High Level Synthesis for Data-Driven Applications

Etienne Bergeron, Xavier Saint-Mleux, Marc Feeley, Jean Pierre David
Département d'Informatique et de Recherche Opérationnelle
Université de Montréal
{bergeret,saintmlx,feeley,david}@iro.umontreal.ca

***Abstract*— John von Neumann proposed his famous architecture in a context where hardware was very expensive and bulky. His goal was to maximize functionality with minimal hardware. Presently, logical gates are nearly free and single chips will soon contain billions of gates. However, most current designs are still based on Von Neumann's architecture because processors are built on this model. Nevertheless, the main current challenge is to be able to design, refine, synthesize and verify new architectures in a minimum time and with a maximum computational performance regardless of the gate count. Data driven architectures enable a high level of parallelism because instead of a single controller managing all the resources (and often a single ALU), tens or hundreds of small controllers can now operate in parallel on local processing units. This paper presents an environment for the high level description, refinement, synthesis and verification of such systems. Our own HDL is presented with its compiler and we show how it can be used as the intermediate language of a compiler for an even higher level functional programming language. Ongoing work will enable the interfacing with other languages (from both hardware and software communities). We also intend to target asynchronous designs.**

I. Introduction

John von Neumann imagined a new way to program the ENIAC machine in 1948 : storing a program in ROM and letting the machine execute the now well known fetch-decode-execute cycle. Thanks to this improvement, he reduced the programming time from several days to a few hours. It was the starting point of a long era of control dominant architectures, which still lasts today.

A new approach was introduced in the 70's by Dennis [1], who proposed the first token machine, an architecture where the operations are triggered by data instead of a central controller managing all the computations. Data move on a graph in the same way as tokens in a Petri net. This architecture theoretically offers the highest degree of parallelism achievable because the operations start as soon as data are ready. The main bottleneck was the implementation of data matching which is required before a (multiple-operand) operation is executed.

Both architectures have evolved but the superscalar processor exceeded token machines in the 90's by integrating the token machine approach but on a restrained data set. More generic than token machines and also much more pervasive in the industry, the central control won this battle.

The present technological context is however completely different compared to 1948. Billions of transistors consuming a few Watts are available in a single chip. Reconfigurable computing allows the implementation of new architectures within seconds. A chip can run several years before encountering a failure. The main challenge now is to be able to manage the implementation of so many resources in a proven safe way and quickly obtain maximum computation efficiency and minimum power consumption.

The major alternative to the processor is the design of dedicated chips in ASIC or FPGA. VHDL [2] and Verilog [3], which were developed in the 80's, continue to be the most widely used HDL in the industry even though they are low level languages. Significant work has been done to develop higher level languages capable of targeting hardware : HandleC [4], HardwareC [5], Transmogrifier C [6]. But C-like languages have complex semantics that make it difficult to formally prove the correctness of a design. Another approach aims to model the design at a higher level. SpecC [7], SystemC [8], SystemVerilog and eSys.Net [9] offer this kind of high level environment. But once validated, these designs must be refined, often manually, to get equivalent RTL code.

We believe that functional languages can take an important role in current hardware design because they are high level and safe languages, and are based on the same concepts as token machines and so can in principle lead to highly parallel architectures. But this is only achievable if the architecture is built from the algorithm description instead of trying to adapt the algorithm to fit a given architecture as was done up to now. Various approaches have already been presented in previous work (Confluence, Lava [10], Hydra [11]). To our knowledge, this is the first implementation allowing fully general recursion. In this paper, we describe an approach to automatically synthesize an algorithm described in a functional language by successive refinements.

The paper is organized as follows: Section II describes our new version of the CASM language and compiler, which will be used as an intermediate representation of a token machines. Section III proposes a methodology to compile a functional language into a specific CASM-based representation of token machine. The transformations are illustrated by examples throughout the whole document. Our conclusions and future work are presented in Section IV.

