Application Specific Asynchronous Microengines for Efficient High-level Control

Hans Jacobson, Ganesh Gopalakrishnan

Technical Report: UUCS-97-007
Department of Computer Science
University of Utah, Salt Lake City, U.S.A.

E-mail: hans@cs.utah.edu, ganesh@cs.utah.edu

Abstract— **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 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 support *efficient* programmable control. This is especially true given that asynchronous (self-timed) circuits are well suited for realizing reactive and control-intensive designs.**

We offer a *practical* solution to programmable asynchronous control in the form of application-specific microprogrammed asynchronous controllers (or *microengines*). The features of our solution include a modular and easily extensible datapath structure, support for two main styles of handshaking (namely two-phase and four-phase), and many efficiency measures based on exploiting concurrency between operations and employing efficient circuit structures. Our results demonstrate that the proposed microengine can yield high performance—in fact performance close to that offered by automated high-level synthesis tools targeting custom hard-wired burstmode machines.

I. Introduction

Sequencing of activities in most VLSI digital circuits is achieved by means of a global clock. Supporting global clocking often comes at very high engineering costs, especially given the trend towards deep submicron VLSI. A well-designed clocking system must, among other things, ensure that the clock cycle time not wasted by the submodules. Ensuring this situation involves considerable engineering effort, given the ever-increasing wire-to-transistor delay ratios. This becomes more of a problem in circuits that are *reactive* and *control-intensive* in nature. Such circuits receive data values from the external world at unpredictable moments and have to perform efficiently a piece of computation for each data value received, where the computations and control decisions may take a data dependent amount of time. Clocking power is also an increasingly important issue, given the packaging and cooling issues that highly dissipative circuits involve. Though advanced clocking techniques in this area such as distributed clocking methods [1], [2] and/or gated clocking [3] offer a solution to these problems, these techniques are not ready yet for widespread incorporation into general application specific integrated circuit (ASIC) design in a manner that is cost-effective and meets the time to market.

Supported in part by NSF MIP-9622587

Asynchronous (self-timed) circuits are quite natural for realizing circuits of a reactive and control-intensive nature. Encouraging results are being obtained by many groups in designing self-timed circuits in this domain, for example in communications components used in multiprocessors [4], hardware to network portable electronic devices [5], and digital signal processing algorithms used in audio-electronics hardware [6]. 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 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 support *efficient* programmable control. This is especially true given that asynchronous (self-timed) circuits are well suited for realizing reactive and control-intensive designs. 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 [7] which uses a microprogram engine, and the FLASH processor [8] which uses a processor-core. Programmable asynchronous circuits have also recently shown advantages in embedded and DSP applications [9], [10].

Many of these programmable approaches are very general purpose in their organization to accommodate both pre- and post-fabrication changes of a broad nature. For example, processor cores can be easily re-programmed, and general-purpose microprogram sequencers can be easily equipped with modified microcode. We demonstrate in this work that *application specific* microprogrammed structures can be easily designed for many classes of circuits, perform at least an order of magnitude better than general-purpose solutions based on processor cores, and even approach the performance of hard-wired control in many cases. The method proposed in this report combines the advantages of programmability and self-timing in an application-specific manner. More specifically, the main contribution of this report is the design and experimental evaluation of a general and structured approach to a fully asynchronous microprogrammed control organization [11], a *microengine*, that targets application specific implementations. The architecture emphasizes simplicity, modularity, and high performance. We will also demonstrate that asynchronous design methods can be used advantageously in the design of microprogrammed control and datapath structures that carry out sequencing on the basis of *completion sensing*, instead of a fixed clock schedule. This makes our solution especially attractive for reactive and control-intensive designs.

This report is organized as follows. After surveying related work and motivating our approach of targeting asynchronous microengines for efficient high level control, we describe our proposed asynchronous microengine architecture in detail, using the simple example of a Differential Equation solver in Section III. Section IV gives a more detailed discussion of the structure and operation of the microengine. Optimizations to enhance the microengines performance are then presented in Section V. Section VI presents system timing constraints that must be met to ensure correct operation. In Section VII, a detailed presentation of performance comparisons between the microengine and state-of-the-art asynchronous hard-wired controllers are presented.

A. Related work

Our approach to programmable control targets implementations where both program store and datapath units are fully customized in capacity and functionality, respectively, while still offering a high degree of programmability. In contrast to microprocessor cores the implementation in our approach is adapted to and optimized for the given design specification, rather than the other way around, for maximum performance and flexibility. While possibly having higher control overhead than hard-wired control, our approach nevertheless allows a higher degree of freedom in how to schedule and sequence actions at a fine-grained level. More specifically, our microengine allows *per-microinstruction programmability of its datapath topology* by arranging its datapath units into series-parallel clusters, for each microinstruction. This feature allows the parallel clusters to run concurrently, while allowing the serial units within a cluster to chain [12], as will be elaborated later. Chaining reduces the number of microinstructions needed to carry out a control task. For example, for the differential equation solver example illustrated in Section III, *four* microinstructions of 24 bits width realize the entire control algorithm. Chaining also reduces the overall overhead of fetching microinstructions, because there are fewer microinstructions to fetch. Chaining, in effect, 'rolls' many microinstructions into one large-grained instruction, thus reducing control overhead since several operations can be performed before a new microinstruction needs to be fetched. Chaining also reduces the relative overhead of completion sensing, because completion is now sensed for larger grains of computation. Chaining in this manner is next to impossible to efficiently support in synchronous microprogrammed controllers because of the difficulty of making sure that all desired chain lengths are integral multiples of the clock period.

Programmable asynchronous structures were investigated around the 1980's [13] in the context of a data-flow computer. However, their organizational style did not support many of the features of microengines, including serial/parallel organization and chaining. It was also not an application-specific customization technique for microprogrammed structures.

Asynchronous microprocessors [14], [15], [16] have lately been a popular target for showing advantages in power consumption and speed. They are not applicable in all embedded control systems however, due to their high fabrication cost, large size, relatively high power consumption, and fixed general purpose instruction set. As an example we implemented a CD player error decoder [17] in our microengine architecture (presented later in this report) 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 [15]. 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. This example serves to illustrate the performance advantage obtainable by special purpose hardware such as our microengine compared to the general purpose hardware of microprocessors.

Other programmable control approaches have recently been investigated [9], [10], [18]. These are best characterized as programmable microprocessor cores. For example, [9] allows a dedicated datapath unit to be added to a microprocessor 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. Since these types of programmable microprocessor cores have fixed control structures and bus widths, they are also not easily adaptable to specific design requirements efficiently.

Another method to obtain programmable control in a self-timed design context is by using FPGAs such as Triptych [19]. However, these and other similar FPGA structures are configuration-time reprogrammable, but not (easily) run-time configurable. In addition, microengines are superior both in terms of area and speed compared to Triptych based structures [19].

II. Architecture 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 thus assumed that the datapath has finished computing and 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. However, 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.

