

Asynchronous Microengines for Efficient High-level Control

Hans Jacobson and Ganesh Gopalakrishnan*
Department of Computer Science
University of Utah
{hans,ganesh}@cs.utah.edu

Abstract

Asynchronous (self-timed) circuits are quite natural for realizing control-intensive designs. Many such designs are of reactive nature and inherently complex due to complicated communication protocols. In these situations programmable controllers are preferable over hardwired controllers to allow design decisions to be bound late, help correct errors that may slip through the verification process, and even permit run-time modification of control algorithms to best suit the current situation. Virtually all recent work in asynchronous controller design focusses on generating hardwired controllers. In this paper, we propose an architecture for programmable asynchronous controllers in the form of a microprogrammed asynchronous “microengine”. Architectures utilizing both two-phase and four-phase handshaking are proposed. The datapath structure of the asynchronous microengine is modular and easily extensible, facilitating changes during the design phase. We ensure high performance of the asynchronous microengine by exploiting concurrency between operations and employ efficient control structures. Initial results show that the proposed microengine can yield performance close to that offered by automated high-level synthesis tools targeting custom hardwired burst-mode machines for control.

1: Introduction

Designing *reactive* and *control-intensive* digital circuits, especially where multiple unsynchronized data inputs are involved, and where the computations and control decisions take variable amounts of time, is a challenging problem in many ways. If one uses the synchronous clocked design style, one has to do considerable timing analysis to ensure that all the clock-cycles are “optimally” filled in all control-flow situations, and/or adopt clock gating/skewing techniques that are not fully understood yet. These are well known to be hard problems. Furthermore, design changes even of a local nature can cause global ripple-effects of control schedule changes in order to regain optimality. Asynchronous (self-timed) circuits are quite natural for realizing circuits of this style, and encouraging results are being obtained by many, for example in designing communications components used in multiprocessors [2], hardware to network portable electronic devices [11], and audio-electronics hardware [16].

This paper addresses one shortcoming in the design spectrum of modern asynchronous controller realization methods, including the ones used in the above cited works: they are hardwired! Programmable control is often desired for several reasons. Many designs in this domain are very complex, often tending to be at the limits of our (human) design abilities. In these circumstances, it is good to have the flexibility to bind design decisions late. Our understanding of the debugging process in this domain is also incomplete, and hence one would like the ability to correct errors even after fabrication. These features are especially handy during prototyping. Allowing run-time modifications of control algorithms may also offer performance benefits. Supporting families of component types, such as bus adaptor chips, may also be facilitated by programmability. Three

* Supported in part by NSF MIP-9622587

examples of systems realized using programmable control (but not using asynchronous control) are the S3MP processor [12] which uses a microprogram engine, the FLASH processor [10] which uses a processor-core, and the Utah Avalanche [1] processor which is expected to use one of these styles. The method proposed in this paper combines the advantages of programmability and self-timing. More specifically, the main contribution of this paper is the design and experimental evaluation of a fully asynchronous microprogrammed control organization [17], a *microengine*, emphasizing simplicity, modularity, and high performance. We will also show that asynchronous design methods can be used advantageously in the design of microprogrammed control and datapath structures.

This paper is organized as follows. After motivating our approach of targeting asynchronous microengines for efficient high level control and briefly surveying related work, we describe our proposed asynchronous microengine architecture in detail. This paper will cover the general structure and operation of the microengine as well as go into details about optimizations to enhance its performance. A performance comparison using an asynchronous circuit benchmark showing encouraging initial results will also be presented.

1.1: Motivations for combining self-timing and microprogramming

Several commonly held advantages of the asynchronous design style are available even when designing microprogrammed control and datapath structures.

Modularity: Function blocks in the datapath can be modularly upgraded during the design iteration phase without risk of violating clock scheduling.

Average case completion: Asynchronous datapaths can take advantage of variable completion times due to input data and environmental conditions. Recent studies have shown up to 19% average latency reduction for random input data in 32 bit Brent-Kung adders using speculative completion detection [13] and 48% performance increase for a 32 bit differential equation solver [18] under normal operating conditions, compared to a worst case synchronous design. We will show that the control time overhead in our microengine approach can be kept low by hiding it in the time for concurrent data propagations, thus retaining the inherent advantage of average case completion offered by asynchronous datapaths.

Sequential/Parallel Evaluation: Explicit acknowledging of completions by asynchronous datapaths allows a richer set of ways in which to sequence actions. For example, our approach allows several datapath functions to be computed sequentially (*chained* [3]) without having to fetch new microinstructions in between, or having to wait for the next clock edge after the chain is completed, as in a synchronous realization. The same functions may also be computed in a parallel or pipelined mode when throughput is more important than latency. This approach can be generalized to allow run-time formation of clusters of parallel/sequential executions to best suit the current situation.

These advantages have sufficient impact on the ease of design as well as flexibility and performance after fabrication to motivate further exploration of asynchronous microengines.

1.2: Related work

Asynchronous microengines were investigated around the 1980's [15]. We do not know of papers (if any) in this area since then. In the approach of [15], the microengine was coded with vertical microcode and used to drive a collection of slave controllers which together with the implementation technology, "structured tiling" called PPL, could introduce considerable control related overhead. In contrast, a distinct feature of our microengine approach is that the RAM in itself provides all the control, and there is tight integration between the control and data path of our microengine. This reduces control overhead by allowing high concurrency between different parts of the microengine. Our method also allows maximum flexibility and low overhead even on small-scale designs.

Another method to obtain programmable control in a self-timed design context is by using FPGAs such as Triptych [4]. However, these and other similar FPGA structures are configuration-time reprogrammable, but not (easily) run-time configurable. In addition, the area/time figures given in [4] suggests that it will not be as efficient for the class of circuits we have in mind for our microengine.

For example, an 8-bit adder in Triptych has a 42nS delay and takes 40 routing and logic blocks (RLBs) while we our microengine can use a customized adder that can operate much faster.

2: Microengine architecture

The basic microengine structure (Figure 1) consists of a microprogram store, next address logic, and a datapath. As usual, 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.

