Application-Specific Programmable Control for High Performance Asynchronous Circuits

Hans Jacobson and Ganesh Gopalakrishnan

Abstract— **The advantages of the programmable control paradigm are widely known in the design of synchronous sequential circuits: easy correction of late design errors, easy upgrade of product families to meet time to market constraints, and modifications of the control algorithm, even at run-time. However, despite the growing interest in asynchronous (self-timed) circuits, programmable asynchronous controllers based on the idea of microprogramming have not been actively pursued. In this paper, we propose an asynchronous microprogrammed control organization (called a *microengine*) that targets application-specific implementations, and emphasizes simplicity, modularity, and high performance. The architecture takes advantage of the natural ability of self-timed circuits to chain actions efficiently without the clock-based scheduling constraints that would be involved in comparable synchronous designs. The result is a general approach to the design of application-specific microengines featuring a programmable datapath topology that offers very compact microcode and high performance—in fact performance close to that offered by automated high-level synthesis tools targeting state-of-the-art asynchronous hard-wired controllers. In performance comparisons of a CD-player error decoder design, the proposed microengine architecture was 26 times faster than the general purpose hardware of a 280 MIPS microprocessor, over 3 times as fast as the special purpose hardware of a low-power macromodule based implementation, and was even slightly faster than a finite state machine based implementation.**

Keywords— **programmable control, microprogram, microcontrol, microengine, chaining, asynchronous circuits, self-timed, application-specific, architecture, ASIC**

I. Introduction

With the resurgence of interest that asynchronous circuits have experienced lately design methodologies are becoming mature and encouraging results are being obtained by many groups in designing self-timed circuits, for example in communications components used in multiprocessors [1], hardware to network portable electronic devices [2], and digital signal processing algorithms used in audio-electronics hardware [3]. Despite the growing interest in asynchronous circuits, programmable asynchronous controllers based on the idea of *microprogramming* have not been actively pursued. Since programmable control is widely used in many synchronous commercial ASICs to allow late correction of design errors, to easily upgrade product families, to meet the time to market, and even effect run-time modifications to control in adaptive systems, we consider it crucial that self-timed techniques also should support efficient programmable control. For example, supporting families of component types, such as bus adaptor chips, is greatly facilitated by programmability. Other examples of systems realized using programmable control (but not using asynchronous control) are the S3MP processor [4] which uses a microprogram engine, and the FLASH processor [5] which uses a processor core.

Although recent work has started to explore programmable designs in the form of asynchronous microprocessor cores, little work has been done in exploring general methods of realizing *application-specific* programmable structures. We believe such structures fill an important part of the design spectrum of programmable asynchronous circuits as special purpose hardware typically is an order of magnitude faster than the general purpose hardware of microprocessors and processor cores. One problem with realizing high-performance application-specific programmable structures, and maybe one reason that such structures have not been pursued previously, is the problem of efficiently orchestrating control of the global nature that microprogrammed designs require, to synchronize the propagation of new microinstructions. Thus it is often argued that programmable methods incur too much control related overhead to be a serious competitor to hardwired approaches for high-performance ASICs. Part of this work will identify some of the reasons why programmable control may introduce overhead and present approaches that in part can overcome these problems. We will demonstrate that these approaches indeed make it possible for application-specific microprogrammable structures to approach the performance levels normally only associated with hardwired control. In a nutshell, self-timed execution and microprogramming seem to go hand-in-hand.

In this work we propose a general and structured approach to a fully asynchronous microprogrammed control organization [6], a *microengine*, that targets application specific implementations. The approach emphasizes simplicity, modularity, compact microcode, and high performance. Natural properties of self-timing, such as the ability to efficiently chain computations, not easily achievable in similar synchronous designs, are explored to achieve these goals. We demonstrate that such application-specific microprogrammed structures can be easily designed for many classes of circuits, often perform at least an order of magnitude better than general-purpose solutions based on processor cores, and even approach the performance of state-of-the-art asynchronous hardwired control.

After surveying related work, Section II identifies performance problems in asynchronous programmable structures and motivates our approach for realizing efficient programmable control. Section III describes the operational details of our proposed asynchronous microengine architecture using a differential equation solver design as an ex-

The authors are with the Department of Computer Science, University of Utah, Salt Lake City, U.S.A. E-mail: hans@cs.utah.edu, ganesh@cs.utah.edu.

Supported in part by NSF MIP-9622587

ample. The microengine organization is then discussed in more detail and optimizations to enhance its performance are outlined in Section IV. In Section V, a detailed performance comparison between our microengine and state-of-the-art asynchronous hardwired controllers for a CD-player error decoder design is presented, followed by conclusions in Section VI.

A. Related work

Programmable asynchronous structures were first investigated around the 1980's [7] in the context of a data-flow computer. Of late, virtually all programmable asynchronous structures have been asynchronous microprocessors [8], [9], [10]. However, asynchronous microprocessors are not applicable in all embedded control systems, due to their high fabrication cost, large size, relatively high power consumption, and fixed general purpose instruction set. To illustrate the performance difference often present between general and special purpose hardware we implemented a CD-player error decoder [11] in our microengine architecture (presented later in this article) and also accurately estimated the best-case performance of the control algorithm of the same error decoder using the MIPS-R3000 instruction set as realized by the 280 MIPS asynchronous microprocessor presented in [9]. The performance difference using the same implementation technology, a 0.6 micron fabrication process, was a factor of 26 times in favor of our microengine.

Other programmable control approaches have recently been investigated [12], [13], [14]. These are best characterized as programmable microprocessor cores. These approaches are general purpose in nature, although for example [12] allows a dedicated datapath unit to be added to the core to speed up computation. However, this organization has a large area due to its on-chip caches (16k instructions, 64k data) to support general purpose microprograms, besides not being easily adaptable to specific design requirements. Self-timed FPGAs such as Triptych [15] also suffer from area and performance disadvantages compared to asynchronous microengines.

II. Architecture motivation and overview

A. Performance considerations

It is often argued that microprogramming incurs too much control overhead, as microinstructions have to be fetched and further decoded. In order to approach the performance of hardwired implementations whose control structure consists solely of finite state machines, it is important for microprogrammed structures to reduce the overhead of fetching microinstructions. One way to achieve this goal is to minimize the number of microinstructions needed to perform a task. In other words, it is better to perform more work per microinstruction, similar to Very Long Instruction Word (VLIW) architectures. We achieve this by using a *programmable datapath topology* that can dynamically schedule computation units in parallel and serial clusters, to best suit the current situation. Forming such serial clusters *dynamically* is virtually impossible in synchronous microengines because there, the propagation delays of all serial partitions of combinational modules must add up to an integral multiple of the clock period. This is akin to dynamically performing *chaining* [16], and is very difficult in practice. With self-timing, each chain of computations can be organized around request/acknowledge flow-through, as explained later.

In contrast to microprocessor cores, the implementation of a microengine can be adapted to and optimized for the given design specification, rather than the specification being adapted to an existing processor core. In our approach based on microengines, we target implementations where both program store and datapath units are customized to the problem at hand, a dynamic topology is supported, and designers have control over the degree of programmability. The programmability can be constrained to a degree that meets the designs performance or area criteria by controlling the number and types of datapath resources and the topologies in which they can be connected. The per microinstruction programmability of the datapath topology allows actions to be chained, which effectively allows rolling many microinstructions into one, considerably reducing the number of microinstructions needed to perform a task, and subsequently the overhead of fetching them. For example, for a differential equation solver 4 microinstructions of 25 bits width realize the entire control algorithm, and for a CD-player error decoder 9 microinstructions of 30 bits width constitutes the whole algorithm (both designs are presented later in this article). The microengine also features a modular datapath which allows easy replacement of datapath functional units thus facilitating upgrading and late binding of design decisions. Similar changes can, in a synchronous design, obviate the clock schedule, thus requiring total redesigns. The overhead of fetching microinstructions can be further reduced by prefetching the next microinstruction in parallel with datapath evaluation. The acknowledge synchronization can also be hidden in concurrent propagation of the next microinstruction thus allowing setting up and propagating data through multiplexors in the datapath.