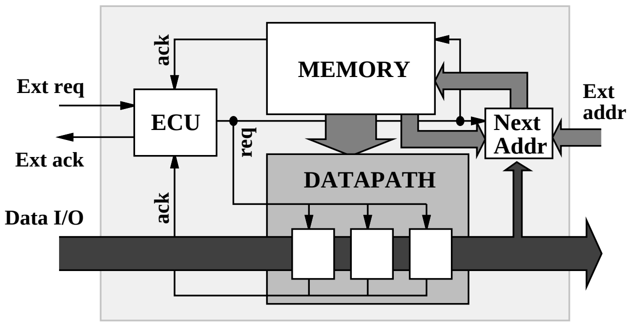


Fig. 1. High Level Structure

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 datapath units to start executing. Each datapath unit then signals its completion by generating an acknowledge. While the current microinstruction is being carried out, the next microinstruction is concurrently fetched predicting branches suitably, as elaborated later. The datapath units must then be explicitly synchronized to ensure they have all completed before the next microinstruction can be propagated to the datapath. This function is performed by the *execution control unit* (ECU in Figure 1). The ECU collects acknowledge signals from all datapath units before generating a request that *propagates* the already waiting next microinstruction to the datapath, thus starting a new execution cycle of the microengine.

Microengine highlights

The microengine achieves its efficiency in a number of ways. Its control and datapath structures are fully customized to the control problem, thus minizing overhead. Its designer has complete control as to the *degree* to which the design should be programmable. A modular datapath also allows easy replacement of datapath functional units, thus facilitating upgrading as well as late-binding of design decisions. Similar changes can, in a synchronous design, obviate the clock schedule, thus requiring total re-designs. The most crucial optimization exploited in the microengine is that of per-microinstruction programmability of its datapath topology, as explained earlier.

The overhead inherent to programmable control structures is further reduced by parallelizing microinstruction prefetch with datapath evaluation, as well as by setting up multiplexors for the next microinstruction concurrently with acknowledge synchronization for the current microinstruction, as will be elaborated later.

III. Microengine operation

The differential equation solver [20] 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* and is used to numerically obtain the values of y satisfying the differential equation $y'' + 3xy' + 3y = 0$ where x ranges from $x(0)$ to a with step size dx. 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. Three threads calculating y, y' (u in figure), and incrementing x are needed per iteration. Computing y' 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.

We decide to allocate one multiplier and one arithmetic unit for the calculation of y', 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.

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 first half of y' while the third calculates x, the loop condition, and the second half of y'. 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).

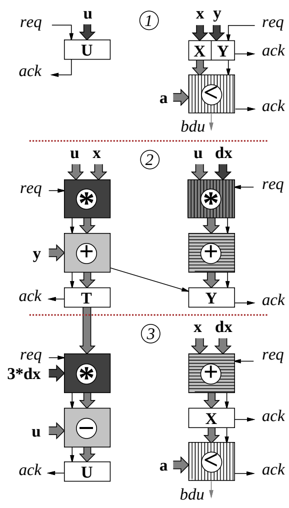
A. Microprogram structure

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 *se* bit may also be used to incorporates the functionality of an *ss* 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.

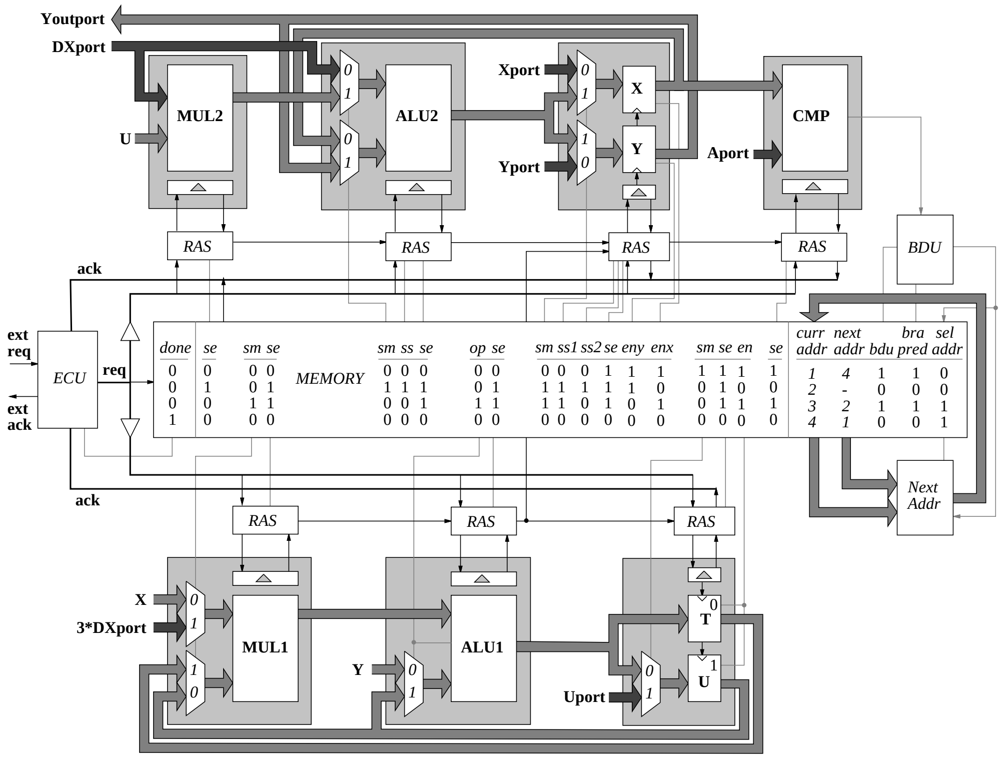
The following bit fields of the microprogram are used to control the global microprogram flow. The current ad-

```
diffeq {
    read(x, y, u, dx, a);
    while (x < a) {
        x1 := x + dx;
        u1 := u - 3 * dx(u * x + y);
        y1 := y + u * dx;
        x := x1; y := y1; u := u1; }
    write(y);
}
```

a) General algorithm



b) Data and Control Flowgraph



c) Microengine Implementation

Fig. 2. Design Example: Differential Equation Solver

dress, *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 when the microinstruction contains a branch operation and specifies the address of the instruction being branched to. The set-branch-detect-unit, *bdu*, bits specifies which conditional expression result 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 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).

B. 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. Since the RAS blocks handles 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 [21] delays for acknowledge generation. 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.

C. Microprogram execution

The following section will step through the execution of the differential equation solver microprogram illustrated in Figure 2(c).