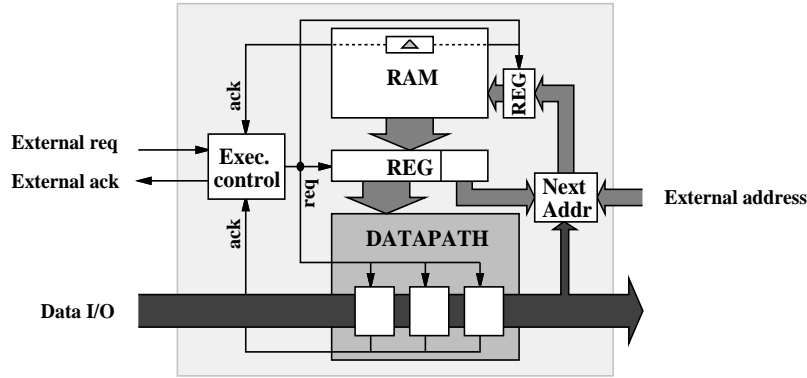


Figure 1. High Level Structure

The only obvious difference between this microengine and well-known synchronous implementations of microengines is the manner in which sequencing, microinstruction fetching, and microinstruction execution are performed. In the asynchronous case this is accomplished by request acknowledge handshaking between the execution control unit and the datapath elements, rather than being driven in lock-step by a global clock. The execution control unit detects when the datapath elements selected for execution have completed their respective operations. The collective completion of datapath elements then marks the end of one execution cycle of the microengine, after which it may start another cycle or await a request from the environment.

A number of issues related to modularity of individual datapath elements, and performance due to branch prediction, completion detection, control overhead, action sequencing, and average case completion will be discussed in the following sections.

2.1: Microengine structure and operation

The microengine starts its execution with a request from the environment at a specified entry-point of the microprogram. In response to this request, the execution control unit generates a global (within the microengine) request which first latches a new microinstruction, and also causes the datapath elements that are set up for computation in the current cycle to start executing. The microinstruction specifies which datapath element are set-up for computation, the multiplexor signal values to determine operand sources and result outputs, and the parallel/sequential connectivity of the datapath elements. As the datapath elements finish their computations, acknowledges are sent back to the execution control unit. Meanwhile, the branch detect unit evaluates the conditional signals generated by the datapath to help determine the next address. After gathering the acknowledges, the execution control unit determines whether the externally requested operation has been completed or not. In the former case, it repeats the above cycle (called a *request cycle*) beginning with the generation of a global request signal. In the latter case, the execution control unit sends an acknowledge back to the environment and then awaits the next invocation. The following subsections will describe the structure and execution of the microengine in more detail.

Next address generation

To increase the performance of the microengine, it is desirable to fetch the next microinstruction in parallel with the execution of the current microinstruction. This pipelining works smoothly except when branches are involved.

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 branch prediction solution based on empirical data [5] according to which average branch probabilities favors a branch taken approach, and the importance of allowing the designer to easily change branch prediction strategy. Each branch can therefore 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, and a delay slot is used to hold the next microinstruction supposedly branched to or the next sequential instruction, depending on the selected branch prediction strategy. This way the instruction predicted to be executed next, and the address to branch to, are readily available and there is no need for extra registers and MUXes to store and select the previous address in case the wrong branch was chosen.

Microcode fields and execution

The microinstruction illustrated in Figure 2 consists of a number of N bit *set-* fields, each bit positionally associated with a corresponding datapath unit. The *set-mux* field is used to represent input and output MUX settings. The *set-seq* field controls if a datapath unit is to be executed in parallel or sequence with respect to other datapath elements. The *set-exe* field controls if a datapath element is to execute during the current request cycle. The *set-bra* field is used to specify which conditional signals returned from the datapath are to be evaluated by the branch detect unit. If any of the *set-branch* bits are non-zero, the instruction is a branch. The *next-addr* field holds the next address for a branch instruction, and is don't-care otherwise. The *bra-pred* field indicates the branch prediction strategy; more specifically it is used to identify if the instruction in the delay slot belongs to the taken or not taken branch of the program. Finally, a *done* bit (not shown) indicates whether the microengine has finished the computation requested by the environment.

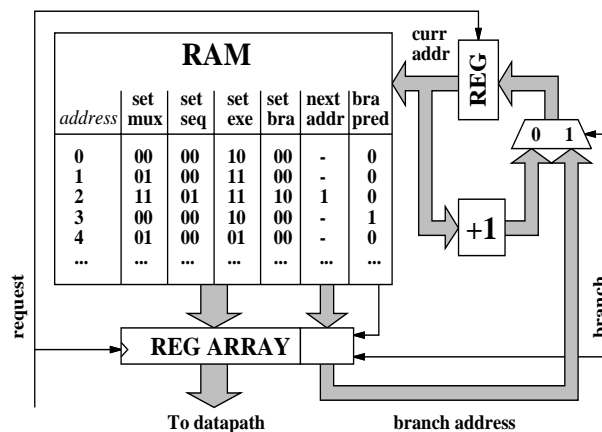


Figure 2. Example of Microengine Program

The execution of the example program in Figure 2 proceeds as follows. Instructions will be referred to by their corresponding address. Assume that the two bits in each *set-* field corresponds to their own datapath elements, say a and b respectively. Assume we start with the microinstruction at address 0 (instruction 0). During the first request cycle, instruction 0 is executed by the datapath while instruction 1 is fetched in parallel. According to the example in Figure 2, only datapath

element a is set to execute. The next request cycle propagates instruction 1 to the datapath for execution and also fetches instruction 2. In our example, datapath elements a and b are set to execute in parallel by instruction 1. During the next request cycle, instruction 2, which is a branch instruction, is propagated to the datapath for execution. This instruction executes datapath elements a and b in sequence. If the branch condition is true, the program is supposed to branch to instruction 0. Since ‘branch taken’ is predicted (indicated by the *bra-pred* bit of instruction 3 being high), the address to branch to in instruction 2 is set to 1 while instruction 3, which forms the delay slot, contains a copy of instruction 0. If the branch condition evaluates to true, instruction 3 is executed and address 1 is propagated to memory as the next address. If false, instruction 3 is cleared (not executed) and the current address incremented by 1 (4 in our example) is instead propagated to memory as the next address.

If a ‘branch not taken’ policy had been employed, the address to branch to and *bra-pred* would have been set to 0, and the instruction at address 4 in our example would have filled the delay slot (instruction 3). If the branch was false, the program would execute instruction 3 and propagate address 4 to memory as next address. If true, instruction 3 would be cleared and address 0 propagated to memory as next address.