II. CASM Language

The CASM language is designed to ease the development of hardware components and applications by people with little or no knowledge of digital circuits. The design philosophy of CASM is to provide a safe language where the execution does not deviate from the language semantics. Traditional

HDLs allow a user to describe circuits whose behavior may become unstable or unpredictable under special circumstances. Glitches, gated or synthesized clocks inducing delays, interfaces between multiple clock domains, combinatorial loops and so on can induce non-deterministic behavior. CASM does not let programmers make this kind of (synthesizable) errors. All programs written in CASM are synthesizable, fully predictable and their execution conforms to the simulation.

Twenty years ago, programmers had to be conscious of the hardware because it was a major limitation in memory and time. A programmer had to optimize his assembly code to increase the speed and reduce the memory requirements as much as possible. Nowadays, most programmers are used to thinking about an algorithm as a flow chart or a finite state machine but very few of them are aware of what happens at the gate or RT level. Most existing HDL serve to describe the actual physical components and connections of the circuitry instead of presenting the algorithmic concepts. But it is difficult to implement these concepts in a common HDL without spending time understanding all the subtleties and physical limitations of hardware. These problems should not be the concern of programmers who are interested in speedup of a hardware implementation.

Algorithmic State Machines (ASM) charts provide an easy and intuitive way to describe small algorithms. Figure 1 represents the chart for Euclid's GCD algorithm. It is easy to understand the behavior, state by state, of the algorithm and thus understand how the problem is solved. Nevertheless, a graphical interface is impractical for large algorithms and is hard to manage.

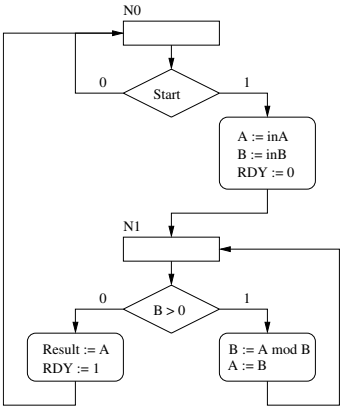

Fig. 1. ASM Chart : Euclid's GCD

To write this algorithm in a standard HDL, programmers must handle the interface of the components and implement a protocol to accept and confirm the input's reception (`inA`, `inB`) and `Result` transmission. Programmers must also correctly handle state transitions from state `N0` to state `N1` so that it only happens when data have been received at both inputs. The return to state `N0` must only occur when the result has been accepted by the outside world.

CASM is an intermediate level language of higher level than RTL (VHDL/Verilog) but which remains at a lower level than languages used for software development (C/Java). Our goal is to obtain a cycle accurate language that allows programmers to implement efficient algorithms with little formal training in CASM.

We based the semantics of our language on state machines. CASM is a textual representation of state machine charts augmented with higher level synchronization, communication, storage and recursion features, which will be described in the following sections. Because hardware can be modeled as a collection of parallel connected state machines (Mealy/Moore) [12][13], our language can be used to describe almost all sequential circuits including multiple clock designs.

```
1. input inA{protocol="FS"}[32];
2. input inB{protocol="FS"}[32];
3. output result{protocol="FS"}[32];
4.
5. asm {
6.   register A[32] = 0, B[32] = 0;
7.
8.   N0: A := ina; B := inb;
9.       goto N1;
10.
11.  N1: if (B>0)
12.          B = A mod B;
13.          A = B;
14.          goto N1;
15.      else result := A;
16.          goto N0;
17.      end;
18. }
```

Fig. 2. CASM Example : Euclid's GCD

Figure 2 shows the implementation of Euclid's GCD algorithm in CASM. The reception of inputs is specified on line 8 in state `N0`. Inputs are received with a full-synchronization protocol (`FS`). The transition to state `N1` occurs when all transfers required by state `N0` are done. Due to the "`result := ...`" on line 15, the receiver must also be ready to receive the result before the system can go back to state `N0`. While the implementation of protocols and conditional transitions are cumbersome in typical HDL, they are much easier in CASM.

The State Machine is a simple conceptual paradigm to represent an algorithm but it is still not expressive enough to easily be used as a programming language capable of describing large and complex components. CASM has builtin features that express frequently used high level concepts; these concepts are explained below.