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 instruction 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 execute 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* whereafter 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 from 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 first half of *y'* : $(MUL1 \rightarrow ALU1 \rightarrow TU)$ and the other computes $y : (MUL2 \rightarrow ALU2 \rightarrow XY)$. 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 subthreads 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 second half of *y'* : $(MUL1 \rightarrow ALU1 \rightarrow TU)$ and the other computes x and 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

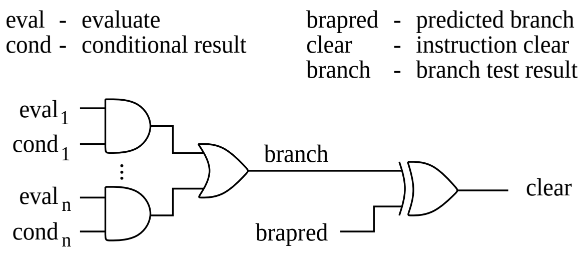

eval - evaluate
cond - conditional result
brapred - predicted branch
clear - instruction clear
branch - branch test result
eval 1
cond 1
eval n
cond n
branch
brapred
clear

Fig. 3. Branch Detection Unit

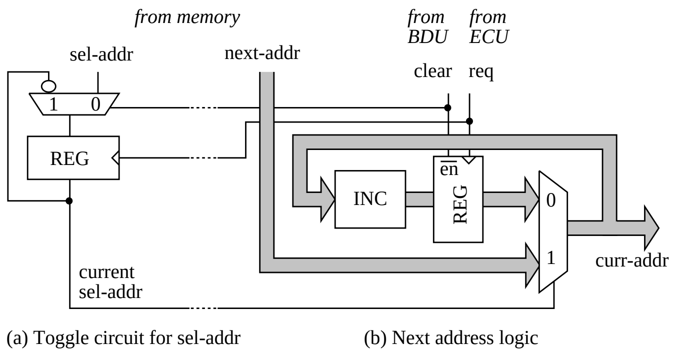


Fig. 4. Next Address Unit

address 4 is propagated to memory. Instruction 4 contains an unconditional jump to instruction 1 and also indicates 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 next address generation, global and local execution control, datapath unit structure, and architecture optimizations.

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 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.

To detect if a branch was correctly predicted, the *Branch Detection Unit* (BDU) communicates the state of the datapath back to the next address logic at the end of the cycle. The structure of the BDU is shown in Figure 3 and can be functionally divided into two parts.

The first part evaluates if the branch condition is true or false. A set of *eval* signals from memory are used to select which conditional results from the datapath, *cond*, to test. This functionality is achieved by a simple AND-OR structure. Note that this branch test structure also allows ORing tests of several conditional results.

The second part compares the branch result with the predicted branch and asserts a clear signal if they differ, i.e. if the prediction was wrong. This clear signal has three different functions. Its first function is to toggle the *sel-addr* bit from memory so that the correct address is propagated to memory at the next global request. The toggle circuit, which is part of the microinstruction register array, for *sel-addr* is illustrated in Figure 4(a). Second, since the propagation of the global request to the datapath is never disabled, the *se* and *ss* signals of the previously executed instruction must be cleared in order to stop the RAS blocks from propagating the request to the datapath units which would otherwise repeat the execution of that instruction. This is achieved by synchronously clearing these bits on the next arriving global request. Other registers are simply disabled from latching new data. The *eval* and *bra-pred* bits are also cleared so as to not toggle the *sel-addr* bit again after fetching the correct microinstruction. Third, the clear signal is also used to disable the next address block, illustrated in Figure 4(b), from changing the internal values of the addresses so that the old incremented address is propagated to the memory correctly.

Note that 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.

Thus if a branch is mispredicted, the *sel-addr* bit value is toggled to propagate the correct address to memory, all *se*, *ss*, *eval* and *bra-pred* bits are cleared, and the next address block is disabled from latching a new incremented address when the next global request arrives. A correctly predicted branch thus has zero overhead while a misprediction requires an extra cycle to fetch the correct microinstruction.

B. Microengine execution control

There are many ways of realizing a structure for request-acknowledge handshaking between the microengine and the datapath units. Since all datapath units must synchronize with the memory before a new microinstruction can be latched, there is little to gain by generating separate request signals to individual datapath units. An approach of having only one global request signal that decides when to fetch a new microinstruction from memory as well as cause the datapath units to start executing is therefore used. This approach reduces the complexity of the request control logic necessary, as well as simplifies parallel datapath unit operation and timing analysis. Our 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, burstmode [4], [22], [23] type

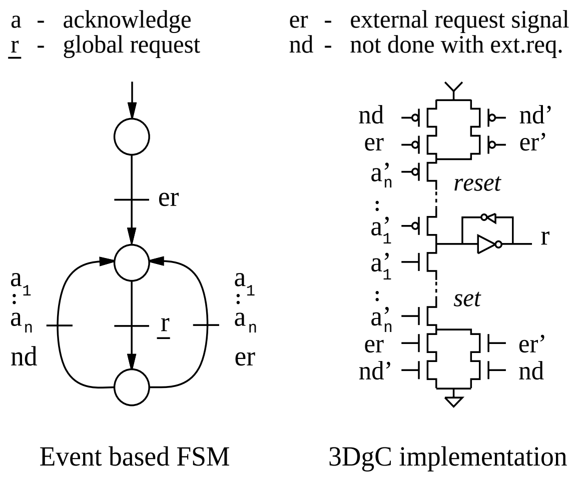


Fig. 5. Execution Control Unit

of asynchronous state machines are used. The operation of a burstmode state machine allows the acknowledge signals from the datapath units that are generated in response to the global request 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 with 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 for the out of phase acknowledge approach. In addition, the same request generation logic can be used for both two and four phase protocols.

An abstract event based FSM for the global request generation and resulting complex gate implementation using the 3D synthesis tool [23] is illustrated in Figure 5. The respective n and p transistor networks can be decomposed into balanced tree structures of gates to simplify timing analysis, or unbalanced ones to improve performance. This request generation logic then forms part of what is called the *Execution Control Unit* (ECU) used to generate a new event on the global request signal.

In our ECU realization, it is assumed that the same protocol is used for communication internal to the microengine as well as with the environment. The ECU is initially quiescent. After receiving a request from the environment an event on the global request signal is generated causing the microengine to start executing. This global request latches the next address and the new microinstruction from memory and triggers the datapath units to execute. For both the two and four phase case, the *not done* signal in Figure 5 is generated by a SELECT-element (not shown) connected to the *done* level signal from memory and the global request signal. While *done* is false, the SELECT-element generates events on the *not done* signal. When *done* is true, an event is instead sent to the environment as an acknowledge that the microengine has completed the requested computation. The ECU then remains quiescent until a new request arrives from the environment.

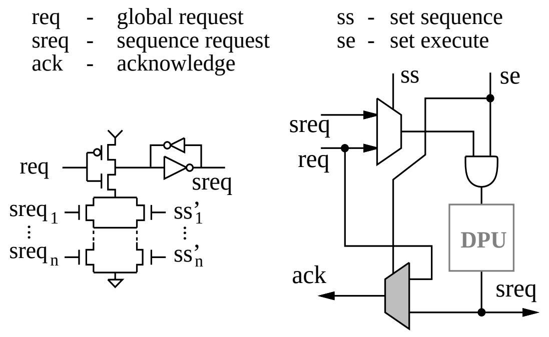
C. Local datapath execution control

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.

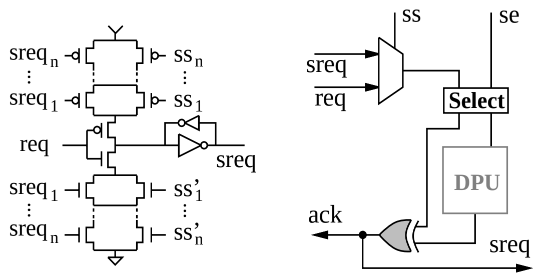
Sequence control. The sequence control function of the RAS can in its simplest form be performed by a MUX, controlled by the set-sequence bit, that propagates either the global request or a sequential request to its datapath unit. The output of the sequence control MUX is hazard free since both the global and sequence request signals will reach stable values before the next microinstruction may alter the MUX control signal (signal *ss* in Figure 6).