The penalty of a correctly predicted branch thus becomes zero while an extra memory fetch cycle must be performed if it was wrongly predicted. The extra memory fetch cycle only depends on the delay of the memory, since no datapath elements are executing, resulting in a quick fetch and thus reducing the time penalty. In addition to simple and fast next address logic, the branch prediction strategy and its implementation allows for very convenient representation of one-instruction loops which occur frequently in applications of iterative nature.

2.2: Microengine and datapath interaction

The following sections will discuss the request/acknowledge handshake between the microengine and its environment and between the microengine and its local datapath, and how branch evaluation is performed.

Microengine execution control

There are many ways of realizing a structure for request/acknowledge handshaking between the microengine and the datapath elements. Since all datapath elements must synchronize with the RAM before a new micro instruction can be latched, there is little to gain by generating separate request signals to individual datapath elements. An approach of having only one global request signal that decides when to fetch a new microinstruction from memory as well as cause all the datapath elements to execute is therefore used. This approach reduces the complexity of the request control logic necessary, as well as simplifies parallel datapath element 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 elements. For implementation of the request generation logic, burstmode [2, 14, 19] type of asynchronous state machines are used. The operation of a burstmode state machine allows the acknowledge signals from the datapath elements that are generated in response to the global request to arrive at the state machine inputs in arbitrary order at arbitrary times, further simplifying timing analysis.

Two approaches for generating the global request signal have been examined, yielding very different logic complexity. The first is based on the assumption that only datapath elements that were requested to perform a computation should respond with an acknowledge. Using the set-execute level signals, the request generation logic could then figure out what acknowledges were required in order to generate the next request event. Although this looks like a nice approach in theory, the request generation logic complexity has a polynomial growth when the four phase protocol is considered, and an exponential growth when the two phase protocol is considered. This will severely limit performance as the number of datapath elements grows. The second approach is based on the assumption that every datapath module generates an acknowledgement in response to every

request. While this requires some extra bypass logic local to each datapath element to generate an acknowledge in case the datapath element is not required to do anything in a given request cycle, the complexity of the request generation logic grows only linearly, requiring only two literals per acknowledge signal. This is because the state machine no longer needs to keep track of which signals it should receive an acknowledge from before generating a new request. An additional benefit of this approach is that all acknowledge signals in the datapath are kept in “phase”: in other words, they have the same logic value at the end of every request cycle. Since this is true for both two and four phase protocols, the *same* request generation logic can be used in both cases. An abstract event based FSM and resulting complex gate implementation using the 3D synthesis tool [19] is illustrated in Figure 3(a). The respective n and p transistor networks can then be decomposed into balanced tree structures of gates to simplify timing analysis. This request generation logic then forms what is called the *execution control unit (ECU)* used to generate a new event on the global request signal.

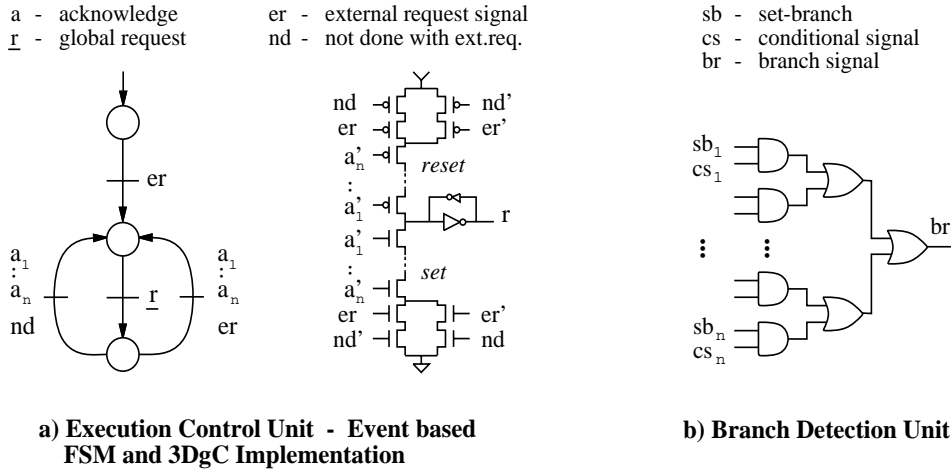


Figure 3. Execution Control and Branch Detection Units

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 elements to execute. For both the two and four phase case, the *not done* signal in Figure 3(a) is generated by a SELECT element (not shown) connected to the latched *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, after synchronizing with all acknowledges, an event is 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. More details are given in [7].

Branch detection

To control the program flow executed by the microengine, the current state of the datapath must be communicated back to the microengine’s next address logic. This functionality is provided by the *branch detection unit (BDU)* which takes a set of conditional signals from the datapath and a set of set-branch control signals from the memory. The set-branch signals then decide what conditional signals currently are of interest to evaluate.

The approach used to implement the branch detect logic is based on the following observations. First of all, the generated branch signal will be used as a synchronous clear signal of the microinstruction register and also as a control signal of an address selection MUX. If the branch signal can be assumed to have stabilized before the next arriving event on the global request signal, it

does not have to be hazard free. (The branch evaluation can also be performed concurrently with the ECU’s completion detection activity.) Alternatively, if an event signal based approach without timing assumptions is used, the logic complexity, and thus delay, of the branch detect unit would increase significantly. Therefore, the branch detect unit is implemented in level-based non hazard free logic. The branch condition is then evaluated by using the set-branch signals to mask the conditional signal of interest as illustrated in Figure 3(b).

2.3: Datapath execution control

A powerful feature of the proposed architecture is its ability to dynamically form clusters of datapath elements for independent series/parallel execution during run-time. To support this fine grained control over execution, a limited form of control structure is associated with each datapath element (*DPE*) as shown in Figure 4(a). This control structure is called the *RAS-block* (Request/Acknowledge/Sequence) and is illustrated in Figure 4. The purpose of this block is to provide control over local request-acknowledge generation and sequencing of actions. Given the *set-execute* (*se*) and *set-sequence* (*ss*) bits from the current microinstruction, the RAS-block controls if its corresponding datapath element is supposed to execute during this cycle, and in what mode (sequential or parallel), with respect to other datapath elements. When the RAS is set to operate in parallel mode, the global request signal generated by the ECU is propagated directly to the datapath element. If set to operate in sequential mode however, the RAS instead propagates the acknowledge called *sequence-request* (*sreq*), from datapath elements set to execute before it in causal order as its request signal. If the datapath element 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 can be performed by a MUX (referred to as the *sequence control MUX* for the purpose of exposition; a two-input realization is shown in Figure 4) that switches between which signal, the global request or one of the sequence-request signals generated by any one of the datapath elements, to propagate depending on the current mode of operation. 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 4). This selection between global request or sequence request will allow the microengine to set the mode of operation for each datapath element dynamically on a per request cycle basis.