Model

A CASM program is a collection of devices connected together by synchronized channels. The behavior of these devices is described by state machines. Every CASM file contains a device description and may contain instances of other devices. Figure 2 describes the behavior of the GCD device. The GCD device contains two registers that are explicit devices and an implicit state register. It also contains input and output channels and combinatorial operators. The model corresponding to this description is shown in Figure 3. A

transaction is a data transfer from a given source to a given target. They are activated by the transaction controller and actually occur when both source and target are ready to complete.

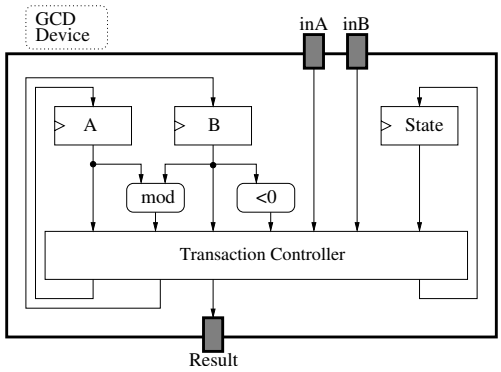

Fig. 3. Model of Euclid's GCD

The square boxes represent device instances, the rounded boxes combinatorial operators and the gray boxes device interfaces. Arrows are channels that contain both data and synchronization signals.

Connections and Transfers

Our language has two kinds of assignments: connections and transfers. Connections use the = symbol and transfers use the := symbol. A connection physically connects a source to a target enabling data transmission implemented by a predefined protocol. A transaction may occur each clock cycle. Transfers are a restricted version of connections where only a single transaction can (and must) occur before a jump to the next state is processed. In Figure 2, a connection can be found on line 12 and a transfer can be found on line 8.

Semantics

Each application written in CASM has an implicit `clk` and `nreset` signals that are used by devices and state machines. When the `nreset` signal is low, the component does a global reset. Steps are :

1) Initialize the devices with initial values.
2) Move all state machines to their initial states.
3) Wait until the `nreset` signal is deasserted.

In Figure 2, the registers will take the initial value of `0` (line 6) on a global reset and the state machine will move to the first state (`N0`).

When an event occurs on the `clk` signal, all the activated transactions are processed in parallel. The current state may be updated if a jump was also activated. From this new global state, the following evaluation steps are performed in preparation of the next clock event.

1) Evaluate conditional expressions and activate selected transactions.
2) Evaluate R-values and completion logic in parallel.
3) If all activated transfers can complete during current clock cycle, activate the jump to the next state.
4) Wait until next clock event.

In Figure 2, when in state `N1`, the conditional expression is evaluated to determine which of the two connections that continue the calculation (line 12) or the transfer that produce the result (line 15) must be activated. When the transfer is activated, the state completion condition depends on the readiness of the receiver. The transaction will not complete until the result is consumed. When the receiver is ready, the transaction and the jump are both activated to immediately move to state `N0` at the next clock event.

Self-synchronized data transfers

Data transmissions are done through channels using standard protocols. The channel contains implicit signals to synchronize both connected devices. The available standard protocols are:

- **Full-Synchronization** *(FS)* The transaction occurs only when both receiver and sender are ready.
- **Half-Synchronization** *(HS)* The transaction occurs when the sender is ready. Data is lost if the receiver is not ready.
- **No Synchronization** *(MAIN)* The transaction always occurs.

To avoid breaking transmission invariants and data loss, stronger protocols cannot be connected to weaker protocols. For example, an FS channel cannot send its data through a HS channel.

Extended Expressions

Expressions have an elaborate type system that includes protocol and variable widths. The type system enforces protocol (full-synchronized data is definitely consumed and never lost). Moreover, no precision is lost in arithmetic calculation without explicit programmer input, even if complex expressions are evaluated. The compiler infers the width through the expressions.

Protocols are automatically inferred through expressions. As an example, to perform the sum of two full-synchronized inputs, the system deduces that the two terms must be present.

