyclic unidirectional handshake dependency. The dependency runs clockwise (C depends on A, A depends on B, B depends on C). Without any more incoming edges to the *wand* nodes the actual state of the nodes will be 1, and the corresponding pipes flow data.

The two elementary graph rewriting rules for cycles removal in dependency graphs are shown in Figure 6. The original dependency graph is partitioned in *strongly connected components* (SCC). The first rule ('cyc.elim') is applied to an arbitrary node A in an SCC; the SCC is reduced by one node. The rule is a form of substitution

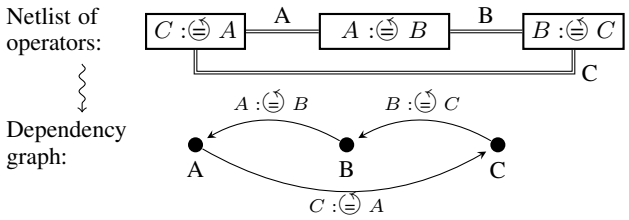


Fig. 5: Example of three operators with a globally cyclic handshake dependency.

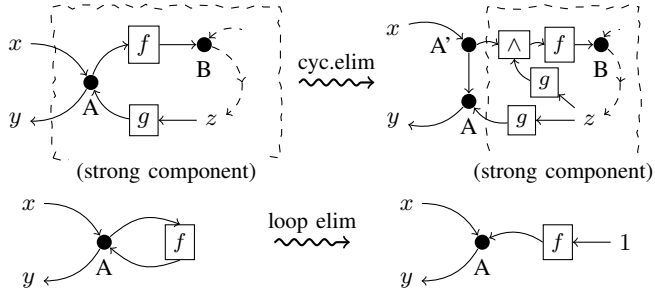


Fig. 6: The two elementary transformations for graph cycles elimination. A, B are *wand* nodes, f, g are arbitrary boolean functions.

of boolean expressions. The second rule ('loop elim') is applied to a node with a loop.

C. The Cell Storage Operator, Time

The *cell* handshake dependency sub-graph is shown in Figure 2 at right. The nodes A, B represent pipes, and the node v (■) represents the internal state (0=empty, 1=full). The internal state is updated separately and the driving of the node is not shown. The node '∨○' represents an OR gate with an inverted input. When $v = 0$, the node B is driven to 0, hence the pipe B cannot flow data. Meanwhile, node A is driven to 1 by the not-or gate, hence the pipe A could accept data if the producer is also ready. When $v = 1$, the node B is driven to 1, hence the pipe B could flow data if the consumer is ready. However, if the consumer is not ready and it drives B to 0, this is propagated to A which also becomes 0. This ensures that a full cell is not overwritten.

Time is a sequence of discreet, global, synchronous instants, commonly known as the *clock cycles*. The dynamic behaviour is specified only in operators with internal state. The *cell* internal state v evolves to the next time instant $t + 1$ according to:

$$t \to t + 1 : \quad v_{t+1} := \phi A_t \vee (v_t \wedge \overline{\phi B_t}) \tag{2}$$

The symbols ϕA_t and ϕB_t refer to the actual flow states of the pipes A, B after the system of the static handshake equations has been resolved.

As an example, Figure 7 shows how the initially cyclic dependency graph from Figure 3 is gradually resolved using the primitive elimination rules given in Figure 6. The final acyclic dependency graph on the right is the input to VHDL code generation.

IV. Evaluation

A. Compiler

The handshake behaviour of all primitive FTL operators is specified using dependency sub-graphs (such as Figure 2 right), and, if needed, by the additional dynamic time-dependent equation (such as Eq. 2). We implemented a prototype compiler in Python 3 that

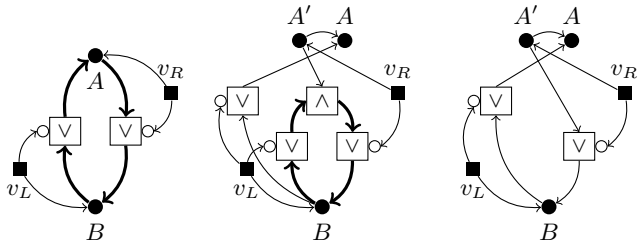


Fig. 7: Resolving the dependency cycle in the two-cell example in Figure 3.

transforms FTL netlist into synthesizable VHDL code. Our compiler first expands input FTL netlist into a global dependency graph. The graph is partitioned in strongly connected components (SCC); cycles in SCCs are gradually eliminated using the primitive rules in Figure 6. The final acyclic dependency graph is translated into a Mealy automata and printed as RTL VHDL code.

B. Design and Implementation of a Floating-point Vector Unit

We compare FPGA implementation results of a configurable pipelined 32-bit floating-point (FP) vector processing unit designed directly in VHDL with a design created in FTL. The original RTL VHDL code was created as a part of our previous work [10] using a classical RTL design flow, and optimized manually.