Carrying the above idea further along, in general it will be necessary for one RAS block to wait for the completion of an arbitrary set of concurrently executing datapath elements before generating the request signal to *its* attached datapath element. An elegant way to realize such high flexibility is illustrated in Figures 4(b,d). Notice that these logic blocks are realizing generalized MUX+AND and MUX+C-element structures for the four and two phase cases respectively. Given a set of *sequence request* signals from other datapath elements this structure can, by using a corresponding set of *set-sequence* signals from the microinstruction, synchronize with all possible combinations of these datapath elements. 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. This efficient structure is, again, possible due to the method of always generating an acknowledge, subsequently also keeping sequence request signals in phase.

In general, sequencing actions between datapath elements will always be faster than starting a new request cycle, because the latter entails detecting completion of all datapath elements and fetching a new microinstruction. However, one should keep the stack height of the structure in Figures 4(b,d) low, in order to gain a significant performance edge. This is easily achieved by limiting the number of series/parallel combinations allowed, as almost all practical realizations seldom call for the “infinite flexibility” of all possible combinations. Any exceptional cases that require an unsupported sequential mode can always be achieved by executing multiple request cycles.

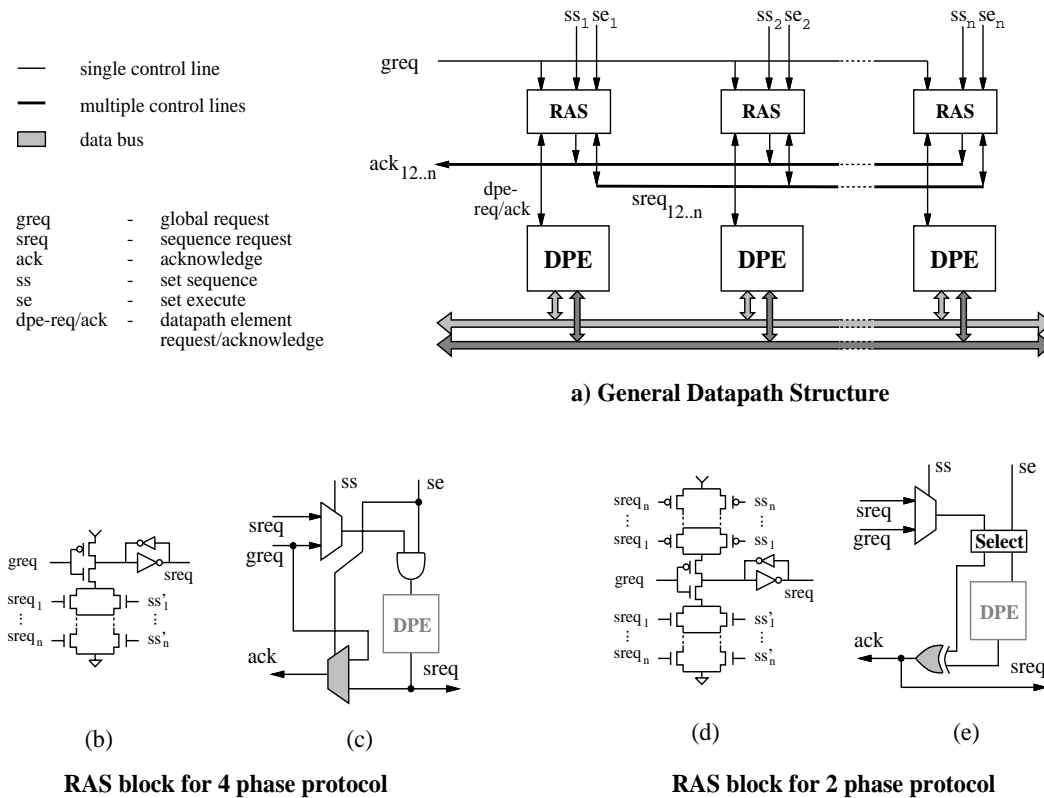


Figure 4. Datapath Structure and RAS-blocks

Request-acknowledge control

Besides sequencing control, the RAS must also provide means to correctly perform an internal request-acknowledge handshake with its datapath element. This includes generating a request to the datapath element if it is scheduled to execute during the current cycle, and also providing a bypass path for acknowledge generation if it is not.

A request signal should only be received by the datapath element 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 element if it is not setup to execute. Correct propagation of the internal request signal to the datapath element can in the case of four phase protocol be implemented by a simple AND-gate. The AND-gate is then enabled if the datapath element 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 element. An element that can generate events to either the datapath element, 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 current value of the level signal (set-execute).

The bypass path, illustrated by the shaded components in Figures 4(c,e), 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 element is not scheduled for execution. Otherwise the MUX is set to propagate the internal acknowledge generated by the datapath element. In the case of two phase a MUX cannot be used since the state (value) of the input signals are not known. An element 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.

The inputs of the XOR-gate are then connected to one of the outputs of the SELECT element and also the internal acknowledge from the datapath element.

2.4: Datapath element structure

Each datapath element 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 element however, can be implemented in an arbitrary fashion.

The structure of a datapath element is important in order to know what kind of optimizations can be done to minimize control overhead. We will therefore consider a convention on the structure of a datapath element. While a datapath element does not have to comply fully with this convention when the unoptimized microengine control structure to be presented in Section 3.1 is used, it should be followed if the optimizations to be presented in Section 3.2 are to be useful. The only requirement that must be followed is that registers be provided to latch input data that, by the DPE designers definition, are not allowed to change during the course of internal computation.