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 6(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.

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 per-



(a) RAS block for 4 phase protocol



(b) RAS block for 2 phase protocol

Fig. 6. RAS block structures

formance 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.

Request-acknowledge control. Besides sequencing control, the RAS must also provide means to correctly perform 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. An 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 6(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. An 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.

D. Datapath unit structure

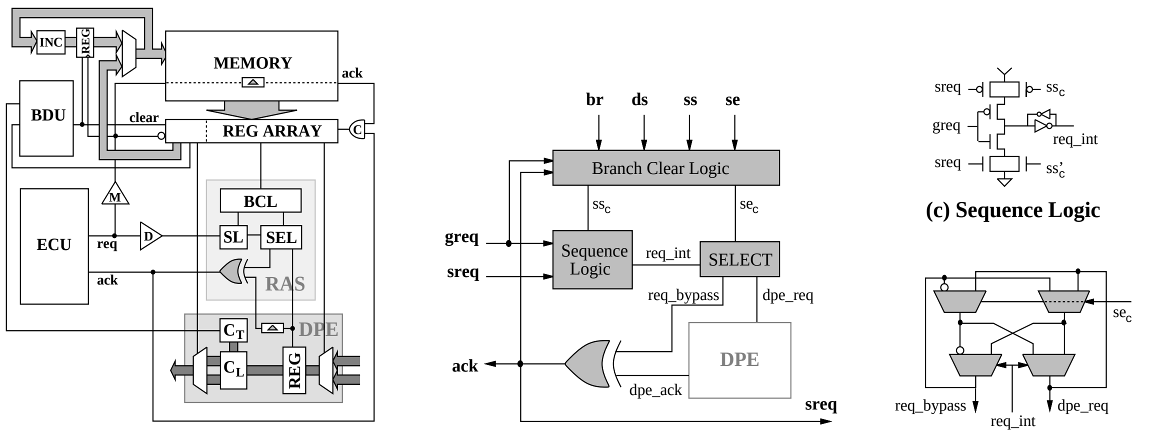
Each datapath unit is assumed to be a self-timed element using single rail bundled data in communication with its environment. The request-acknowledge handshaking, completion detection, and data representation internal to a datapath unit however, can be implemented in an arbitrary fashion. For example, some datapath units can be implemented using simple standard gates with matching delays while others can use sophisticated completion sensing such as complex gate domino-logic. A datapath unit may also form complex structures such as a selftimed loop or even a hierarchy of microengines. Assumptions about safe data latching in the face of eventual datapath dependencies, e.g. should cross-thread synchronization be used or not, while performing scheduling is left to the designer to decide based on knowledge about datapath timings. If the designer choose to apply timing assumptions regarding concurrent propagation of data signals through input MUXes while the ECU performs completion synchronization and the request propagates through the RAS block it is also left to the designer to verify these assumptions.

V. Architecture optimizations

The structure presented for the microengine control so far brings forth the high level concepts of the microengine architecture in a clear fashion. However, it is not very 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 following sections will discuss operational and architectural optimizations that can reduce the control and data computation overhead considerably

A. 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



(a) Microengine Structure **(b) RAS Block Structure** **(c) Sequence Logic** **(d) SELECT**

Fig. 7. Optimized Two Phase Structure and 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 request 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.

Our goal with the optimized control approach then is to reduce the control overhead by allowing the microinstruction to propagate to the datapath and allow data propagation through MUXes, concurrently with the ECU performing completion synchronization. The following sections will present optimized approaches for the two phase and four phase protocol implementations respectively. For the two phase case, a solution where each datapath element latches its own part of the new microinstruction upon completion of its current task is presented. For the four phase case, a simpler solution where the new microinstruction is latched during the passive phase of the handshake is presented.

A.1 Optimization for two phase

In this section we will present a solution for the two phase protocol where each datapath element latches its own part of the new microinstruction upon completion of its current task. Necessary changes in the RAS to ensure a hazard-free behavior under the new signal arrival order will also be discussed. An overview of this optimized architecture is illustrated in Figure 7(a).

Latching the next microinstruction. Using an approach where each datapath element latches its own part of the new microinstruction upon completion of its current task allows propagation of new control and data signals to take place concurrently with the evaluation of the execution control unit. The acknowledge signal local to each RAS could then be used as a request signal to latch the corresponding part of the next instruction. Since datapath elements may execute in sequence however, data dependencies may exist between such stages. Early latching of the new instruction must therefore be restricted to control signals that do not alter the data output values of a datapath element. Other control signals such as set-mux signals for output MUXes must not be latched until all datapath elements have completed their scheduled actions. These signals can then be latched using the global request signal since they in general have sufficient time to propagate to their respective components inside the datapath units before new data arrives.

Since this approach may cause a datapath element to request latching of a new instruction before the fetch from memory has completed, synchronization logic for the RAS and memory acknowledges must be provided. Since the method of always generating an acknowledge keeps these signals in phase, it is possible to realize this synchronization with a simple C-element.

RAS block optimizations. Allowing new control signals to arrive before all acknowledge signals have reached the same phase again requires somewhat different logic implementations of the RAS to avoid hazards. If a simple MUX was used as the sequence logic part of the RAS it could exhibit glitches if the set sequence signal of the next microinstruction was allowed to arrive before the sequence request signals had attained the same state (phase) as the global request. The RAS logic therefore must be made insensitive to such early changes of the set sequence control signal. The implementation of such a circuit is illustrated in Figure 7(c). In this realization, the set sequence and sequence request signals, *ss* and *sreq*, are allowed to arrive in arbitrary order. These signals may only cause the branches of the currently conducting transistor network (say P transistors) to go on or off. The opposite transistor network (N transistors) however, will remain non-conducting until the next event on the global request arrives. The output is thus hazard free and kept at its current logic level by a sustainer in the form of cross-coupled inverters. The output of the programmable sequence combination logic in Figure 6(d) is then connected to the *sreq* inputs of the sequence logic in Figure 7(c). Note that due to their similar structure, these two logic blocks can be merged into a single complex gate.

The original approach of latching the new instruction word relied on a synchronous clearing of the microinstruction register array. Subsequently it also required the branch to be resolved before latching a new microinstruction. Since the new approach means the new microinstruction might be latched *before* the branch has been resolved, other means of clearing the instruction before the next request arrives to the datapath must be provided. This function is implemented by introducing asynchronous branch clear logic local to each DPE. The structure of the RAS block under the assumption of early instruction propagation is illustrated in Figure 7.

A.2 Optimization for four phase

In this section we will present a solution for the four phase protocol where the new microinstruction is latched during the passive phase of the handshake. While the method presented for two phase could be used, using this alternate approach enables further optimizations of the RAS block for fast request propagation and also removes the restriction on latching control signals that may alter the data outputs separately. While precharging and data propagation through transparent latches can be done, we assume that no computations dependent on data inputs to a datapath unit are performed during the passive phase. An overview of this optimized architecture is illustrated in Figure 8(a).