CASM provides functions that simplify the implementation of data-driven applications. Functions like `queue`, `buffer` and `pipeline` are used to implement pipelined expressions. The expressions `reg = pipeline{cycle=4}(a*b);` and `reg = queue(queue(buffer(c)-d) >> e);` are pipelined calculations. A queue acts as a pipeline register between two stages. Queue is a synchronized component and it automatically stalls when needed. A buffer consumes inputs when the sender is ready and thus frees the sender. It sends the data to the receiver as soon as it is ready, possibly in the same clock cycle. The pipeline function automatically produces a pipeline with its expression. The buffer size and the pipeline depth are parameterizable options.

Recursion

To increase the expressive power of our language, recursion is allowed. In addition to jumps, one can use the `call` instruction to branch to another state and keep a reference

of the return point of the subfunction. It resembles the JAL (Jump and link) instruction. The return state is the return point. In Figure 4, on line 12, there is a call to the fact state and the return point is the send state. The return, which is a branch to the first return point on the stack, occurs on line 15.

```
1. input n{protocol="FS"}[8];
2. output r{protocol="FS"}[32];
3. queue qn{size=16}[32], qr{size=4}[32];
4.
5. always { qn = n; r = qr; }
6.
7. asm fact{depth=1<<(n.width-1)} {
8.    stack args{size=fact.depth}[8];
9.    register p[32];
10.
11.   receive: p := qn;
12.            call fact; return send;
13.   send:    qr := p; goto receive;
14.
15.   fact:   if (p==0) p = 1; return;
16.           else args := p;
17.             if (fact.complete)
18.               p = p-1;
19.             end;
20.             call fact; return mul;
21.           end;
22.   mul:    p := args * p; return;
23. }
```

Fig. 4. CASM Example : Buffered Recursive Factorial

Recursive state machines must specify a stack depth because it is not possible to statically bound it in the general case. When state machines are not recursive, the language states that the stack depth is guaranteed to be large enough and programmers can safely omit this information.

Anticipative transaction evaluation

CASM can determine whether a transfer has already completed or can complete in the current clock cycle. The completion information can be accessed using the complete attribute. When associated to a state name, the meaning is that the jump is activated and will occur at the next clock event. This is illustrated in Figure 4, line 17 where p must not be modified before being pushed in the stack args. So when in state fact and (p!=0), the state machine waits until the queue args is ready to receive new data. As soon as the condition is met, the jump and the connection between p and p-1 are activated in order to complete all transactions in a single clock cycle. When a transfer is conditional on another complete signal, the system becomes anticipative. This is illustrated by the example in Figure 5.

```
1. if (t1.complete) t2: a:=x; end;
2. if (t2.complete) t1: b:=y; end;
```

Fig. 5. Anticipative transaction example

Transfer t2 is activated only if t1 completes in the current clock cycle but transfer t1 is activated only if t2 completes in the current clock cycle. There are theoretically two possible solutions : t1 and t2 complete together or t1 and t2 do not complete. The resolution of such ambiguities is processed by considering that *a-priori* all transactions complete. When this is impossible, conflicting transactions are iteratively deactivated until a solution is found. It may happen that no solution exists, or can be found by this heuristic, which generates an error at compilation time.

Generic Devices

Generic devices have predefined behavior and synchronized interface but nothing is stated about their implementation. Our system provides parameterizable generic devices for commonly used components such as memories, stacks, and queues. These components are so widely used that we provide a syntax that easily fits in a CASM program and different implementations. The system also allows users to provide an other implementation that better suits their requirements.

For example, in Figure 4 queues are declared on line 3. These queues will be used to buffer inputs and outputs. Inputs are received and added to the queue on line 5. Since the inputs are queues, the sender does not block when transferring its data. When a buffer is full, the sender is blocked until there is some space in the queue. An example of input consumption can be found on line 11.

A special syntax has been created to easily interface with these components. For example, the array operator is used to bind an expression to an address channel. So, one can use the x := mem[expr]; syntax to read something from memory and mem[expr] := expr; to write something to memory.

We also provide specific implementations for different FPGA technologies. Some FPGAs provide embedded memories, CPUs, or others components. Users can configure the behavior and the chosen implementation using parameters or let the compiler choose appropriate parameters.