Datapath elements used in our microengine can be divided into two categories, *basic*, whose only form of internal control is completion detection, and *complex* having a more advanced internal control structure. A complex datapath element may in itself be a hierarchy of microengines. It is assumed that a complex datapath element can be realized using basic datapath elements operated by the internal control structure. We will therefore only consider the internal datapath structure of basic datapath elements. We now consider three kinds of register placement:

1. *Use of input and output registers:* Due to the extra area and data propagation overhead incurred by latching both input and output data, this is typically only used for larger modules (e.g. *complex* datapath elements) with internal control structures that may need to hold certain inputs and outputs stable over multiple clock cycles.
2. *Use of output registers only:* Having only output registers is an acceptable choice in *synchronous* systems where all submodules operate in every cycle and latch data in parallel. It also is an acceptable choice in a system that always operates in a sequential fashion, where each sequential stage is requested to execute by completion signals from the previous stage. Using output registers only runs into problems when modules are allowed to dynamically operate in mixed series/parallel modes. In this case, a datapath element must have the ability to latch data before and after a computation has completed depending on the mode of operation. This results in a complex control structure internal to each datapath element that is dependent on the current mode of operation for completion detection and data latching. Additional problems with this approach are discussed in [7].
3. *Use of input registers only:* Having only input registers suits well with parallel and sequential modes as well as in mixed modes of operation. Since the input registers are always latched *before* the requested computation is started, the control structure for completion detection is independent of the mode of operation. This is therefore the preferred approach concerning register placement in a *basic* datapath element.

As far as MUX placement is concerned, they are commonly placed at the inputs, rather than the outputs of registers, as it results in a reduction of the number of registers needed. This approach is thus proposed for *basic* datapath elements in our microengine. The imposed convention on the structure of a basic datapath element then, is that illustrated in Figure 5. According to this convention, each datapath element should have a set of internal input MUXes and input registers. Output MUXes and registers are also allowed, but not required. Bundled data is assumed when communicating data sequentially between datapath elements. When a datapath element operates in parallel mode, it is assumed that input values are latched as soon as the request signal arrives, before new values from other datapath elements may arrive.

For the purpose of optimization it is assumed that worst case propagation time through input MUXes and registers, setup and hold times for input registers, and buffering of the register request

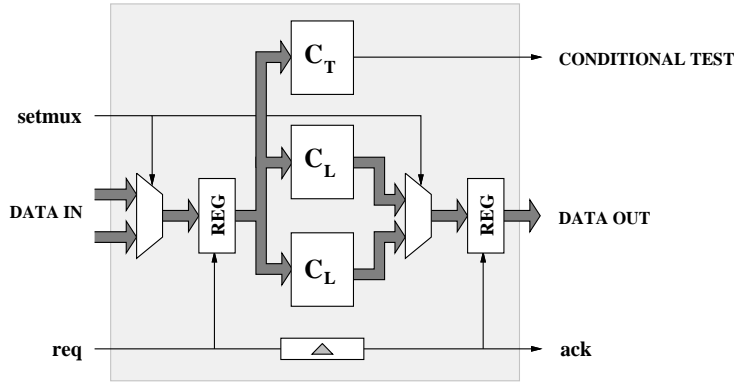


Figure 5. Example of Datapath Element

signal are provided as design parameters by designers of each datapath element.

3: Reducing control overhead

The intuitive functionality and logic presented for the microengine and RAS-block in Section 2.3 while explaining the higher level concepts of the microengine architecture in a clear fashion, are not very optimal seen from a performance point of view. This section presents some optimizations that allow actions in control and datapath to take place concurrently, thus reducing control overhead.

3.1: Unoptimized approach

The performance problem of the solution presented so far is due to the microinstruction being latched using the same request event that triggers the datapath elements to execute. To setup the RAS for no, parallel, or sequential execution and also ensure latching of correct input data requires timing constraints on the arrival of the request signal to the datapath.

One constraint requires the new microinstruction to propagate through latches and meeting setup times of components internal to the RAS before the request signal is allowed to reach the RAS. Another constraint requires the set-mux control signals of the new microinstruction to propagate through the register array, switch and allow propagation of data signals through input MUXes, and meet setup times of the input registers, before the request signal is allowed to arrive at the datapath elements. To meet these constraints, the request signal to the datapath has to be delayed suitably. Assuming the second constraint is the most stringent, the resulting delay that has to be introduced can be in the order of several gate delays. With this delay in addition to the control overhead caused by the ECU, the total delay related to control can become quite significant.

We will show that this overhead can be significantly reduced by fetching the next microinstruction concurrently with the ECU performing completion synchronization. In addition, if the suggested structure of a datapath element as outlined in Section 2.4 is followed, the overhead can be further reduced by allowing simultaneous propagation of data signals through input MUXes and meeting register setup times.

3.2: Optimized approach

The approach of using the same request event to latch the new microinstruction and to trigger the datapath elements to execute causes extra control overhead due to the restrictions on signal arrival order presented in Section 3.1. These restrictions are all a result of meeting propagation and setup times of datapath signals which are dependent on the control signals of the new microinstruction. It would therefore be desirable to allow these control signals to be latched at an earlier point

in time, allowing the correct data signals to propagate concurrently with the ECU performing completion synchronization. Our goal with the optimized control approach then is to reduce the control overhead by allowing concurrent completion synchronization, microinstruction latching, and data propagation.

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.

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 6(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 would allow 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 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 6(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 4(d) is then connected to the $sreq$ inputs of the sequence logic in Figure 6(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

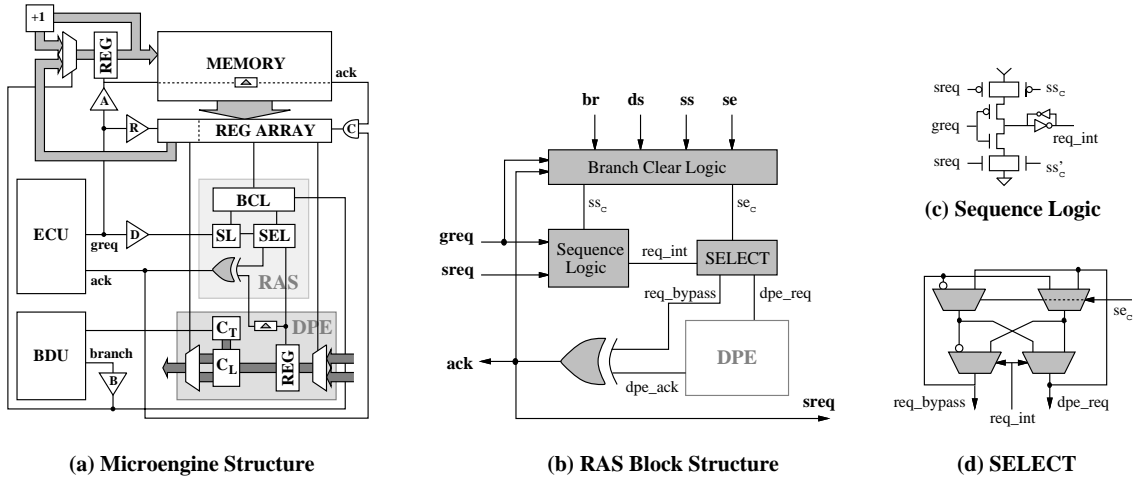


Figure 6. Optimized Two Phase Structure and RAS Block

asynchronous branch clear logic local to each DPE. The structure of the RAS block under the assumption of early instruction fetch is illustrated in Figure 6.