B. Architecture and operation overview

A conventional (*synchronously clocked*) microprogrammed control structure consists of a microprogram store, next address logic, and a datapath. Microinstructions form commands applied on the datapath and control flow is handled by the next address logic that, with the help of status signals fed back from the datapath, generates the address of the next microinstruction to be executed. In a synchronous realization the execution rate is set by the global clock which must take the worst case delay of all units into account. When the next clock edge arrives it is assumed that the datapath has finished computing, the next address has been resolved, and the next microinstruction can be propagated to the datapath. Our *asynchronous* microengines have an organization similar to those of conventional synchronous microprogrammed controllers. How-

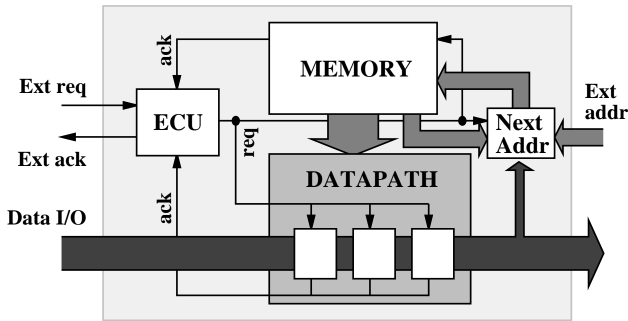

Fig. 1. High Level Structure

ever, as illustrated in Figure 1, major differences between these approaches stem from the use of handshaking to orchestrate both datapath as well as microprogram store related activities.

In conventional synchronous microprogrammed controllers, the computation is started by an arriving clock edge and the datapath is assumed to have completed by the following clock edge. In the asynchronous case we have no clock to govern the start and end of an instruction execution. Instead a request is generated to trigger the memory to latch the new microinstruction and the datapath units to start executing. The memory and each datapath unit then signals their completion by generating an acknowledge.

A microengine that is quiescent is started by the environment sending a request to the *execution control unit* (ECU in Figure 1). The ECU then generates a request on the global request wire (req) which causes the memory to latch the first microinstruction, and the datapath to start executing. While the current microinstruction is being executed by the datapath, the next microinstruction is concurrently fetched predicting branches suitably. Before the next microinstruction can be propagated to the datapath, the acknowledges from the datapath units and memory must be synchronized to ensure they have all completed. This function is performed by the ECU which collects all acknowledge signals before generating a new global request that starts a new execution cycle of the microengine. This repeats until the microengine has finished the requested computation. The ECU then generates an acknowledge back to the environment, and the microengine then remains quiescent until a new request arrives from the environment.

Small FSMs are responsible for locally handling *Request/Acknowledge/Sequencing* (RAS) of their respective datapath units, as dictated by the current microinstruction. This supports a standardized way of programming the datapath topology. The datapath units themselves then communicate with their local RAS block by using standard two- or four-phase request/acknowledge protocols. This also makes the datapath modular which means datapath units can be easily replaced without changing any control structures.

III. Microengine operation

This section will present a straight-forward implementation that captures the essence of our microengine architecture and operation using a differential equation solver as a working example. Architecture optimizations to enhance performance will be discussed in later sections.

A. Differential equation solver

The differential equation solver [17] in Figure 2 is a popular benchmark that will be used throughout this section to illustrate the general operation of the microengine. The algorithm illustrated in Figure 2(a) implements the *forward Euler method* which is an iterative approach suited for hardware implementation. The algorithm numerically obtains the values of y satisfying the differential equation $y'' + 3xy' + 3y = 0$ where x ranges from $x(0)$ to a with step size dx. Three threads calculating y, y' (u in figure), and incrementing x are needed per iteration. The $x1$, $u1$, and $y1$ variables in Figure 2(a) represent shadow registers to allow concurrent computation of the three threads. Computing u requires two multiplications, an addition, and a subtraction operation. Computing y requires one multiplication and one addition, x requires only an addition, and evaluating the while loop condition requires a comparator.

To avoid unnecessary detail in the example it is assumed that the input port values are stable throughout the algorithm execution, and that the constant $3*dx$ is available on an input port. We decide to allocate one multiplier and one arithmetic unit for the calculation of u, a multiplier and an adder for y and x, and a comparator for the loop condition. The three threads of the algorithm can then be scheduled as illustrated in Figure 2(b). Dataflow is identified by wide shaded arrows while control sequencing, the propagation of the request signal through the datapath units, is illustrated by thin black arrows. As opposed to the general algorithm in Figure 2(a), this scheduling does not require any shadow registers. However, an extra register, t, to store the intermediate result of the computation of u is needed.

Only four microinstructions are needed to formulate the algorithm. The first instruction loads the X, Y, and U registers with their initial values and then tests the initial loop condition. The second calculates y and the second half of u while the third calculates x, the loop condition, and the first half of u. The second and third instructions are then repeated until the loop condition $x < a$ becomes false at which time the fourth instruction makes an unconditional jump back to the beginning of the program and signals the completion of the computation. The complete microengine implementation with associated microprogram is illustrated in Figure 2(c).

B. Microprogram structure

The following bit fields of the microprogram are used to control the global microprogram flow. The current address, *curr-addr*, specifies which microinstruction that is currently being fetched by the memory (but is not part of the instruction). The next address, *next-addr*, is only used