III. Compiling Functional Programming Languages to CASM

An interesting application of the CASM language is as an intermediate language for even higher-level hardware-description languages. It is particularly attractive to use CASM's high-level synchronization mechanisms for expressing data-flow architectures. Consequently we have begun investigating the use of the CASM language for implementing a compiler for parallel functional programming (FP) languages. Given that we are in an early exploratory stage, we have chosen to use a small FP language with few builtin operators and a parenthesized prefix syntax based on the Scheme programming language [14], which is both easy to parse and extend if the need arises. Currently our FP language is purely functional (no side-effects are possible), lexically-scoped (variables refer to the closest enclosing declaration in the program source code), and strict (arguments are evaluated before the function is called).

There are several reasons why an FP language makes a good HDL. The formal semantics of an FP language is typically smaller and cleaner than that of an imperative language such as C and Java. This is helpful for verifying the correctness of programs and also provides a solid framework for the program transformations and optimizations that are performed

by the compiler, such as function inlining, loop unrolling, code motion, partial evaluation, and so on. FP languages are declarative in nature, which means that the source code is closer to a specification than an implementation. It is thus more likely that a subsystem designed for one application can be reused in another application with little or no change. The absence of side-effects makes it easy to exploit parallel execution. Finally, given the existence of compilation techniques for compiling FP languages to efficient machine code, it seems that a single FP language could be used as an approach for hardware/software co-design. Only one language needs to be learned and used by the designer, and the partitionning of the system into hardware and software subsystems can be done by the compiler with some assistance from the designer (through source code annotations) or possibly fully automatically based on design constraints (silicon area, processing throughput, etc).

Recursion and Higher-Order Functions

A fundamental problem in compiling a parallel FP language to hardware is the handling of recursion and higher-order functions (functions which take functional arguments or that return functions). These features are necessary for full support of the FP programming style. To be consistent with our goal of providing a high-level programming style we would like to avoid placing restrictions on the type and depth of recursion and the use of higher-order functions. A (practically) unrestricted recursion depth can be achieved using a single large memory to store a stack of call frames for each process. The problem with this approach is that the memory would be a bottleneck that limits the system's parallelism. Moreover the long access latency of a large memory will reduce the performance of recursion. Our approach is to use one small memory for each function return point. This way the access time is short and different processes can execute in parallel as long as they are executing different sections of the program. Memory fragmentation due to alignment is eliminated because the width of the memory is equal to the size of the frame associated to that particular return point. To allow several processes to coexist the frames are explicitly linked in a chain rather than using a contiguous stack representation.

Closures

The traditional representation of lexically-scoped functions is the closure; a piece of data containing the value of the function's free variables and a reference to the implementation of the function's abstraction (i.e. where the free variables are viewed as additional parameters). Through the continuation-passing-style (CPS) transformation [15], a functional program is converted to a form where function returns are emulated using function calls. After a CPS conversion, all function calls and returns correspond to tail function calls (for which no stack frame needs to be pushed on the stack). Each non-tail function call in the original program is transformed into a tail function call to the function with an additional "continuation" parameter, which is a closure that contains the values that are needed at the return point by the calling function. A tail function call to the continuation closure corresponds to returning a result to the return point. Thanks to CPS conversion, the implementation of recursion and higher-order functions boils down to the implementation of tail function calls and closure allocation/deallocation.

In hardware, a CPS converted function is a device that receives requests through an input channel. A tail function call corresponds to sending a request containing the parameters to the input channel of the device that implements the function. A device typically performs some internal computation on the parameters received and then sends a request to some other device (either to return a result or perform a non-tail call of the source program). Our implementation of closures consists in adding a small memory local to a device (the closure memory), wide enough to hold its free variables. A closure reference is the address of the free variables in this memory and some tag bits that identify the device that implements the function. The tag is used in the implementation of tail function calls to closures to direct a request containing the parameters and the closure's address to the appropriate device for this closure. Upon receiving such a request the device uses the closure address to lookup the free variables in the closure memory and combine them with the device's other parameters to perform the computation appropriate for this function. Note that a function with no free variables does not need a closure memory, but the tag bits are needed to distinguish it from other closures.