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 external data are performed during the passive phase. An overview of this optimized architecture is illustrated in Figure 7(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 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 signal, 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 7 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 4(b) is connected to the *sreq* inputs of the SEQ/REQ logic in Figure 7(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

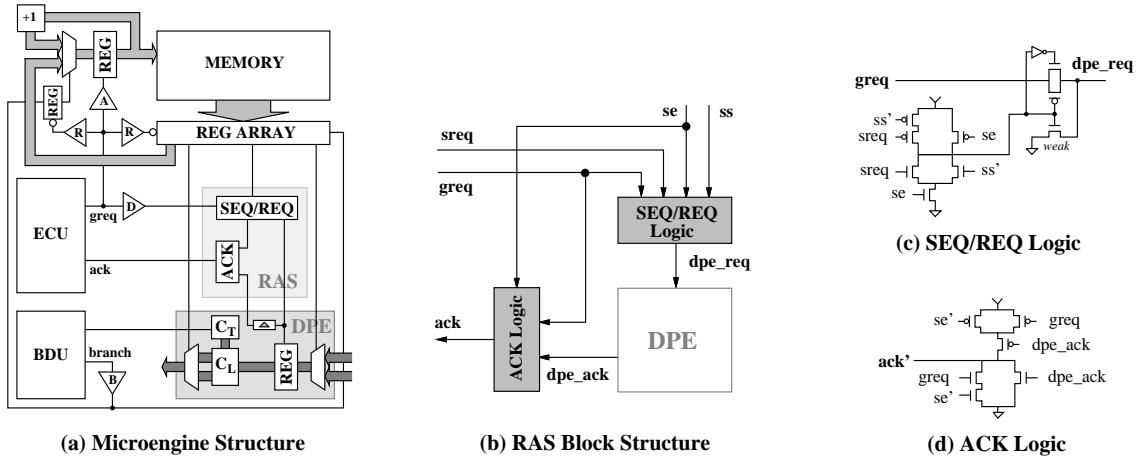


Figure 7. Optimized Four Phase Structure and RAS Block

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.

4: System level operation and timing analysis

When hiding control overhead in the concurrent propagation of signals in the datapath inherent timing assumptions are made as to the arrival order between control and data signals. This section will therefore first discuss the overall system operation of the microengine using two and four phase protocols and then present corresponding timing constraints that must be satisfied to ensure correct operation. Timing assumptions not on critical paths or otherwise of less importance have been left out in this paper but are presented in full detail in [7]. All timing assumptions have single sided constraints and can always be satisfied by inserting appropriate delays.

The following conventions are used in the timing diagrams of Figure 8. Figures 6(a) and 7(a) illustrate the architectures corresponding to these timing diagrams. A vertical line crossing the beginning of an event signal marks that event to be the reference of this time-line. If a line crosses the middle of an event, the edge can occur either before or after this time-line. If an event or data signal change occurs between two lines, then the event or change must occur in the interval of these two time-lines. Shaded boxes correspond to unstable level signals. A line crossing the beginning of a data signal change indicates it is allowed to go from stable to unstable at this point in time but not before. A line crossing the end of a data signal change indicates that the signal must have stabilized at this time.

The following conventions and abbreviations are used in the timing equations to follow. 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, *CLR* for the synchronous clear logic in case of four phase, *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, *branch* for branch signal, *req* and *sreq* for global and sequence request signals, *DP*, *ADR*, and *MI* for datapath, next address, and microinstruction respectively, and *REG* for register.

4.1: Two phase

The system level operation and structure of the optimized microengine implementation using the two phase protocol is illustrated in Figure 8(a) and Figure 6(a) and is described below. In the communication between environment and microengine, bundled data is assumed. When requesting the microengine to perform a computation, address and data supplied by the environment must be stable at the inputs to the microengine before the request is allowed to arrive at the ECU. In response to the request from the environment, the ECU generates an event on the global request signal internal to the microengine. At this time, a new microinstruction is propagated to the datapath at the same time as the next address is propagated to the memory. If the next address is supplied from the environment, this first propagated instruction should therefore not be executed. This can be achieved by letting the last instruction fetched by the previously requested computation be a nop instruction. As the datapath elements finish their computations and the memory completes its instruction fetch, their respective acknowledges are sent back to the ECU. Each acknowledge also latches the part of the next microinstruction corresponding to its DPE. For each request cycle until the microengine has completed the computation requested by the environment, a not done event signal is also generated and sent to the ECU. These event signals can occur in any order. The only timing restriction at this point is that the microinstruction signals have stabilized at the inputs to the register array at the time the memory acknowledge is generated. The memory and the individual DPE acknowledges are synchronized by C-elements to make sure the correct microinstruction is latched. In parallel with the ECU synchronizing the acknowledges, the BDU evaluates the branch condition. The branch result must be available in time for the next address signals to be calculated and stabilized at the inputs of the next address register, and the BCL to evaluate, by the time the ECU is finished and generates a new event on the global request. Microinstruction signals such as output set-mux signals, must be latched by the global request rather than individual DPE acknowledges to not change the data outputs of the DPEs prematurely. This global request event then marks the end of the first request cycle. In Figure 8(a), four more such cycles are executed before the microengine has completed the computation requested by the environment. The last cycle signals an event on the environment acknowledge signal instead of the not done signal, telling the environment it has completed the requested computation and that eventual data is available at the outputs of the datapath. To comply with the bundled data protocol, the environment acknowledge is not generated until after all DPE and memory acknowledges have occurred. Since the ECU does not receive the not done signal, it will remain quiescent until the environment requests a new computation.

The following timing equations illustrates timing constraints that must be satisfied for correct operation. The architecture corresponding to this optimized two phase microengine is illustrated in Figure 6(a).