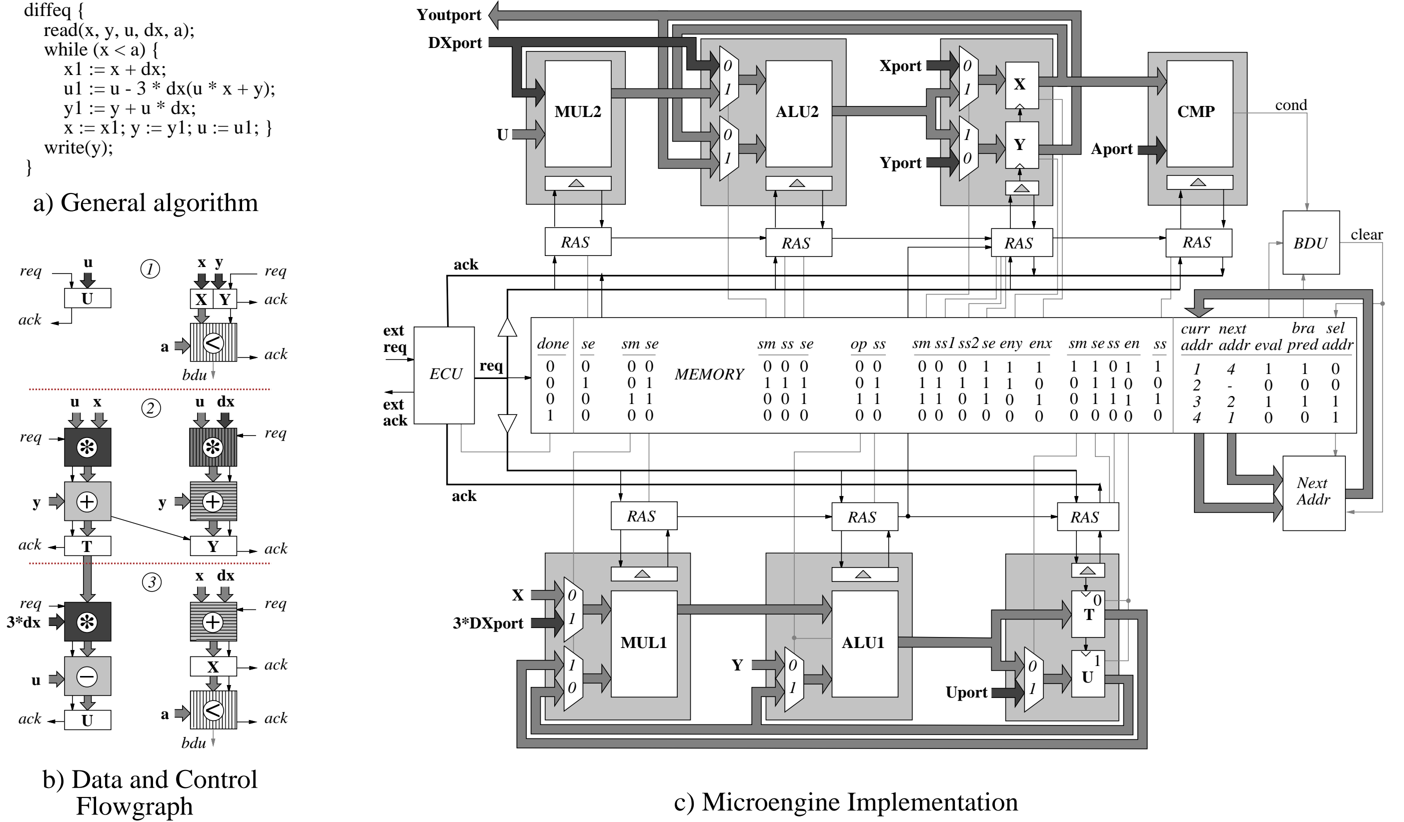

Fig. 2. Design Example: Differential Equation Solver

when the microinstruction contains a branch operation and specifies the address of the instruction being branched to. The *eval* bits specifies what conditional signals from the datapath that the branch detect unit (BDU) should test on a branch operation. The branch prediction, *bra-pred*, bit is used to specify if the branch test evaluation was predicted to be true or false. The select address, *sel-addr*, specifies which microinstruction, the next sequential one or the one specified by *next-addr*, to prefetch. The *done* bit indicates to the execution control unit when the microprogram has completed its computation and eventual data is available on output ports.

The following bit fields of the microprogram are used to control the local operation mode of each datapath unit (DPU). The set-execute, *se*, bits in the memory are used to specify when a datapath unit is supposed to execute while the set-sequence, *ss*, bits specifies if it is setup to execute in sequential (chained) or parallel mode. Note that if a datapath unit is setup to always operate in chained mode the *ss* bit also incorporates the functionality of the *se* bit. The set-mux, *sm*, and op-code, *op*, bits are used to specify which operands and operation the datapath unit should use. The enable, *en*, bits are used to enable which registers, when there are multiple registers in the same datapath unit, should latch data.

The logic blocks that different microinstruction bits operate on are indicated by the thin shaded lines connecting each logic block with its corresponding microinstruction bits in the memory block in Figure 2(c).

C. Local datapath control

To keep the datapath units modular and support a standardized way to implement sequential and parallel scheduling, a local control block associated with every datapath unit is introduced. These control blocks are represented by the RAS components as illustrated in Figure 2(c) and are responsible for handling request, acknowledge, and sequencing for their respective datapath unit. The possible combinations of sequential chains are easily identified in the figure by the horizontal arrows connecting the corresponding RAS blocks. Since the RAS blocks handle the control aspect of the datapath units, the microengine datapath forms a regular and modular structure where datapath units can be implemented in arbitrary styles, all using a simple request/acknowledge handshake protocol. In our example the datapath units, identified by the shaded boxes in the figure, are implemented in a standard gate library and use bundled data [18] delays for acknowledge generation.

D. Microprogram execution

The following paragraphs will step through the execution of the differential equation solver microprogram illustrated in Figure 2(c). The datapath units will be referred to by their internal components names. Thus XY refers to the

unit containing registers X and Y while *MUL1* refers to the unit containing the *MUL1* labeled function block etc.

Instruction 1. The microengine starts its execution at a specified entry point in the microprogram, address 1 in our example, upon receiving a request from the environment (*ext-req*). Bundled data is assumed in the communication between microengine and its environment, meaning the values on data buses are valid by the time the request arrives. The *Execution Control Unit* (ECU) receives the external request and in turn issues an event on the global request wire, *req*, fanning out to the memory and all datapath units. The microinstruction currently addressed, instruction 1, is then latched to a register array internal to the memory by the global request. The request fanouts to the datapath are sufficiently delayed to allow the microinstruction to propagate to the RAS blocks and datapath units first.

Datapath execution. When the global request arrives at the RAS blocks, those setup for parallel execution propagates the request to their corresponding datapath unit while those setup for sequential execution awaits the completion of previous datapath units in the chain. When the datapath units have completed their computation they generate an acknowledge to their respective RAS blocks. In our example, microinstruction 1 has setup datapath units XY and TU to latch the values on input ports *Xport*, *Yport*, and *Uport* in parallel. Datapath unit *CMP* is setup to await the completion of unit XY before starting its own computation. Instruction 1 thus executes two parallel threads, one thread containing units XY and *CMP* which are setup to execute in a chained fashion, and one thread executing unit TU. We represent this as $(XY \rightarrow CMP)||(TU)$.

As the XY and TU units complete their computation they generate acknowledges to their respective RAS blocks that in turn propagate the acknowledges back to the ECU. The RAS block acknowledges are also propagated as *sequential request* signals to other RAS blocks whose datapath units are setup for chained execution. The RAS block of datapath unit *CMP*, which is setup for chained execution, therefore waits until it gets a sequential request from the RAS block of unit XY, indicating that unit XY has completed its execution and that the values of registers X and Y are now available on its outputs. The sequential request is then propagated by the RAS to its datapath unit *CMP* which computes the conditional branch expression $X < Aport$ after which its acknowledge is sent back to the ECU. While the BDU tests the result of the branch expression the ECU synchronizes the completion of the datapath units.

Microinstruction prefetch. While the datapath is executing, the microinstruction predicted to be executed next is prefetched. If the current microinstruction does not contain a branch, the next address unit propagates the incremented value of the current address as the next microinstruction to be fetched from memory. If the microinstruction contains a branch, the prediction strategy is controlled by the *sel-addr* and *bra-pred* bits. If the *sel-addr* bit is set to a 1 the *next-addr* value is propagated, otherwise the current address incremented by one is propagated to the memory. In our example microinstruction 1 has the *bra-pred* and *sel-addr* set to 1 and 0 respectively, since it is likely that $X < Aport$ when entering the while loop, and address 2 is propagated to memory as the next microinstruction. After the memory has fetched the instruction it generates an acknowledge to the ECU and then waits for the next global request before propagating the instruction to the datapath.

If $X < Aport$ is false however, the prediction was wrong so microinstruction 2 must not be executed and microinstruction 4 be fetched instead. This is achieved by toggling the value of *sel-addr* if the *bra-pred* value is different from the evaluated branch result in the BDU the next time a global request arrives. An extra cycle is thus needed to fetch the correct microinstruction when a branch prediction is wrong.