Memory Management

An entry in a closure memory must be allocated when a closure is created. All free entries are linked in a free list to simplify the allocation mechanism. When the closure is no longer needed for the computation it must be deallocated to make room for future closure creations. This is done by adding the deallocated entry to the free list. This operation could be performed with a garbage collector, like in software implementations of FP languages. However this is hard to implement in hardware, especially with several distributed closure memories. Instead we perform automatic deallocation of continuation closures (the closure entry is freed when the closure is called) and rely on explicit deallocation operations in the program for non-continuation closures.

An example: factorial

To illustrate the compilation of recursive FP programs to CASM, we use the factorial function. The input code supplied to the compiler is shown in Figure 6. The factorial function itself is defined on line 1 while line 4, which is the program's body, is a call to this function with a free variable (`n`) as an input. The resulting circuit will have a single input channel for the `n` parameter and a single output channel for the result, both full-synchronized. Annotations may be added to specify, for example, `n`'s width in bits.

The input code is first CPS converted, as described above. The next phase does a combined control and data flow analysis

```
1. (define fac
2.   (lambda (x)
3.     (if (= x 0) 1 (* x (fac (- x 1))))))
4. (fac n)
```

Fig. 6. Factorial function, as input

(0-CFA) [16], which attaches an abstract value to every variable that might refer to a function. Thus, every function call is annotated with a list of all functions it might jump to. Finally, the annotated code goes through closure conversion [17], which makes a closure's free variables explicit.

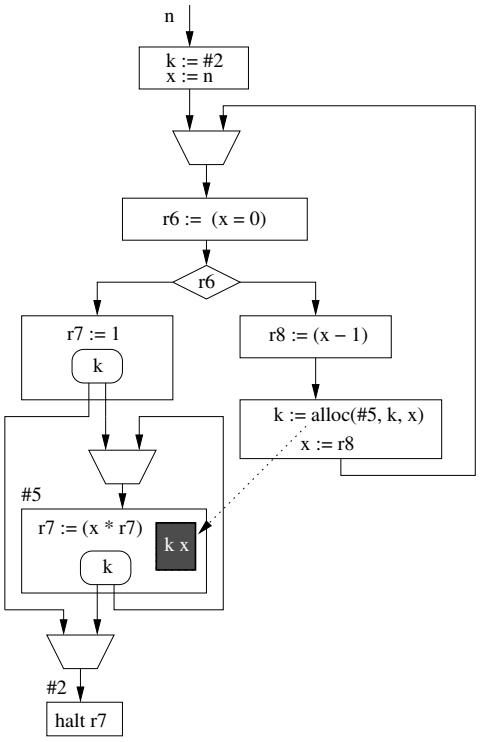

Fig. 7. CDFG for factorial

Figure 7 shows a Control and Data Flow Graph (CDFG) derived from the FP compiler's output. Processes flow through solid lines (data + control signals) while dotted lines denote closure allocations. Each rectangle represents an ASM device with a single state which performs basic calculations on its input (direct transfer, primitive operation or closure allocation) and sends the result to its output. Rectangles with rounded corners are function calls to closures: they send their input to either output depending on the closure's (`k`) tag bits.

This is illustrated in Figure 8, which represents the continuation to a recursive call to `fac` (closure #5); the continuation for this device may be the initial continuation (#2, output final result) or itself (#5), through merge nodes. At the top of the device is the closure memory. For an allocation (e.g. `alloc(#5, k, x)` in Figure 7), free variables are received on `alloc_data` and the corresponding address is sent back on `alloc_addr`. For a call, the address of the