Branch prediction. Since no execution should take place if the branch prediction was incorrect, the branch 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 equation holds.

$$(1) \quad ECU + DP_reqbuf + SL > BDU + branchbuf + BCL$$

Where the delays of *SL* and *BCL*, and also *DP_reqbuf* and *branchbuf* are comparable, reducing the

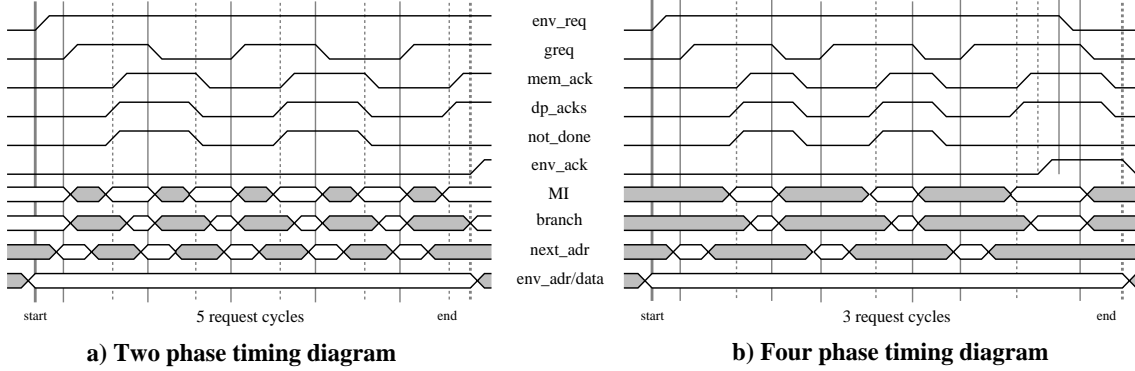


Figure 8. Microengine Timing Diagrams

constraint in all practical aspects to only require the delay through the *ECU* to be greater than through the *BDU*.

The following equation ensures latching of correct next address value, i.e. that MUX propagation delay and register setup times are met before the global request arrives at the next address register.

$$(2) \quad ECU + ADR_reqbuf > BDU + MUX + ADR_REG_{setup}$$

Data latching. The assumption that data values are latched correctly by datapath elements operating in *parallel* mode is correct if the following equations are satisfied.

$$(3) \quad ECU + DP_reqbuf + RAS_req + DPE_reqbuf > C + MI_REG + MUX + DPE_REG_{setup}$$

$$(4) \quad DP_reqskew + DPE_REG_{hold} < DPE_REG + MUX$$

Where equation 3 ensures that new data values have time to propagate through MUXes and meeting register setup times before the next request arrives at the datapath element. Equation 4 ensures, assuming delay internal to a datapath element is unknown, no new values can propagate from the outputs to the inputs of registers latching data in parallel. $dp_reqskew$ accounts for skew related to difference in request arrival time at each RAS as well as to each DPE's input registers due to wire delay and signal buffering.

The assumption that data values are latched correctly by datapath elements operating in *sequential* mode is correct if the following equation, is satisfied.

$$(5) \quad XOR + RAS_{sreq} + DPE_reqbuf > MUX + DPE_REG_{setup}$$

This equation then essentially describes the bundled data assumption constraint.

4.2: Four phase

The system level operation and structure of the optimized microengine implementation using the four phase protocol is illustrated in Figure 8(b) and Figure 7(a) and is described next. As in the two phase case, communication between environment and microengine assumes bundled data, requiring external address and data to be stable at microengine inputs before the request is allowed to arrive at the ECU. In response to the request from the environment, the ECU initiates the active phase of the four phase handshake by generating a rising edge on the global request signal internal to the microengine. At this time, the next address signals must have stabilized and are propagated to the memory, and the datapath elements requested to execute. The last instruction from the previous requested computation was already propagated to the datapath at that time. If the next address is supplied from the environment, this instruction should therefore be a nop as in the two phase case. As the datapath elements finish their computations and the memory completes its instruction fetch, their respective acknowledges are sent back to the ECU. A not done signal is generated for each event on the global request signal until the microengine has completed the requested computation. These event signals may occur in arbitrary order. The microinstruction

signals must have stabilized at the inputs to the register array at the time the memory acknowledge is generated. In parallel with the ECU synchronizing the acknowledges, the BDU evaluates the branch condition. The branch result must be available in time for the next address signals to be calculated and stabilized at the inputs of the next address register, and allow the synchronous clear signal to clear the register array, by the time the ECU has finished its synchronization. After the ECU has received events on all acknowledges and the not done signal, the passive return to zero phase is initiated by generating a falling edge on the global request signal. During the passive phase, all datapath elements are returned to zero in parallel, and the microinstruction fetched during the active phase is propagated to the datapath. As the branch signal must be kept stable until the next address is latched during the active phase, but may change value due to the new microinstruction being propagated, the branch signal fanout going to the next address logic must also be latched at the falling edge on the global request. Once the ECU has received falling transitions on all acknowledge and the not done signal, it generates a rising transition on the global request signal and one request cycle has completed. In Figure 8(b), two more such cycles are executed before the microengine has completed the computation requested by the environment. The last cycle then signals an event on the environment acknowledge signal instead of the not done signal. The environment responds to this acknowledge event by lowering its request signal, causing the global request to go low. Once all acknowledges has been returned to zero, a falling transition on the environment acknowledge indicates the requested computation has completed and eventual data is available at the outputs of the datapath. The ECU now remain quiescent until the environment requests a new computation.

The following timing equations illustrates timing constraints that must be satisfied for correct operation. The architecture corresponding to the optimized four phase microengine is illustrated in Figure 7(a).

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

$$(6) \quad ECU + MI_reqbuf > BDU + branchbuf + CLR$$

Latching of correct next address value is ensured if the following equation is satisfied.

$$(7) \quad ECU + ADR_reqbuf > MUX + ADR_REG_{setup}$$

Data latching. The assumption that data values are latched correctly by datapath elements operating in *parallel* mode is correct if equation 4 presented earlier, and the following equation are both satisfied.

$$(8) \quad DP_reqbuf + RAS_{rtz} + ECU + DP_reqbuf + RAS_{req} + DPE_reqbuf > MI_reqbuf + MI_REG + MUX + DPE_REG_{setup}$$

The assumption that data values are latched correctly by datapath elements operating in *sequential* mode is correct if the following equation, is satisfied.

$$(9) \quad RAS_{sreq} + DPE_reqbuf > MUX + DPE_REG_{setup}$$

This equation then essentially describes the bundled data assumption constraint.