Latching the next microinstruction. Latching the new microinstruction during the passive phase of the handshake allow propagation of new control and data signals to take place concurrently with eventual precharge of datapath units and the return to zero evaluation of the execution control unit. Since no data dependent computations are performed during the passive phase, the whole microinstruction, including control signals that may change data outputs, can be latched at once using the falling edge of the global request signal. Using this approach a synchronous clear signal derived from the branch result and predicted branch signals, as in the original solution, can still be used.

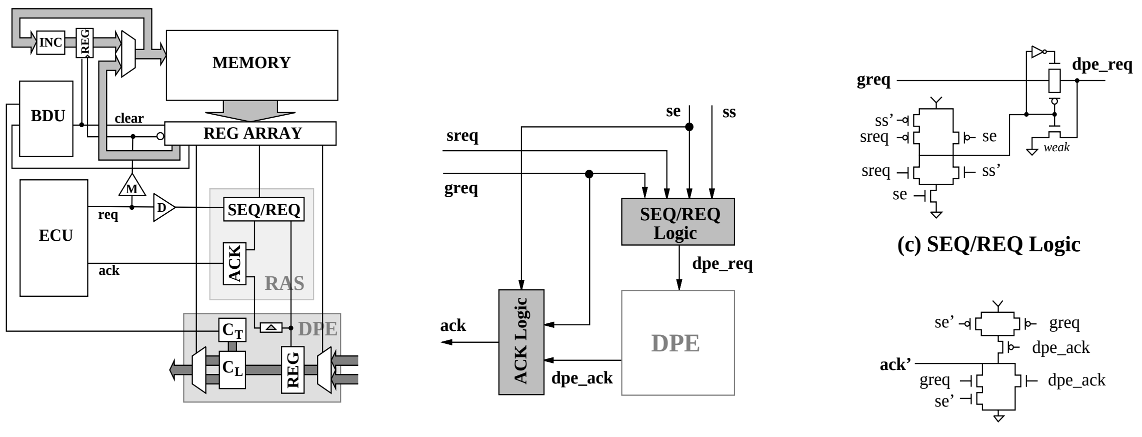
RAS block optimizations. When using the four phase protocol, further optimizations can be made to the RAS logic if the falling edge of the request signal is used to latch the new microinstruction during the passive phase of the handshake. The solution illustrated in Figure 8 reduces the propagation delay of the global request through the RAS to that of a single transmission gate, while still providing a lower delay for sequence requests than that of the original approach. As with two phase, the output of the programmable sequence combination logic in Figure 6(b) is connected to the *sreq* inputs of the SEQ/REQ logic in Figure 8(c).

In this solution, the global request is always used as the signal to be propagated. Since the microinstruction signals controlling execution and sequencing, *se* and *ss* are latched during the passive phase, the transmission gate will already be setup to its current mode of operation by the time the rising edge of the global request arrives. If set to execute in parallel mode, the global request is thus directly propagated to the datapath element, yielding only the delay of passing through an already conducting transmission gate. If set to execute in sequential mode the transmission gate will be closed, disabling the global request from propagating, until the arriving sequence request causes it to open.

An important feature when using the four phase protocol, is the ability to generate a parallel return to zero, regardless of the actual mode of operation of the individual datapath elements. This is possible since no useful computation is performed, and hence no data-dependencies exist, during the passive phase of the handshake. Since the transmission gate is guaranteed to remain open at least until the next microinstruction has been fetched, the falling edge of the global request will always pass through the transmission gate (if setup to execute). This generates a fast parallel return to zero of all datapath elements even for datapath elements setup to execute in sequence. Since the propagation of the global request is concurrent with the latching of the new microinstruction, one restriction is placed on signal arrival order to this RAS realization. The global request must always arrive to the RAS block before any new control signals of the next microinstruction. Otherwise a change in the *se* and *ss* control signals might cause a glitch on the propagated request signal. This restriction is trivially satisfied since the number of datapath elements will always be less than or equal to the number of registers in the register array, requiring less buffering, and also since the instruction signals must propagate through registers before arriving to the datapath.

B. 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. This section presents how ECU and RAS blocks must be altered to support decoupling of arbitrary clusters of datapath units for the four



(a) Microengine Structure **(b) RAS Block Structure** **(c) SEQ/REQ Logic** **(d) ACK Logic**

Fig. 8. Optimized Four Phase Structure and RAS Block

phase protocol.

ECU alterations. In order to allow a datapath unit to decouple itself from the microengine execution, that is making itself independent of the execution of other parts of the microengine, the always acknowledge scheme must be abandoned. The reason for this is that the decoupled datapath units are setup to execute and therefore cannot generate an acknowledge until they have completed their respective computations. With an always acknowledge scheme this would lock up the execution of the rest of the microengine until all acknowledges, including those from the decoupled units, have been generated.

We must devise a method that makes the ECU insensitive to the acknowledge generations of decoupled datapath units until it wants to resynchronize with them again. While this behavior cannot be realized efficiently by burst-mode FSMs, a hand made complex gate ECU circuit can be made quite efficient. One approach to realize the desired behavior of making the ECU insensitive to acknowledges from certain datapath units is to provide a bypass transistor that conducts, much in the style of the sequence RAS logic presented earlier in Figure 6(b), whenever the corresponding datapath unit is not setup to execute. By providing such a bypass transistor path controlled by the set-execute signals of the datapath units the ECU can be programmed to ignore acknowledges from datapath units not setup to execute. Note that this approach also alleviates the problem of having the RAS block provide a bypass path for the acknowledge, reducing its complexity and delay. Each datapath unit that can be used in decoupled mode also has an extra bypass transistor in both transistor stacks.

Figure 9(a) illustrates the new structure of the ECU that supports both "out of phase" acknowledges and decoupled execution. When a datapath unit is setup to execute, the n-stack transistor connected to the set-execute signal from memory is not conducting and the ECU is forced to wait for the corresponding datapath units acknowledge. If a datapath unit is not setup to execute, the transistor instead provides a bypass path, enabling the ECU to continue without receiving an acknowledge from the corresponding datapath unit. Only the n-stack needs the set-execute bypass transistors since the acknowledge of a datapath unit not setup to execute will remain low, automatically providing a bypass path for the p-stack. The observant reader might have noticed that if no datapath unit is setup to execute, the n and p-stacks in the ECU would short-circuit. This can never happen however, since the memory is always setup to fetch new instructions and thus does not have a bypass transistor on its acknowledge path through the ECU.

If a datapath unit is setup to execute in decoupled mode, the transistors connected to the set-decoupled, sd, signal provide a bypass path effectively allowing the ECU to ignore the acknowledge from the decoupled datapath unit until it wishes to resynchronize with the decoupled datapath unit by setting sd low. If the decoupled datapath unit finishes and generates an acknowledge before the microengine wants to resynchronize with it, the acknowledge is simply ignored until the microengine is ready to resynchronize and sets the sd bit low. If the datapath unit has not finished its computation by the time sd is set low, the ECU will simply wait until the computation has finished and the corresponding acknowledge generated. This resynchonization takes place between two completely asynchronous entities, the microengine and the datapath unit. However, since the ECU always initiates the resynchronization and then waits for the datapath units acknowledge to arrive, there is never any race present between the sd and acknowledge signals, and metastability or glitches cannot occur.