Instruction 2. Assuming the while loop condition was true, instruction 2 is propagated to the datapath at the next arriving global request. As illustrated in Figure 2(b), instruction 2 contains two parallel threads. One computes the second half of $u : (MUL1 \rightarrow ALU1 \rightarrow TU)$ corresponding to the assignment $t = u * x + y$. The other computes $y : (MUL2 \rightarrow ALU2 \rightarrow XY)$ corresponding to the assignment $y = y + u * dx$. The chained request propagation in each thread commence as described previously for instruction 1. One difference however is the latching of Y. Since Y is an operand to *ALU1* we must at least make sure that *ALU1* has completed before latching the new value for Y (we assume T has time to latch its new value before the changes in Y propagates to its inputs). We therefore introduce a cross-thread synchronization point by requiring XY to wait for the completion of both *ALU2* and *ALU1* before latching the new value of Y. This is illustrated in the microinstruction by both set-sequence signals, *ss1* and *ss2*, for XY being set. Note that in the other thread TU still only has to wait for *ALU1* to complete. The TU thread can thus complete before the XY thread but never the other way around. It is worth observing the generality in which the microengine structure allows threads to be formed and synchronized. By letting several RAS blocks wait for the same sequential request(s), multiple threads can be spawned from a single thread. These threads can then be freely split into sub-threads or joined with other threads to form any combination of series/parallel clusters of executing datapath units. It is left to the designer as a performance/area/generality tradeoff to specify to which extent such formations should be supported. In our example, also note that since *MUL1*, *ALU1*, *MUL2*, and *ALU2* according to our scheduling can never be last in a chain, their RAS blocks are not required to generate acknowledges thus reducing the complexity of the ECU. Therefore only the RAS blocks for XY, and TU need to generate acknowledges this cycle. Since instruction 2 does not contain a branch, instruction 3 has been guaranteed correctly prefetched by the memory while the datapath was executing.

Instruction 3. Once the ECU has synchronized the acknowledges from the datapath instruction 3 is propagated to the datapath. This instruction also has two parallel threads. One computes the first half of u : $(MUL1 \rightarrow ALU1 \rightarrow TU)$ corresponding to the assignment $u = u - (3 * dx) * t$. The other increments x and tests the while loop condition : $(ALU2 \rightarrow XY \rightarrow CMP)$. This time no cross-thread synchronization is necessary and therefore only *ss1* for XY is set, i.e. this time the RAS block only waits for *ALU2* to complete before generating a request to the XY datapath unit. This instruction also contains a branch. Since the *sel-addr* bit is set the value of *next-addr*, which is 2, is specified to be propagated to memory as the address of the instruction to prefetch.

Instruction 4. While the loop condition holds true, instructions 2 and 3 are executed as described above. Once the condition becomes false, the *sel-addr* value is toggled and address 4 is propagated to memory. Instruction 4 contains an unconditional jump to instruction 1 and also indicates, by setting the *done* bit high, to the ECU that the computation requested by the environment has been completed and the y output value is available on port *Youtport*. The ECU then generates an acknowledge (*ext-ack* in figure) to the environment and then remains quiescent until the next request from the environment arrives.

IV. Architecture details

The following section provides a more in-depth discussion regarding the three key parts that have most impact, functionality and performance wise, on the proposed microengine architecture. These three parts consist of the next address logic, the global (ECU) and local (RAS) execution control and together capture the most essential parts of the microengine. Important architecture optimizations to improve performance will also be outlined at the end of this section.

A. Next address generation

To reduce control related overhead of the microengine, it is desirable to fetch the next microinstruction in parallel with the execution of the current microinstruction. We solve this problem of branch prediction in our microengine by fetching the next microinstruction most likely to be executed, but not committing it before the address selection has been resolved. We provide a flexible solution which allows each branch instruction to be individually programmed to employ a taken or not taken static branch prediction strategy. In order to keep the next address logic simple, the next address in case of a branch instruction is stored as part of the microinstruction.

Two units are involved in the next address computation. The *next address* unit is used to calculate the address of the microinstruction predicted to execute next at the start of the execution cycle. Based on status signals fed back from the datapath at the end of the execution cycle the *branch detection unit* (BDU) checks if the prediction was correct or not. The next address unit takes as input the *sel-addr* and *next-addr* signals from memory, the *clear* output

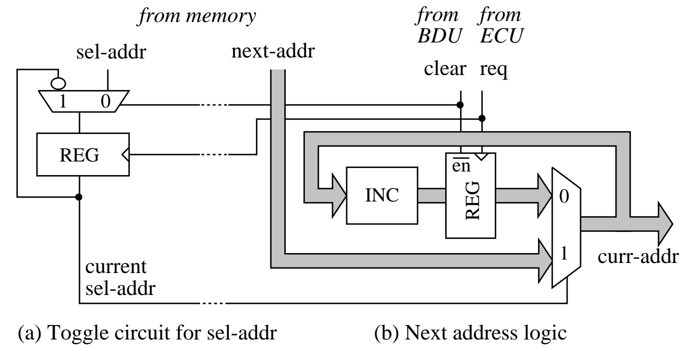

Fig. 3. Next Address Unit

of the BDU, and the global request from the ECU and generates the address of the next microinstruction to fetch as output. The BDU takes as input a set of conditional signals from the datapath, a set of evaluate signals from memory, and the predicted branch, *bra-pred*, signal from memory, and outputs a *clear* signal going to the memory and next address unit.

At the start of the execution cycle, the next address unit propagates the address of the microinstruction predicted to execute next. This is determined by the *sel-addr* bit from memory as illustrated in Figure 3(b). At the end of the execution cycle, the BDU evaluates if the branch condition is true or false. A set of evaluate signals from memory are used to select which conditional signals from the datapath (*eval* and *cond* in Figure 4), to test. The actual branch condition is then compared to the predicted one. If they differ the *clear* output is asserted. If asserted, the clear signal has three different functions. First, it is used to toggle the *sel-addr* bit from memory so that the next address unit propagates the correct address to memory as illustrated in Figure 3(a). Secondly, on the next global request, it will synchronously clear the *se* and *ss* bits of the microinstruction so as to not re-execute the old microinstruction. The *eval* and *bra-pred* bits are also cleared so as not to toggle the *sel-addr* bit again after fetching the correct microinstruction. The rest of the microinstruction registers are simply disabled from latching new values. Third, it is used to disable the next address unit from changing the values of its internal addresses so that the old incremented address, if selected, is propagated to the memory correctly.

In case of a mispredicted branch, the correct microinstruction is thus fetched once the *clear* signal has been asserted and the next global request arrives, thus requiring one extra cycle before the next computation can be started. A correctly predicted branch on the other hand has zero overhead. Unconditional branches are supported by specifying all *eval* and the *bra-pred* signals to be 0, thus guaranteeing that whatever microinstruction specified by the *sel-addr* bit will be fetched and executed.

B. Global execution control

The *execution control unit*, ECU, is the main control unit of the microengine. Its main task is to provide the

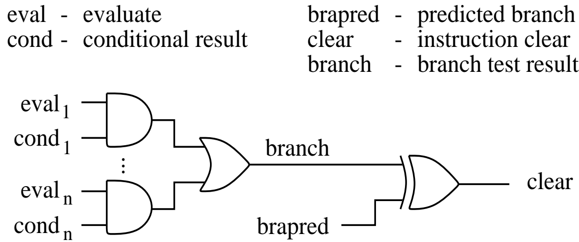

Fig. 4. Branch Detection Unit

global control needed in order to synchronize fetching of microinstructions and datapath execution. Thus its interface to the internal parts of the microengine is a global request signal that propagates the next microinstruction from memory and then triggers the datapath to execute, and the acknowledge signals from memory and all datapath units to detect the completion of the memory and datapath. The ECU also provides a request/acknowledge interface to the environment that is used by the environment to request the microengine to start executing, and by the microengine to acknowledge the end of the requested computation. The *done* signal from memory is used to indicate to the ECU when the microengine has finished its computation. The microengine uses the same handshake protocol for communication with its environment as with its internal components.