5: Results and conclusions

In this paper an asynchronous microengine architecture for programmable control has been presented. The architecture provides a systematic way to hide latencies caused by control evaluation and subsequent datapropagation. 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 its ability to dynamically form clusters of datapath elements for independent series/parallel execution during run-time. 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 detection, 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 elements not executing, thus returning all control signals to the same state at the end of each execution cycle, facilitates efficient control logic structures for both two and four phase implementations.

So far two designs have been built using the presented microengine approach, an SRT divider [6], and a CD player error decoder [8]. The error decoder circuit implements error-detection on the audio information recorded on Compact Discs using a syndrome computation algorithm. To get an estimation of how efficient our microengine approach is compared to a custom control implementation using the same datapath structure, the error decoder was also implemented using our high level synthesis framework for asynchronous circuits, ACK [9]. This framework takes a high level description in Verilog+, a synthesizable subset of Verilog extended to handle channels, as input and targets customized interacting burstmode FSMs as control structure. The Verilog+ design specification of the error decoder is a faithful translation of the Tangram program presented in [8] which also enables us to compare 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 Tangram implementation described in [8] which used dual rail logic and a 1.2 micron technology was reported to have an approximate worst case cycle time of 20 microseconds, each cycle decoding a sequence of 32 8-bit input words. Using our design tool ACK to automatically generate a customized 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. Using the same datapath, the microengine implementation had a resulting cycle time of about 4.0 microseconds also using a four phase protocol (and also had a higher area cost). 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 Design Analyzer tool. It should be noted that since the datapath was tailored to a custom control implementation, control overhead in the microengine could not be hidden in concurrent data propagation. To get a fair comparison, both designs were therefore implemented without using any timing based optimizations. Better results are to be expected when timing optimizations are applied. In the context of what automated synthesis tools can achieve, also considering the datapath used and that the control structure was implemented with standard gates, these initial 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 sequence the actions of an arbitrary number of datapath elements while still being able to perform a parallel return to zero. Although a custom control implementation featuring request flow-through, which ACK does, could be used to execute arbitrary sequences of actions, this is hard to realize in a structure suited for customized controllers without introducing too much overhead.

We are currently working on generating more examples to facilitate a comparison on a broader base of designs. Other priorities are to examine more complex formations of decoupled clusters of datapath elements, allowing selftimed loops that execute without having to interact with the microengine on a per cycle basis, and evaluate advantages of allowing more elastic execution by removing the global synchronization constraint using self-generated requests on the part of the datapath elements. We are also looking into reducing control overhead when using the four phase protocol by performing two phase handshakes between ECU and RAS and local four phase handshake between the RAS and its DPE. This would potentially allow a more efficient return to zero and also allow datapaths with mixed 2 and 4 phase datapath elements, Future work also include allowing double instruction fetch while branch instructions are executing, evaluate sequence efficiency when using transparent latches, and implementing efficient complex gate realizations of RAS block and execution control unit since they are both static entities given the number of inputs. We also intend to automate the microengine synthesis procedure, and incorporate it in the ACK synthesis framework allowing descriptions entered in Verilog+ to be realized as a custom implementation, as a microengine implementation, or both.

References

- [1] CARTER, J., KUO, C.-C., AND KURAMKOTE, R. A comparison of software and hardware synchronization mechanisms for distributed shared memory multiprocessors. Tech. Rep. UUCS-96-011, University of Utah, Salt Lake City, UT, USA, Sept. 1996.
- [2] DAVIS, A., COATES, B., AND STEVENS, K. The Post Office Experience: Designing a Large Asynchronous Chip. In *Proceedings of the Hawaiian International Conference on System Sciences, Volume 1* (Jan. 1993), T. Mudge, V. Milutinovic, and L. Hunter, Eds., pp. 409–418.
- [3] GAJSKI, D. *Principles of Digital Design*. Prentice Hall, 1997.
- [4] HAUCK, S., BORRIELLO, G., AND EBELING, C. Triptych: An fpga architecture with integrated logic and routing. In *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*. MIT Press, 1992, pp. 26–43.
- [5] HENNESSY, J., AND PATTERSON, D. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.
- [6] HWANG, K. *Computer Arithmetic: Principles, Architecture, and Design*. Wiley, NY, 1979.
- [7] JACOBSON, H., AND GOPALAKRISHNAN, G. 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.
- [8] KESSELS, J., VAN BERKEL, K., BURGESS, R., RONCKEN, M., AND SCHALIJ, F. An error decoder for the compact disc player as an example of VLSI programming. Tech. rep., Philips Research Laboratories, Eindhoven, The Netherlands, 1992.
- [9] KUDVA, P. *Synthesis of Asynchronous Systems Targeting Finite State Machines*. PhD thesis, Computer Science Department, University of Utah, 1995.
- [10] KUSKIN, J., AND ET AL., D. O. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (May 1994), pp. 302–313.
- [11] MARSHALL, A., COATES, B., AND SIEGEL, P. Designing an asynchronous communications chip. *IEEE Design & Test of Computers* 11, 2 (1994), 8–21. Summer.
- [12] NOWATZYK, A., AYBAY, G., AND PONG, F. Design of the s3mp processor, 1995.
- [13] NOWICK, S., YUN, K., BEEREL, P., AND DOOPLY, A. Speculative completion detection for the design of high-performance asynchronous dynamic adders. In *Proceedings of the 1997 International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 1997), pp. 210–223.
- [14] NOWICK, S. M. *Automatic synthesis of burst-mode asynchronous controllers*. PhD thesis, Computer Systems Laboratory, Stanford University, 1993.
- [15] STEVENS, K. The soft controller: A self-timed microsequencer for distributed parallel architectures. Tech. rep., Department of Computer Science, University of Utah, Dec. 1984.
- [16] VAN BERKEL, K., BURGESS, R., KESSELS, J., RONCKEN, M., SCHALIJ, F., AND PEETERS, A. Asynchronous circuits for low power: A dcc error corrector. *IEEE Design & Test of Computers* 11, 2 (1994), 22–31. Summer.
- [17] WILKES, M. The best way to design an automatic calculating machine., July 1951.
- [18] YUN, K., BEEREL, P., VAKILOTOJAR, V., DOOPLY, A., AND ARCEO, J. 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.
- [19] YUN, K. Y. *Synthesis of asynchronous controllers for heterogeneous systems*. PhD thesis, Stanford University, Aug. 1994.