While all se and ss signals from memory are latched on the negative global request edge, sd must be latched on the positive edge. Otherwise the sd signal could be set low, i.e. telling the ECU to wait for a rising edge on the acknowledge from the decoupled datapath unit, while the ECU in fact is waiting for falling edges on the other acknowledges. The p-stack will thus never conduct if the decoupled datapath units acknowledge has aldready gone high, and the ECU will deadlock waiting for a falling acknowledge that will never occur.

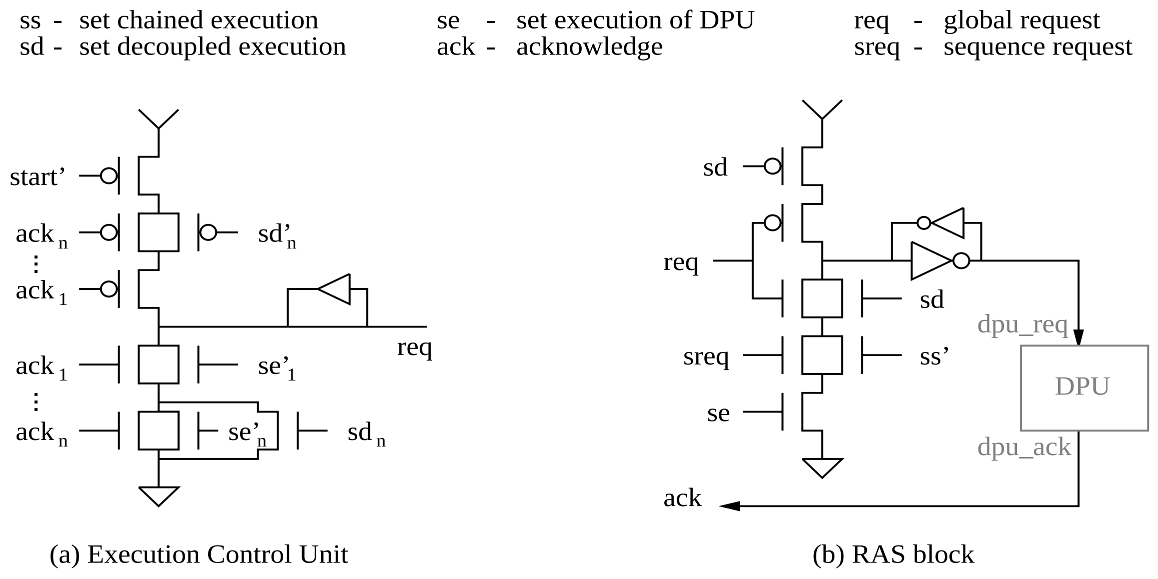


Fig. 9. ECU and RAS supporting decoupled execution

RAS block modifications. As mentioned, with the ECU supporting “out of phase” acknowledging, any datapath unit not executing during a cycle should not generate an acknowledge. The extra logic previously required for bypass acknowledge generation by the RAS is therefore no longer needed as illustrated in Figure 9(b).

Since the computation of a decoupled datapath unit may span over several microengine cycles the RAS block must be made insensitive to further events on the global and sequential request signals. This is achieved by using the set-decoupled, *sd*, signal to block further events from propagating through to the datapath unit. This means that once the rising edge of the request has been propagated through the RAS block, the following falling edge must not be propagated (until the ECU initiates the resynchronization that is). This is achieved by a transistor connected to the *sd* bit that cuts off the p-stack of the RAS logic illustrated in Figure 9(b) throughout the decoupled computation. To support decoupling of an entire chain of datapath units an extra bypass transistor connected to the *sd* bit in the n-stack is needed to allow sequential requests to propagate regardless of the state of the global request. This bypass is necessary since the microengine might be in the middle of executing another microinstruction and the global request be in an unknown state at the time the sequence request from one decoupled datapath unit propagates to another. The transistors connected to the *se*, *ss*, and *sreq* fullfill their usual functionality.

Since the *sd* bit must be latched on the positive edge of the global request, as discussed earlier, a timing restriction must be imposed on the arrival order of the global request and *sd* signals. The global request must always arrive at the RAS block before any change in the *sd* bit. Otherwise a request may be generated to the datapath unit before the global request arrives to the RAS if the *se* and *sd* bits are set and the *ss* bit is not set. If a decoupled chain is executed we may also run into the problem of the microengine initiating the resynchronization with the chain before the chain has completed. That is, the sequence request has not propagated to, for example, the last datapath unit in the chain. Setting *sd* low at such a time would mean cutting off the transistor allowing the sequence request to propagate through the RAS regardless of the status of the global requests. However, since we imposed the restriction that the global request must arrive before any change in the *sd* bit, the n-transistor connected to the global request will conduct and allow the sequence request to propagate through the RAS block. Since the ECU is then blocked waiting for the resynchronized chains acknowledges no further events will be generated on the global request until all sequence requests, and their subsequent acknowledges, of the chain have finished propagating. The imposed arrival order of the global request and *sd* signals is trivially satisfied since the global request buffer tree to latch the microinstruction is longer than the buffer tree to the datapath, and since the *sd* bit must also propagate through a register before arriving to its RAS block.

VI. System Timing

The following sections will discuss the most important timing constraints that must be satisfied for correct operation of the microengine. Timing inequalities that illustrates these timing constraints will also be presented. Inequalities for hiding e.g. input MUX delays in the concurrent evaluation of the ECU and propagation of the global or sequential requests through RAS blocks are not presented but can easily be derived from the given inequalities. Unless wire delays are explicitly mentioned, they are assumed to be negligent.

The following conventions and abbreviations are used in the timing inequalities. If no subscript indicates otherwise, the signal propagation through the component in question is referred to. The term *buf* stands for delay through buffering of a multiple fanout signal, *BDU* stands for branch detection unit, *ECU* for execution control unit, *RAS* for request/acknowledge/sequence block, *DPU* for datapath unit, *clr* for clear, *SL* for the sequence logic and *BCL* for the branch clear part of the RAS in case of two phase, *C* for C-element, *SEL* for select element, *clear* for the branch clear signal, *req* and *sreq* for global and sequence request signals, *rtz* for return to zero, *DP*, *ADR*, and *MI* for datapath, next address, and microinstruction respectively, and *REG* for register.

A. Two phase

Branch prediction. Since no execution should take place if the branch prediction was incorrect, the branch clear signal must arrive in time to set the select element to propagate the event to the right output. Because the select element requires no setup time, this timing property is satisfied if the following inequality holds.

(1) $ECU + DP\_req_{buf} + SL > BDU + clear_{buf} + BCL$

Where the delays of SL and BCL, and also DP_req_{buf} and $clear_{buf}$ are comparable, reducing the constraint in all practical aspects to only require the delay through the *ECU* to be greater than through the *BDU*. This timing constraint is trivially satisfied in most designs since the number of acknowledges to the ECU tends to be larger than the number of conditional inputs to the BDU. If the timing constraint is not met, a delay must be inserted on the global request.
The following inequality ensures latching of correct next address value, i.e. that BDU propagation delay and next address register setup times are met before the global request arrives at the next address register.