There are many ways of realizing a structure for request/acknowledge handshaking between the ECU and the datapath units. Since all current computation in the datapath must have finished before a new microinstruction can be propagated to the datapath, there is little to gain by generating separate requests to individual datapath units. A single global request fanning out to memory and all datapath units is therefore used. This approach reduces the complexity of the request control logic, as well as simplifies assumptions on parallel datapath unit operation and timing analysis. An initial performance concern was the capacitive load on this single request. Experience from implemented designs however, has shown the load to be of acceptable size and in fact smaller than some inputs of finite state machines.

Since all datapath units acknowledges must be synchronized in order to detect that the datapath has finished its computation, the design problem then reduces to one of designing request generation logic that offers low overhead and good scalability with regard to the number of datapath units. For implementation of the request generation logic, burst-mode [1], [19], [20] type of asynchronous state machines are used. The operation of a burst-mode state machine allows the acknowledge signals from the memory and datapath units to arrive at the state machine inputs in arbitrary order at arbitrary times.

For efficiency reasons we impose the requirement that all RAS blocks should always respond with an acknowledge even when their datapath units are not setup to execute. This will keep all acknowledges in phase and results in greatly reduced logic complexity for the request generation logic. By using this strategy the number of transistors of the request generation logic grows only linearly, to be more precise one transistor in the p and n transistor networks respectively, with the number of acknowledge inputs. If the acknowledges were allowed to get out of phase the logic would become much more complex. When using this approach of always acknowledging the RAS blocks must generate a bypass path for acknowledge generation when their datapath units are not scheduled for execution. The cost for this however is very small compared to the extra ECU complexity if the out of phase acknowledge approach was to be used. In addition, the same request generation logic can be used for both two- and four-phase protocols.

C. Local datapath execution control

A powerful feature of the proposed architecture is its ability to dynamically form clusters of datapath units for independent series/parallel execution during run-time. To support this fine grained control over execution, a limited form of control structure, the RAS block, is associated with each datapath unit as previously shown in Figure 2(c). The RAS block provides control over local request/acknowledge generation and sequencing of actions. Given the set-execute and set-sequence bits from the current microinstruction, the RAS block controls if its corresponding datapath unit is supposed to execute during this cycle and in what mode, sequential or parallel, with respect to other datapath units. In parallel mode, the global request is propagated directly to the datapath unit. In sequential mode, the sequential request (acknowledge) of the previous RAS block in the execution chain is propagated. If the datapath unit is not set to execute during the current cycle, a special bypass path is provided to generate a quick acknowledge.

A RAS block interface thus consists of the following. The *se* and a set of *ss* signals from memory that control its mode of execution, the global request and a set of sequence-request signals which are used to trigger the datapath unit to execute, and an acknowledge going back to the ECU (and as sequence-request signal to other RAS blocks). A standard request/acknowledge interface is used in the communication with its local datapath unit.

Request/acknowledge control. One responsibility of the RAS block is to provide means of correctly performing an internal request/acknowledge handshake with its datapath unit if it is scheduled to execute during the current cycle, and also provide a bypass path for acknowledge generation if it is not. A request signal should only be received by the datapath unit if it is supposed to execute during the current cycle. A *blocker gate* is therefore needed to block the request from propagating to the datapath unit if it is not setup to execute. Correct propagation of the internal request signal to the datapath unit can in the case of four-phase protocol be implemented by a simple AND-gate. The AND-gate is then enabled if the datapath unit is scheduled for execution, and disabled otherwise, respectively propagating or blocking the request generated by the sequence control. The request generation is more complicated for

the two-phase protocol, since the control must keep track of the value of the request signal last propagated through to the datapath unit. A logic block that can generate events to either the datapath unit, if it is scheduled for execution, or to the bypass path if not is therefore needed. The corresponding functionality is satisfied by a SELECT-element, which takes a level signal and an event signal, and generates an event on either of two outputs depending on the value of the level signal set-execute.

The bypass path, illustrated by the shaded components in Figures 5(a,b), can in the case of four-phase protocol be implemented by a MUX that directly propagates the global request signal as the acknowledge if the datapath unit is not scheduled for execution. In the case of two-phase a MUX cannot be used since the state (value) of the input signals are not known. A logic block that generates an event on its output whenever receiving an event on either of its inputs is therefore needed. An XOR-gate satisfies this behavior, and is then used to generate the acknowledge signal.

Sequence control. The other responsibility of the RAS block is to provide a standard way of implementing sequencing of actions. The sequence control function of the RAS can in its simplest form be performed by a MUX, controlled by the set-sequence bit from memory, that propagates either the global request or a sequential request to its datapath unit. The output of the sequence control MUX is hazard free since both the global and sequence request signals will reach the same values before the next microinstruction may alter the MUX control signal (signal *ss* in Figure 5). Carrying the above idea further along, in general it will be necessary for a RAS block to wait for the completion of an arbitrary set of concurrently executing datapath units before generating the request signal to its attached datapath unit. An efficient way to realize such high flexibility is illustrated by the complex gate structure on the left-hand sides of Figures 5(a,b). Given a set of set-sequence signals from the microinstruction and sequence request signals from other RAS blocks, this structure can synchronize with all possible combinations of these datapath units. The set-sequence signals provide a bypass path around the sequence request signals in the transistor stack that are not currently of interest. This forces the sequence logic to wait for an event on all sequence request signals in the current subset of interest before a path in the transistor network will conduct. Note however that regardless of the specified sequencing, the RAS blocks provide a fast return to zero by generating falling acknowledges in parallel when the four-phase protocol is used.

In general, sequencing actions between datapath units will always be faster than starting a new cycle, because the latter entails detecting completion of all datapath units and fetching a new microinstruction. To gain a significant performance edge however, the number of sequential request signals to a RAS should be restricted, as practical realizations seldom call for the "infinite flexibility" of all possible combinations.

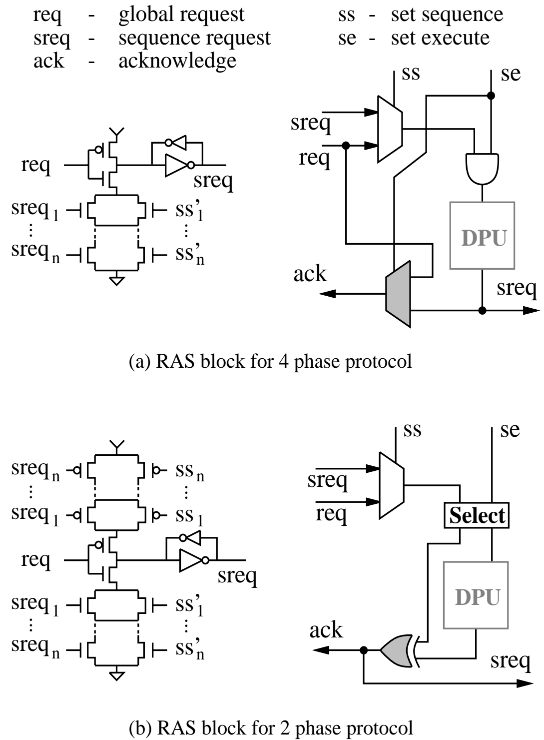

Fig. 5. RAS block structures

D. Architecture optimizations

The structure presented for the microengine control so far brings forth the high level concepts of the microengine architecture in a concise manner. However, it is not optimal seen from a performance point of view. Since the microinstruction is latched only once the ECU has synchronized the datapath completion and also must be allowed sufficient time to propagate to the datapath and setup the RAS blocks and datapath units, significant control related overhead is introduced. Also, since the microengine is required to synchronize with all datapath units before fetching the next microinstruction, significant computational overhead can be introduced in the datapath since the microengine has to wait for the longest thread to complete before starting the next cycle. The paragraphs below will briefly outline operational and architectural optimizations that can reduce the control and data computation overhead considerably, but are out of scope for detailed presentation in this article.