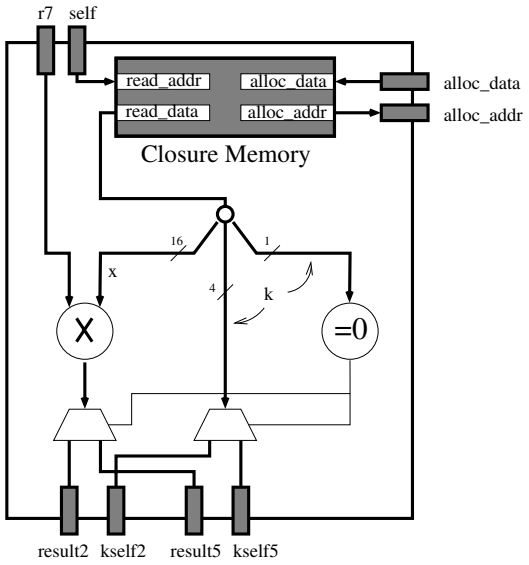

Fig. 8. Closure #5 device

free variables is received on `self`, with a result on `r7` since this is a continuation. With `self` directly connected to the closure memory, the corresponding variables (`x` and `k`) are automatically fetched. Then, `x` is multiplied with `r7` and the continuation (`k`) is called with the result. The call to `k` is represented by the multiplexers at the bottom of the figure: the tag part of `k` controls the multiplexers while it's address part flows through, along with the multiplier's result.

This is where CASM's high-level synchronization comes into play: the FP compiler's job is essentially limited to instantiating components and connecting them together; it is then guaranteed that for every (`self`, `r7`) pair received, a single output will be produced on either (`result2`, `kself2`) or (`result5`, `kself5`).

Also, the closure memory is an abstract component and different implementations can be created in minutes to fit a project's specific constraints (target technology, device size, required performance . . .) An example of such an instantiation is shown in Figure 9. This one targets FPGA, with a priority on speed rather than space; power consumption *et al.* are non-issues. It implements a simple "free-list" algorithm with separate memories for closure data and pointers so that faster memories can be used (in LUTs, single-port, no read-on-write) and any transaction can be completed in a single cycle. Every pointer has one bit added to indicate a full memory: the last element of the free list is the only one to have this bit on. The pointer memory is instantiated on line 12 and it's initial value is taken from a file named "`initfile`". The `if` structure that starts on line 15 makes read (dealloc.) requests have a priority over allocation requests. Finally, lines 18 and 26 ensure that the "next" pointer is not changed until all other transactions have either completed or are ready to complete in the current cycle.

```
1. input  read_addr{"FS"}[AWIDTH]; //read
2. output read_data{"FS"}[CWIDTH];
3. input  alloc_data{"FS"}[CWIDTH]; //alloc
4. output alloc_addr{"FS"}[AWIDTH];
5.
6. define CWIDTH; //supplied on instantiation
7. define AWIDTH = pmem.width;
8. define MEM_DEPTH = 1 << AWIDTH;
9.
10. register nxfree[AWIDTH+1] = 0;
11. memory cmem{MEM_DEPTH}[CWIDTH];
12. memory pmem{MEM_DEPTH,"initfile"}[AWIDTH+1];
13.
14. asm {
15. S0: if(request(read_addr))
16.       read_data := cmem[read_addr];
17.       pmem[read_addr]:=nxfree.[AWIDTH-1..0];
18.       if(S0.complete)
19.         nxfree := 0::read_addr;
20.       end;
21.     elsif(request(alloc_data)
22.           && !nxfree.[AWIDTH])
23.       post addr = nxfree.[AWIDTH-1..0];
24.       cmem[addr] := alloc_data;
25.       alloc_addr := addr;
26.       if(S0.complete)
27.         nxfree := pmem[addr];
28.       end;
29.     end;
30.     goto S0;
31. }
```

Fig. 9. CASM source code for a closure memory

IV. Conclusion

Due to the recent technological evolution of computing hardware the problem of implementing a system has moved from mapping an algorithm to an existing architecture to that of constructing a special purpose architecture tailored to the algorithm. The goal of our work is to design hardware description languages that allow non-experts to easily implement high-performance provably correct systems by synthesizing them from a high level algorithmic description. The CASM language is inspired from the token machine paradigm where the system's operations are triggered by the presence of data. This gives a programming model where synchronization is implicit and it enables a high degree of parallelism between operations. CASM also supports recursion which is absent from all common HDLs.