(2) $ECU + ADR\_req_{buf} > BDU + ADR\_REG_{enable}$

This timing inequality is also trivially satisfied in most designs since the delay of the ECU tends to be larger than the BDU, and the next address request buffer delay, which is the same as the buffer delay to the microinstruction register array, is larger than the enabling/disabling time of the next address registers.

Data latching. Inequality 3 ensures that the new microinstruction has time to propagate to the datapath before the next global request arrives to the datapath. This inequality is trivially satisfied for most designs. If not, a delay needs to be inserted on the global request.

(3) $ECU + DP\_req_{buf} > C + MI\_REG$

If an assumption that data values are latched correctly by datapath units operating in parallel mode when data dependencies are present is made, it is left to the designer to verify the correctness of the assumption. The designer would then have to make sure the following timings are satisfied.

(4) $DP\_req_{skew_{ij}} + DPU\_REG_{hold_i} < DPU\_REG_j + MUX_i$

Inequality 4 ensures, assuming delay internal to a datapath unit is unknown (zero), no new values can propagate from the outputs to the inputs of registers latching data in parallel. dp_req_{skew} accounts for skew related to difference in request arrival time at each RAS as well as to each DPU's input registers due to wire delay and signal buffering.

The assumption that data values arrive to a datapath unit operating in sequential mode before the sequential request is trivially satisfied since it only means that the data wire delays are smaller than the propagation delay of the sequence request.

(5) $XOR_i + sreq_{wiredelay} + RAS_{sreq_j} > Data_{wiredelay}$

B. Four phase

Branch prediction. The following inequality ensures that the synchronous branch clear signal arrives before the next request to the microinstruction register array.

(6) $ECU + MI\_req_{buf} > BDU + clear_{buf} + MI\_REG_{clr}$

This constraint is usually satisfied depending on the difference in number of acknowledges to the ECU and conditional signals to the BDU. If the timing constraint is not satisfied, a delay must be inserted on the global request.
As in the two phase case, the following inequality ensures latching of correct next address value, i.e. that BDU propagation delay and next address register enable times are met before the global request arrives at the next address register.

(7) $ECU + ADR\_req_{buf} > BDU + ADR\_REG_{enable}$

As in the two phase case this timing inequality is also trivially satisfied in most designs.

Data latching. Inequality 8 ensures that the new microinstruction has time to propagate to the datapath before the next global request arrives to the datapath. This inequality is trivially satisfied for most designs. If not, a delay need to be inserted on the global request. Note that the microinstruction is latched on the falling edge of the global request.

(8) $2 * DP\_req_{buf} + RAS_{rtz} + DPU_{rtz} + ECU + RAS\_req >$
$MI\_req_{buf} + MI\_REG$

As in the two phase case, if an assumption that data values are latched correctly by datapath units operating in parallel mode when data dependencies are present is made, it is left to the designer to verify the correctness of the assumption. The designer would then have to make sure inequality (4) is satisfied.
As in two phase, the assumption that data values arrive to a datapath unit operating in sequential mode before the sequential request is trivially satisfied since it only means that the data wire delays are smaller than the propagation delay of the sequence request.

(9) $sreq_{wiredelay} + RAS_{sreq} > Data_{wiredelay}$

There are no extra inequalities needed to describe the operation of decoupled execution of datapath units, as their operation is the same as for non-decoupled datapath units, the only difference being that the ECU does not wait for their acknowledges.

VII. Design Example: CD-player Error Decoder

To estimate the efficiency of the presented microengine implementation style compared to a custom control implementation using the same datapath structure, a CD-Player error decoder [17] was built as a design example. In addition to the microengine style, the decoder was therefore