Reducing control overhead. Control related overhead can be reduced considerably by fetching the next microinstruction concurrently with the ECU performing completion synchronization. This can be achieved by, in the two-phase case, letting each RAS block be responsible for latching its own portion of the microinstruction directly after its datapath unit has completed its execution, and, in the four-phase case, latching the new microinstruction during the return to zero phase. These approaches also allow setup and propagation of data through input muxes of the datapath units while the ECU performs synchronization and the global re-

quest propagates through the RAS blocks. In most cases the microinstruction propagation to the datapath and data propagation through input muxes can be completely hidden in the ECU and RAS computations. The RAS blocks can also be optimized to yield lower latency. For example, the propagation of the global request through a four-phase RAS block can be reduced to the propagation delay through a single pass-gate. New functionality and hazard considerations for these optimizations are discussed further in [21].

Reducing datapath overhead. Although control overhead can be reduced considerably as mentioned above, there may still be significant computational overhead in the datapath since the microengine still has to wait for the longest thread to complete before starting the next cycle. This is not always desirable since long latency operations may block other, concurrent, operations that finish quickly and need to fetch a new microinstruction in order to continue their execution. We therefore introduce the concept of *decoupling* clusters of datapath units from the microengine operation during run-time. This allows the microengine to fetch new microinstructions and continue execution of non-decoupled datapath units without having to wait for the completion of the decoupled clusters. When the microengine needs the result of a decoupled cluster, it initiates the resynchronization with the cluster. As with the formation of series/parallel clusters, this decoupling of clusters and resynchronization with the same can be done on a per cycle basis. Further discussions regarding decoupled datapath units can be found in [22].

V. Design Example

To estimate the efficiency of the presented microengine implementation style compared to a hardwired control implementation using the same datapath structure, a CD-player error decoder [11] was built as a design example. In addition to the microengine style, the decoder was therefore also implemented using our high level synthesis framework for asynchronous circuits, ACK [23]. This framework takes a high level description in either the HOP language [23] (illustrated in figure 7) or Verilog−+, a synthesizable subset of Verilog extended to handle channels, as input and targets customized interacting burst-mode FSMs as control structure. The datapath being created by ACK was used in both implementations. The HOP design specification of the error decoder is a faithful translation of the Tangram program presented in [11] which also enables comparisons to the respective results obtained therein. Although the microengine design was implemented by hand, careful attention was given to ensure that the implementation corresponds to what would easily be achievable using an automated synthesis tool.

A. CD-player error decoder

The CD-player error decoder circuit implements error-detection on the audio information recorded on Compact Discs using a syndrome computation algorithm. Figure 6 illustrates the structure of the microengine implementation

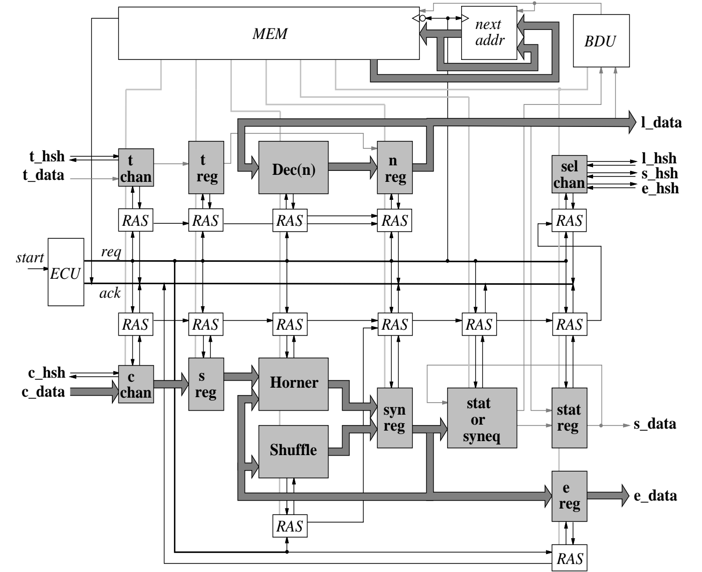

Fig. 6. CD-player error decoder structure

and Figure 7 the behavioral HOP language specification. The decoder processes a sequence of either 32 or 27 input words indicated by the value on the t channel. The words are read in, processed, and checked for errors in two sequential loops. The status of the decoding is then reported to the environment via the s, e, and l channels. Further details of the decoder can be found in [11].

To reduce the control overhead thus improving the performance of the design, several sequential chains are introduced. This significantly reduces the number of times the DPUs must be synchronized in order to fetch a new microinstruction, also reducing the number of instructions necessary. Since no DPU contains any precharged logic, only those DPUs that can actually end an execution cycle, i.e. any DPU accessed last in a chain, need to acknowledge their completion to the ECU. As can be seen in the figure, some RAS acknowledges can therefore be removed (5 out of 13), reducing the complexity of the ECU. The possible sequential chains are easily identified in the figure by the horizontal arrows connecting the corresponding RAS blocks.

The microengine execution proceeds as follows. A *start* signal from the environment causes the microengine to start by first loading a new microinstruction. This microinstruction is propagated to the datapath which then reads in a value from the t channel, initializes the n register accordingly, and resets the syn register.

The $SYNDROME$ loop, decoding the stream of n input words is then entered. This loop executes two chains in parallel, one that decrements the n-counter and one that reads in a new word and processes it in the $Horner$ procedure. The actions carried out by these chains can be viewed as follows where $\rightarrow$ and $||$ indicates sequential and parallel execution respectively.

$$(Dec(n) \rightarrow n_reg) \; || \; (c_chan \rightarrow s_reg \rightarrow Horner \rightarrow syn_reg)$$

The completion of the two chains are then synchronized by

```
Module   CD_PLAYER_ERROR_DECODER

Event      start??     : bit;
Channel    T?, S!      : bit;
Channel    C?, E!, L!  : array [7:0] of bit;
Variable   syn         : array [31:0] of bit;
Variable   e, s        : array [7:0] of bit;
Variable   n           : array [5:0] of bit;
Variable   t, stat     : bit;

Function  Horner (
   InPort   si     : array [7:0] of bit;
   InPort   syni   : array [31:0] of bit;
   OutPort  syno   : array [31:0] of bit; )
   { syno[7:0]    := GFadd(si, syni[7:0]),
     syno[15:8]   := GFadd(si, Alpha(syni[15:8])),
     syno[23:16]  := GFadd(si, Alpha(Alpha(syni[23:16]))),
     syno[31:24]  := GFadd(si, Alpha(Alpha(Alpha(syni[31:24])))) }

Function  GFadd (
   InPort   si, syni  : array [7:0] of bit;
   OutPort  syno      : array [7:0] of bit; )
   { syno := si XOR syni }

Function  Alpha (
   InPort   syni   : array [7:0] of bit;
   OutPort  syno   : array [7:0] of bit; )
   { syno[7:5]  := syni[6:4],
     syno[4:2]  := syni[3:1] XOR syni[7],
     syno[1,0]  := syni[0,7] }

Function  Shuffle (
   InPort   syni   : array [31:0] of bit;
   OutPort  syno   : array [31:0] of bit; )
   { (syno[7:0],syno[15:8],syno[23:16],syno[31:24]) :=
     (syni[15:8],syni[31:24],syni[7:0],syni[23:16])   }

Behavior

<START>  :  start?? ->  <INPUT>

<INPUT>  :  fork  <F0> : T?t ->  if (t == 0) -> n := 27 ->   <join>
                                 else -> n := 32 ->   <join>
            ||     <F1> : syn := 0 ->  <join>
            join  ->  <SYNDROME>

<SYNDROME> :  if (NOT(n[5] == 1)) ->
                 fork  <F0> : n := n - 1 ->   <join>
                 ||      <F1> : C?s -> syn := Horner(s,syn) ->   <join>
                 join  ->  <SYNDROME>
              else ->  <SHUFFLE> ;

<SHUFFLE> :  fork  <F0> : if (t == 0) -> n := 27 ->   <join>
                          else -> n := 32 ->   <join>
             ||      <F1> : e := syn[7:0] ->  <join>
             join ;
             syn := Shuffle(syn) ;
             syn := Shuffle(syn) ->  <ERR_CHECK>

<ERR_CHECK>  :  if (NOT((n[5] == 1) OR (syn[7:0] == syn[15:8]))) ->
                   fork  <F0> : n := n - 1 ->   <join>
                   ||      <F1> : syn := Horner(0,syn) ->   <join>
                   join  ->  <ERR_CHECK>
                else ->  <IS_ERR> ;

<IS_ERR>  :  stat := n[5] ;
             syn := Shuffle(syn) ->
             stat := (stat OR (syn[7:0] == syn[15:8])) ;
             syn := Shuffle(syn) ->
             stat := (stat OR (syn[7:0] == syn[15:8])) ->   <RESULT>

<RESULT>  :  S!stat, E!e, L!n ->   <INPUT> ;

EndBehavior

//*  ';' or conditional test indicates ECU synchronization point
//*  '->' indicates chaining of current and next statement
```