The central part of the FTL-based design of the vector unit–a reduction pipeline–is shown in Figure 8. The vector unit has two data input pipes A, B, and an output pipe Z. It can execute any of the following operations: vector addition, vector summation, minimum, maximum, index of the minima, index of the maxima. Only the first operation (vector addition) uses both the inputs. The later operations perform *reduction* of the input vector on A to a single scalar result. We used the FP adder core generated in the Xilinx ISE 12.4 tool with the configured latency of 3 clock-cycles. The vector reduction is therefore pipelined in the unit. As depicted in Figure 8 there are two feedback pipes: the one on the left routes adder outputs directly back to the input, the other on the right goes through the *cell* Q.

The FP adder core was generated in Xilinx Coregen 12.4, the latency was configured to 3 cycles. Functional correctness was verified using VHDL simulation. The VHDL code was synthesised in Xilinx XST 12.4, giving us the resource utilization report (Table I). The target device for synthesis was Spartan 6 S6LX150T. Resource utilization in the FTL-based unit is slightly smaller; we attribute it partly to a noise and partly to differences in coding style. The central pipeline contains a cascade of registers (or *cells*) parallel to the FP core. In the code generated by our compiler they are recognized during VHDL synthesis as a single shift register, which has an efficient implementation in FPGA, but in the original hand-written code they are synthesised into independent flip-flops.

TABLE I: Resource utilisation after FPGA synthesis.

Resource	FTL	manual [10]
Registers	308	430
LUTs (logic + memory)	489 (405 + 84)	719 (719 + 0)

V. Conclusion

The approach formalizes handshake in clock-synchronous designs. Although the communication pipes are unidirectional for data, the roles of producers and consumers are symmetrical (equal) when negotiating data transfers. Handshake dependencies can be transitive across a subgraph of operators. Cyclic handshake dependencies are resolved

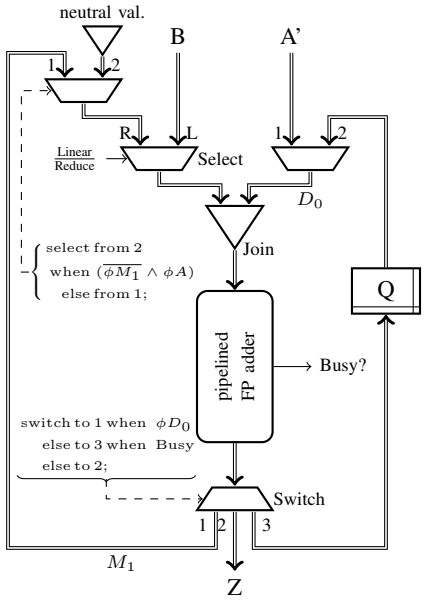


Fig. 8: The data pipeline implements primarily the feedback loop for pipelined vector reductions.

as they represent only a dependency in the promises to transfer data. This enables modelling of feedback pipelines.

The implementation results of a vector unit designed manually in RTL VHDL and in FTL show that the approach does not incur overhead in resource costs or critical path. In this particular case study the resource costs in the FTL-based design are even smaller than in manual approach because our compiler generated code is easier to optimize in the FPGA synthesis tool.

References

[1] J. Sparso and S. Furber, *PRINCIPLES OF ASYNCHRONOUS CIRCUIT DESIGN – A Systems Perspective*. Springer, 2001.

[2] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proceedings of the 43rd annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: ACM, 2006, pp. 657–662. [Online]. Available: http://doi.acm.org/10.1145/1146909.1147077

[3] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 10, pp. 1437 –1455, oct. 2009.

[4] G. Hoover and F. Brewer, "Synthesizing synchronous elastic flow networks," in *Design, Automation and Test in Europe, 2008. DATE '08*, march 2008, pp. 306 –311.

[5] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, sept. 1987.

[6] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *In International Conference on Compiler Construction*, 2001, pp. 179–196.

[7] J. Serot, F. Berry, and S. Ahmed, "Implementing stream-processing applications on FPGAs: A DSL-based approach," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, sept. 2011, pp. 130 –137.

[8] J. C. Hoe and Arvind, "Hardware synthesis from term rewriting systems," 1999.

[9] N. Dave, M. Ng, and Arvind, "Automatic synthesis of cache-coherence protocol processors using Bluespec," in *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, july 2005, pp. 25 – 34.

[10] J. Sykora, L. Kohout, R. Bartosinski, L. Kafka, M. Danek, and P. Honzik, "The architecture and the technology characterization of an FPGA-based customizable application-specific vector processor." in *DDECS*. IEEE, 2012, pp. 62–67.