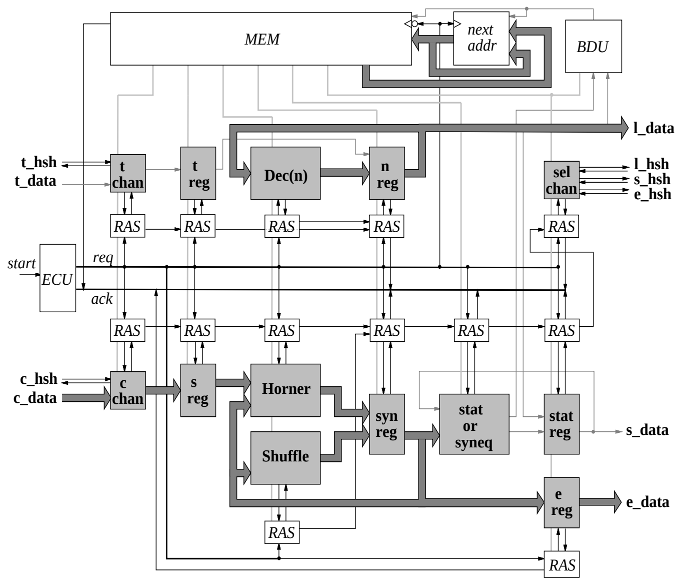


Fig. 10. CD-Player error decoder structure

```
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[7:0])),
     syno[23:16]   := GFadd(si, Alpha(Alpha(syni[7:0]))),
     syno[31:24]   := GFadd(si, Alpha(Alpha(Alpha(syni[7:0]))))   }

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]) :=
     (syno[15:8],syno[31:24],syno[7:0],syno[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  ->  <SYNDROME> ;
              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. 11. CD-Player error decoder HOP specification

also implemented using our high level synthesis framework for asynchronous circuits, ACK [24]. This framework takes a high level description in either the HOP language [24] (illustrated in figure 11) or Verilog−+, a synthesizable subset of Verilog extended to handle channels, as input and targets customized interacting burstmode 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 [17] 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 correspond to what would easily be achievable using an automated synthesis tool.

The error decoder circuit implements error-detection on the audio information recorded on Compact Discs using a syndrome computation algorithm. Figure 10 illustrates the structure of the microengine implementation and Figure 11 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 [17].

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 DPU's 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 DPU's 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, many RAS acknowledges can therefore be removed (7 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

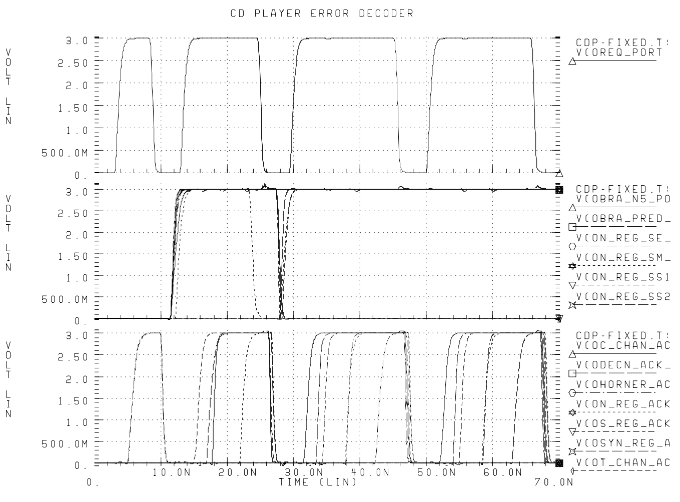


Fig. 12. CD-Player error decoder SPICE waveforms

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 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. 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 until n is negative or the two low end bytes of *syn* differ, in which case an error has been found. 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 12 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 11 and tests the *SYNDROME* loop condition. The use of chained execution can be observed by the sequentially occuring acknowledges in Figure 12'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 12. 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.

A. Results

The Tangram implementation described in [17], which was targeted for low-power, used dual rail logic and a 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 [6] a factor of 1.5 in performance improvement and a 40% smaller area can be attributed to single rail over double rail in an implementation of a similar, but more complex, error decoder for the DCC player. A single rail Tangram implementation of the CD player decoder could therefore be expected to have a cycle time of about 13 microseconds and an area of 1.2 mm^2.

Using our design tool ACK to automatically generate a customized hard-wired implementation targeting a 1.2 micron CMOS technology, the corresponding worst case cycle time was in the order of 3.8 microseconds using a four phase handshake protocol and with an area of 1.3 mm^2. Using the same datapath, the microengine implementation had a resulting cycle time of about 4.0 microseconds also using a four phase protocol and an area of 1.7 mm^2.

We also implemented the CD player error decoder in a 0.6 micron technology [1] and performed post-layout SPICE simulation of the same. The cycle time under worst case transistor models and temperature was 1.46 microseconds. Feature size scaling under constant field assumption [25], except for voltage, would result in gate delays in a 1.2 micron 5V technology being approximately 2.5 times that of a 0.6 micron 3V technology while wire delays stay the same. The corresponding cycle time for the microengine implementation of the decoder should therefore be in the order of 3.7 microseconds in a 1.2 micron technology. The gate level delay analysis would thus be inside a 10% error margin of post-layout SPICE simulation. We assume that the post-layout hard-wired implementation has similarly accurate cycle time. The timing assumptions inherent to the microengine control structure such as that the branch clear arrives to the microinstruction register array before the global request, and that the microinstruction arrives at the

[1]Our updated VLSI tools no longer feature a 1.2 micron library

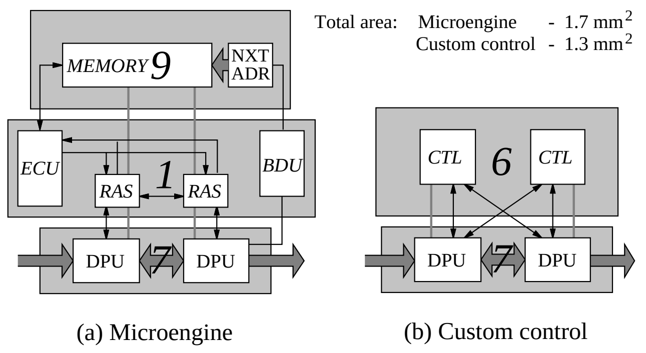


Fig. 13. CD-Player error decoder area breakdown. The large digits add up to the total area of the respective designs.

datapath before the global request were, as we expected, trivially satisfied by the natural delays of the components involved. No delays needed to be inserted to ensure correct operation.

The area breakdown for the customized and microengine implementations is illustrated in Figure 13. As illustrated, the area for the microengine control units is very small, and the major part of the area is spent on memory. The ability of our microengine to chain actions is therefore important not only for performance but also for saving area since it reduces the number of microinstructions needed. The CD player error decoder for example requires only 9 30-bit wide microinstructions. Techniques such as code compression and bit-sharing may also be used to 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. The area for the burstmode controllers is surprisingly large, and serves to illustrate how hard it is to estimate the implementation complexity of finite state machine controllers, even for moderately sized designs.

The designs were synthesized to a gate level representation and performance measured via timing analysis using worst case gate delay and wire load models in Synopsys^{TM} Design Analyzer tool. Bundled data delays were obtained via three-point best/typical/worst case gate level timing analysis using this tool and 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^{TM} Epoch layout tool. It should be noted that both designs were implemented without using any explicit timing based optimizations. Better results are to be expected for both designs when timing optimizations are applied to hide control overhead. 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. One of the reasons the microengine approach is able to perform so well compared to the custom control approach is due to the ability to naturally and efficiently chain the actions of an arbitrary number of datapath units while still being able to perform a parallel return to zero. An initial performance concern about the microengine control structure was the presumed high capacitance load on the global request wire. The resulting implementations however, showed that while the global request in the microengine had a capacitance of 1.3nF, some acknowledge wires to the burstmode controllers in the customized control implementation that were on the critical path actually had even higher loads, the worst being 2nF.

VIII. Conclusions

An asynchronous microengine architecture for programmable control has been presented. We believe that for many types of designs, this structure can provide performance close to that of designs with customized control while still offering the flexibility and ease of design that programmable control and a modular datapath provides. A powerful feature of the architecture is the per-microinstruction programmability of its datapath into clusters of independently executing serial chains. The problem of having to wait for the longest datapath chain to complete is solved in an approach by allowing run-time formation of decoupled clusters of datapath units. In this approach, the microengine can thus continue to fetch and execute new microinstructions without having to wait for the completion of the decoupled clusters, reducing overhead related to datapath computation. The ability to form decoupled series/parallel clusters allows a richer set of schedulings and thereby promises to increase the efficiency of a microengine implementation significantly.

Timing assumptions that are considered safe have been used to reduce various control overhead. These timing assumptions could potentially be incorporated into automated microengine generator tools, thus avoiding case by case validation. Examples of hiding control latency is to let branch calculation, propagation of data signals through input MUXes, and meeting register setup constraints be performed concurrently with completion synchronization, and also pipeline the datapath execution with branch prediction and fetching of the next instruction. Using an approach where requests are always acknowledged even for datapath units not executing, thus returning all control signals to the same state at the end of each execution cycle, facilitates efficient logic control structures for both two and four phase implementations.

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 in Verilog−+ to be realized as both hard-wired and microengine implementations.

References

[1] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark, "One billion transistors, one uniprocessor, one chip," *IEEE Computer*, vol. 30, no. 9, pp. 51–58, Sept. 1997.
[2] Gill A. Pratt and John Nguyen, "Distributed synchronous clocking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 3, pp. 314–328, Mar. 1995.
[3] James Montanaro et.al., "A 160-mhz, 32-b, 0.5-w CMOS RISC

microprocessor," *Digital Technical Journal*, vol. 9, no. 1, pp. 49–62, 1997.

[4] A. Davis, B. Coates, and K. Stevens, "The Post Office experience: Designing a large asynchronous chip," in *Proc. Hawaii International Conf. System Sciences*. Jan. 1993, vol. I, pp. 409–418, IEEE Computer Society Press.

[5] 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.

[6] 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.

[7] Andreas Nowatzyk, Gunes Aybay, and Fong Pong, "Design of the S3MP processor," 1995.

[8] 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.

[9] 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.

[10] 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.

[11] Maurice Wilkes, "The best way to design an automatic calculating machine.," July 1951.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Scott Hauck, Gaetano Borriello, and Carl Ebeling, "TRIPTYCH: An FPGA architecture with integrated logic and routing," in *Advanced Research in VLSI*, pp. 26–43. MIT Press, 1992.

[20] 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 *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 1997, pp. 140–153.

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

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

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

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

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