Fig. 7. CD-player error decoder HOP specification

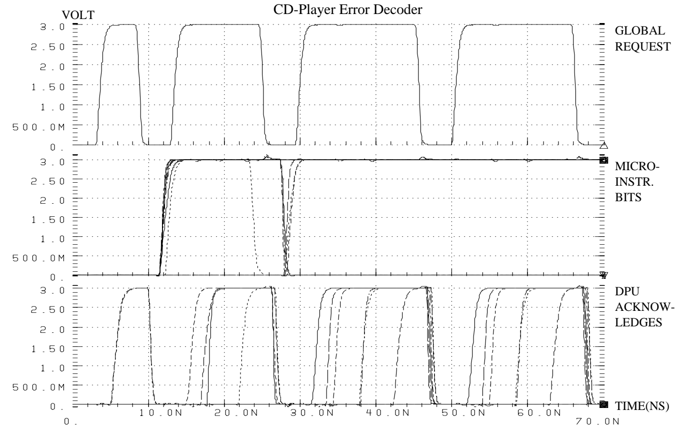

Fig. 8. CD-player error decoder SPICE waveforms

the ECU concurrently with the BDU testing the branch condition. The loop will continue to be iterated until the branch condition is false, that is, when n becomes negative. The syn register is then reshuffled to accommodate the input to the ERR_CHECK loop. This loop detects eventual errors in the decoded sequence caused by lost or misread bits from the CD. The action sequence in this loop is as follows.

$$(Dec(n) \rightarrow n_reg) \;||\; (Horner \rightarrow syn_reg \rightarrow stat_or_syneq)$$

The loop is iterated as long as n is non-negative and the two low end bytes of syn differ. If the loop terminates with a negative n, the word has multiple errors. Otherwise, depending on the value of e, either one or zero errors are present. The last computation action is then to set the status bit according to the error calculation which is done by two invocations of the following chain.

$$Shuffle \rightarrow syn_reg \rightarrow stat_or_syneq \rightarrow stat_reg$$

The status information is then communicated to the environment via the s, e, and l channels, containing the status of the computation, the starting word of the sequence, and the position of the eventual error.

Figure 8 illustrates a post-layout SPICE simulation of the initialization and first couple of cycles executed by the microengine. The top panel shows the global request signal, the middle panel shows the bits of the latched microinstruction and the branch clear signal, and the bottom panel shows the acknowledges of the datapath units. Note that this implementation uses the optimization of latching the microinstruction during the falling edge of the global request as discussed earlier. On startup, the microinstruction is initially cleared. On the first cycle after a request from the environment the microengine therefore only fetches the next microinstruction to be executed. Since all RAS blocks are in acknowledge bypass mode no datapath unit will execute and the branch clear signal will be low. The first microinstruction executes the fork-join statement in the $INPUT$ state of the HOP code in Figure 7 and tests the $SYNDROME$ loop condition. The use of chained execution can be observed by the sequentially occurring acknowledges in Figure 8's bottom panel. This nicely illustrates how the execution propagates through the threads of chained datapath units. Note the parallel return to zero of the acknowledge signals on the global requests falling edge. Since the $SYNDROME$ loop condition is initially true, the branch clear signal goes low as illustrated by the single falling dotted line in the middle panel of Figure 8. The next microinstruction implementing the $SYNDROME$ state of the HOP code is thus propagated to the datapath and executed next. No more changes are visible in the microinstruction bits since we continue to iterate over the $SYNDROME$ loop instruction until n becomes negative.

B. Result Comparison

The Tangram implementation described in [11], which was targeted for low-power, used double rail logic and a 5V 1.2 micron technology and was reported to have an approximate worst case cycle time of 20 microseconds, each cycle decoding a sequence of 32 8-bit input words, and a core area of 2.0 mm^2. According to [3] a factor of 1.5 in performance improvement and a 40% smaller area can be attributed to single rail over double rail in a Tangram implementation of a similar, but more complex, error decoder for the DCC player. With feature size scaling under constant field assumption [24], except for voltage, a single rail Tangram implementation of the CD-player error decoder in a 3V 0.6 micron technology could therefore be expected to have a cycle time of about 5 microseconds and an area of 0.3 mm^2.

Our design tool ACK was used to automatically generate a hardwired implementation targeting a 3V 0.6 micron CMOS technology and using a four-phase handshake protocol. As illustrated in Table I the corresponding post-layout cycle time as simulated with SPICE using worst case transistor models and temperature was, in the current implementation of ACK, 1.58 microseconds, with an area of 0.25 mm^2. Using the same datapath, the microengine implementation had a resulting cycle time of 1.46 microseconds also using a four-phase protocol and an area of 0.20 mm^2. The timing assumptions inherent to the microengine control structure such as that the branch clear arrives at the microinstruction register array before the global request, and that the microinstruction arrives at the datapath before the global request were, as expected, trivially satisfied by the natural delays of the components involved. No delays needed to be inserted to ensure correct operation. Further discussions regarding timing constraints can be found in [22].

A large part of the microengines power comes from its ability to chain actions with very little control overhead. To achieve good performance it is therefore desirable to have designs which allow long chains to be formed. While this would certainly be one area in which microengines could be used advantageously, we were interested in how well it could do in less than ideal situations. One such situation would be where datapath function units were reused closely in time, effectively hindering long chains by frequently forcing the global request to be reset and a new microinstruction to be fetched. Another situation would be where opportunities to exploit chaining are restricted by the algorithm itself, for example by very tight loops that use only one or two datapath function units per iteration.

We therefore implemented two other designs to get a comparison of such types of designs. One was the differential equation solver presented earlier in this article. This design was chosen since it represents designs in which the datapath function units are reused closely in time, hindering formation of long chains. The hardwired implementation in this case was slightly optimized to increase the concurrency by using shadow registers. As illustrated in Table I however, the microengine still has the advantage in performance. The other design was a barcode reader used in supermarket scanners. This design was chosen since it had quite restricted opportunities for chaining in its most frequently executed code segments, and thus would provide insight to how severe the overhead of a microengine in such designs would be. The lower performance of the microengine barcode reader was expected due to the lack of chaining opportunities. It is encouraging that the performance is still quite close to the hardwired approach even when the opportunity for chaining is very limited.