Although CASM can be used for directly programming applications, we have also begun investigating its use as an intermediate representation for compiling functional programming (FP) languages to hardware. Indeed CASM excels at expressing data-driven dataflow systems that are conceptually related to FP languages. Our prototype FP compiler handles recursion and higher-order functions using the continuation-passing-style conversion and a combined control and data flow analysis (0-CFA). To maximize parallelism we use distributed memories and linked frames to implement the run time stacks of the processes. A functional closure is implemented by a device with a local memory containing the free-variables of the closure. Deallocation of closures is handled automatically in the case of continuation closures and must be performed explicitly by the programmer otherwise.

Once our compilers are fully operational we plan to evaluate their ease of use and performance by implementing realistic systems. Of particular interest is the speed and size of the resulting circuit compared to other HDLs. We anticipate some extensions to CASM and our FP language to improve their ease of use. Various optimizations are possible to the CASM compiler to minimize number of registers, delays, connections, etc. We also plan to target various back-ends including asynchronous designs, which seems to be the best known approach for designing low power circuits.

Our FP language will be extended with programmer declarations to allow the compiler to determine through analyses the width of data paths and closure memories (type annotations and type inference), the size of memories (maximal recursion depth and number of processes), and absence of deadlocks when multiple recursive processes are used. Our FP language also needs a way to link to devices written in other HDL (CASM, VHDL, etc.).

V. Acknowledgment

This work was supported by the National Sciences and Engineering Research Council of Canada (NSERC) under its Discovery Grant program.

References

[1] J. Dennis and D. Misunas, "A preliminary architecture for a basic data-flow processor," in *2nd ISCA*, January 1975, pp. 126–132.

[2] S. A. B. et all, *IEEE standard VHDL language reference manual, IEEE Std 1076-2002*. The Institute of Electrical and Electronics Engineers, Inc., May 2002.

[3] M. M. M. et all, *IEEE Standard Verilog Hardware Description Language, IEEE Std 1364-2001*. The Institute of Electrical and Electronics Engineers, Inc., September 2001.

[4] Celoxica, "HandleC Home page." [Online]. Available: http://www.celoxica.com/methodology/handelc.asp

[5] D. Ku and G. D. Micheli, "HardwareC: a language for hardware design," Computer Systems Laboratory, Stanford Univ., Stanford, Calif, Tech. Rep. SCSL/CSL/TR-90-419, August 1990.

[6] D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs," *Proc. Symp. on FPGAs for Custom Computing Machines*, pp. 136–144, April 1995.

[7] J. Zhu, D. D. Gajski, and R. Doemer, "Syntax and semantics of the spec C+ language, Tech. Rep. ICS-TR-97-16, 1997. [Online]. Available: citeseer.ist.psu.edu/article/zhu97syntax.html

[8] M. B. et all, "SystemC 2.0.1 Language Reference Manual," 2003.

[9] J. Lapalme and E. M. Aboulhamid, "eSys.Net Home page." [Online]. Available: http://www.esys-net.org

[10] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*. ACM Press, 1998, pp. 174–184.

[11] J. J. O'Donnell, "From transistors to computer architecture: Teaching functional circuit specification in hydra," in *FPLE '95: Proceedings of the First International Symposium on Functional Programming Languages in Education*. Springer-Verlag, 1995, pp. 195–214.

[12] G. H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Tech J. vol 34*, pp. 1045–1079, September 1955.

[13] E. F. Moore, "Automata Studies," *Annals of Mathematical Studies*, 1956.

[14] R. Kelsey, W. Clinger, and J. Rees, "Revised^5 Report on the Algorithmic Language Scheme." [Online]. Available: http://www.schemers.org/Documents/Standards/R5RS/

[15] J. Guy L. Steele, "Rabbit: A compiler for scheme," Tech. Rep., 1978.

[16] O. G. Shivers, "Control-flow analysis of higher-order languages of taming lambda," Ph.D. dissertation, 1991.

[17] A. W. Appel and T. Jim, "Continuation-passing, closure-passing style," in *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1989, pp. 293–302.