	Hardwired		Microengine		
Design	Time	Area	Time	$A_{mux/ram}$	Spdup
CD-player	1.58	0.25	1.46	0.20/0.46	+8.0%
Diff-eqn	1.75	0.26	1.69	0.28/0.47	+3.5%
Barcode	3.30	0.10	3.61	0.10/0.25	−9.5%

TABLE I

Design comparisons of microengine vs. hardwired

The area taken up by the ECU, BDU, and RAS blocks is very small and typically takes up only a few percent of the total circuit area. As illustrated in Table I MUX-based ROM structures are quite area efficient and can directly compete with hardwired FSM-based implementations. The area for microengines is considerably larger when using a conventional RAM-based memory structure. This is in part due to the use of automated memory layouts that are far from optimal. Optimized custom memory layout provided by vendors, and manual placement should yield significantly smaller area overhead. Techniques such as code compression and bit-sharing may potentially be used to further reduce the size of the memory but may introduce delay overhead or restrict reprogrammability. Chaining actions also gives additional time for the microinstruction prefetch to complete, potentially allowing use of slower, more area efficient memory.

Our experience with these designs have indicated that microengines are quite efficient in realizing iterative algorithms, but also that many types of designs seem to lend themselves to exploitation of chained execution which is a desirable feature in obtaining efficient microengines. In the context of what automated synthesis tools can achieve, also considering the control structure was implemented with standard gates, these results about the microengines performance are encouraging. It should be noted that both type of designs were implemented without using any explicit timing based optimizations. Better results are to be expected for both types of designs when timing optimizations are applied to hide control overhead. The designs were synthesized to a gate-level representation and bundled data delays were obtained via three-point best/typical/worst case gate-level timing analysis using Synopsys Design Analyzer™ tool. This timing analysis is to our experience very accurate allowing use of relatively small safety margins. Post-layout area numbers and SPICE models were obtained using the Cascade Epoch layout tool.

VI. Conclusions

An asynchronous microengine architecture for application-specific programmable control has been presented. We believe that for many types of designs, this structure can provide performance close to that of designs with hardwired control while still offering the flexibility and ease of design that programmable control and a modular datapath provides. A powerful feature of the proposed architecture is the per-microinstruction programmability of its datapath topology into clusters of independently executing serial chains. These chains can also be decoupled from the microengine execution to allow high latency computations without blocking other parts of the microengine. This programmable datapath topology allows a richer set of schedulings, resulting in compact microcode and high performance. Other methods used to achieve high performance are microinstruction prefetch, and hiding acknowledge synchronization in data propagation through muxes. Using an always acknowledge approach that returns all control signals to the same state facilitates efficient control structures for both two- and four-phase implementations.

Although this work on asynchronous microengines is still in its early stages, the design comparisons presented in this article suggests that asynchronous microengines can yield competitive implementations compared to hardwired approaches for at least some classes of designs, despite using programmable control. Many types of designs seem to lend themselves to the implementation structure of a microengine, and we believe a fair amount of effort and optimizations, not easily obtained in automated high level synthesis, must be made to achieve faster implementations using hardwired approaches. We are currently working on generating more examples to facilitate a comparison on a broader base of designs. We intend to automate the microengine synthesis procedure, and incorporate it in the ACK synthesis framework allowing descriptions entered to be realized as both hardwired and microengine implementations. As the possibility of using test-oriented microinstructions seems highly attractive, we also intend to investigate the testability of microengines.

References

[1] Bill Coates, Al Davis, and Ken Stevens, "The Post Office experience: Designing a large asynchronous chip," *Integration, the VLSI journal*, vol. 15, no. 3, pp. 341–366, Oct. 1993.

[2] Alan Marshall, Bill Coates, and Polly Siegel, "Designing an asynchronous communications chip," *IEEE Design & Test of Computers*, vol. 11, no. 2, pp. 8–21, 1994, Summer.

[3] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, Frits Schalij, and Rik van de Wiel, "A single-rail re-implementation of a DCC error detector using a generic standard-cell library," in *Asynchronous Design Methodologies*. May 1995, pp. 72–79, IEEE Computer Society Press.

[4] Andreas Nowatzyk, Gunes Aybay, and Fong Pong, "Design of the s3mp processor," in *Proc. of Europar Workshop*, 1995.

[5] J. Kuskin and D. Ofelt et al., "The Stanford FLASH multiprocessor," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, May 1994, pp. 302–313.

[6] Maurice Wilkes, "The best way to design an automatic calculating machine," in *Manchester University Computer Inaugural Conference*, July 1951, pp. 16–18.

[7] Kenneth Stevens, "The soft controller: A self-timed microsequencer for distributed parallel architectures," Tech. Rep., Department of Computer Science, University of Utah, Dec. 1984.

[8] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver, "AMULET2e: An asynchronous embedded controller," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. Apr. 1997, pp. 290–299, IEEE Computer Society Press.

[9] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings, "The design of an asynchronous MIPS R3000 microprocessor," in *Advanced Research in VLSI*, Sept. 1997, pp. 164–181.

[10] Akihiro Takamura, Masashi Kuwako, Masashi Ima, Taro Fujii, Motokazu Ozawa, Izumi Fukasaku, Yoichiro Ueno, and Takashi Nanya, "TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model," in *Proc. International Conf. Computer Design (ICCD)*, Oct. 1997, pp. 288–294.

[11] Joep Kessels, Kees van Berkel, Ronan Burgess, Marly Roncken, and Frits Schalij, "An error decoder for the compact disc player as an example of VLSI programming," in *Proc. European Conference on Design Automation (EDAC)*. Mar. 1992, pp. 69–75, IEEE Computer Society Press.

[12] M. Renaudin, P. Vivet, and F. Robin, "ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 22–31.

[13] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu, "A low-power, low-noise configurable self-timed DSP," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 32–42.

[14] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura, "TITAC: Design of a quasi-delay-insensitive microprocessor," *IEEE Design & Test of Computers*, vol. 11, no. 2, pp. 50–63, 1994.

[15] Scott Hauck, Gaetano Borriello, and Carl Ebeling, "Triptych: An FPGA architecture with integrated logic and routing," in *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pp. 26–43. MIT Press, 1992.

[16] Daniel Gajski, *Principles of Digital Design*, Prentice Hall, 1997.

[17] Kenneth Yun, Peter Beerel, Vida Vakilotojar, Ayoob Dooply, and Julio Arceo, "The design and verification of a high-performance low-control-overhead asynchronous differential equation solver," in *Proceedings of the 1997 International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 1997, pp. 140–153.

[18] Ivan Sutherland, "Micropipelines," *Communications of the ACM*, June 1989, *The 1988 ACM Turing Award Lecture*.

[19] S. M. Nowick, *Automatic synthesis of burst-mode asynchronous controllers*, Ph.D. thesis, Computer Systems Laboratory, Stanford University, 1993.

[20] K. Y. Yun, *Synthesis of asynchronous controllers for heterogeneous systems*, Ph.D. thesis, Stanford University, Aug. 1994.

[21] Hans Jacobson and Ganesh Gopalakrishnan, "Asynchronous microengines for efficient high-level control," in *Advanced Research in VLSI*, Sept. 1997, pp. 201–218.

[22] Hans Jacobson and Ganesh Gopalakrishnan, "Asynchronous microengines for efficient high-level control," Tech. Rep. UUCS-97-007, Department of Computer Science, University of Utah, Salt Lake City, UT, USA, June 1997.

[23] Prabhakar Kudva, *Synthesis of Asynchronous Systems Targeting Finite State Machines*, Ph.D. thesis, Computer Science Department, University of Utah, 1995.

[24] Neil H. Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design*, Addison Wesley, 1992.