High-Level Asynchronous System Design using the ACK Framework

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

Prabhakar Kudva
IBM T.J.Watson Research Center
Yorktown Heights, NY 10598
kudva@watson.ibm.com

Abstract

Designing asynchronous circuits is becoming easier as a number of design styles are making the transition from research projects to real, usable tools. However, designing asynchronous "systems" is still a difficult problem. We define asynchronous systems to be medium to large digital systems whose descriptions include both datapath and control, that may involve non-trivial interface requirements, and whose control is too large to be synthesized in one large controller. ACK is a framework for designing high-performance asynchronous systems of this type. In ACK we advocate an approach that begins with procedural level descriptions of control and datapath and results in a hybrid system that mixes a variety of hardware implementation styles including burst-mode AFSMs, macromodule circuits, and programmable control. We present our views on what makes asynchronous high level system design different from lower level circuit design, motivate our ACK approach, and demonstrate using an example system design.

1 Introduction

Asynchronous design in the small is a difficult, but well researched problem area. There are many choices, each with their own advantages and drawbacks to specifying and synthesizing small, highly concurrent asynchronous control modules at a very detailed signal transition level [34, 44, 14, 7]. Asynchronous design in the extremely large where complete (synchronous) computer systems are interconnected by asynchronous networks is also a well-studied and understood problem [2, 17, 3, 18]. We feel there is a middle ground of system design characterized by systems that are too large to be tractable with direct hard-wired controller synthesis, and too small or special-purpose to be efficient with a general microprocessor solution. This middle-ground is where "high-level" synthesis is important, and where there is a major gap and opportunity for further research. System descriptions at this level include data paths and control components at a procedural level, and typically require extensive partitioning and refinement of both before lower level tools can be exploited.

Although asynchronous system design and synthesis has been shown to be possible using existing tools [22, 33, 8], it can be a very labor intensive process. The problem is that most existing asynchronous circuit tools target specific pieces of the design process, but not necessarily at the right level of abstraction, or with an overall approach that fits the system design view. System specifications that include complex control and significant datapaths require a different style of tool support than smaller specifications that might, for example, involve a single fine-grain controller. The distinctive features of a *system* design tool can be described using the following categories.

1.1 System Description

As its most fundamental requirement, a system design language must be able to easily describe features at the procedural level that designers think about when they describe systems. For the purpose of improving the designer's understanding of the both the design specification and the framework used to validate, optimize, and synthesize the design, we believe it is important to use the same language at the front-end, before synthesis, as well as at the back-end, after synthesis (e.g., as a structural netlist). Using a single standard language throughout the design process simplifies the designer's job in many ways, most directly by letting the designer view and understand the result of the various stages of the design. Standard HDLs such as Verilog and VHDL are widely used for synchronous system design and as a result their syntax and semantics are already well understood by designers. On the one hand this makes them a natural choice for a specification language. On the other hand, although they include support for a wide range of both high and low level constructs, one challenge with using standard "synchronous" HDLs is understanding their affinity for expressing concepts used by designers of asynchronous circuits.

An asynchronous design puts several additional demands on a specification language. Because of the nature of asynchronous control flow, it should support concurrency, sequencing, and choice in a natural way. Asynchronous constructs such as channels and signal events should also be expressed easily. Support for specification of interface timing is also important in order to interface to the environment correctly. Together with min-max timing bounds on operations internal to a module such timing annotations can aid the designer in driving the optimizations in a desired direction. Another challenge is that the simulation semantics of Verilog and VHDL, while seemingly independent of synchronous or asynchronous circuit operation, in fact have several subtle complications when a truly asynchronous system is simulated due to the event driven model used by most simulators. There are workarounds for each of these problems that might have been avoided by designing a language that directly supports asynchronous concepts, but we believe that the benefits of using a standard HDL outweigh the costs of the workarounds.

1.2 Design Exploration

Support for iterative design exploration is an important feature in a system design tool. Whereas a circuit design tool is often used once the desired controller is already fully specified, a higher-level system design is usually evolved from a much less refined specification. The system designer may want to explore a variety of organizations at a high level before refining individual components to a specific implementation. While there is no clear consensus on how such design exploration is best performed, there are a few basic requirements which play an important role in iterative exploration of design alternatives. For example, there should be provisions for feedback to the designer regarding any optimizations that the tool performs. The more the tool modifies the system description, the more important it becomes for the effect of the optimizations to be understood by the designer. Without an understanding of the effect of the optimizations it becomes difficult to drive the design in a specific, desired direction.

Another aspect which is important in design exploration is estimation of system performance, where performance may be measured in speed, size, power, or some other metric. When the synthesized circuits are a fairly close match to the specification, such as in a macromodular or programmable controller implementation, such estimation is relatively straightforward. When using AFSMs for control, however, high-level estimation at the specification level is much more difficult. The area and performance of AFSM controllers is highly dependent on how their output functions can be covered by minimized logic equations. Even very subtle differences in state assignment and signal reshufflings can result in significant changes of the final controller logic. Because of this it may be necessary to take the AFSM-based controller much closer to the actual implementation to get a good estimate of performance. In this case fast synthesis of the controller is essential to an iterative exploration approach.

1.3 Implementation Style

System synthesis can involve much larger and more complex datapath and control circuits than lower level synthesis where the partitioning has already been done. Typically, a high level specification is split into control and datapath sections fairly early in the process. This is mainly done because control and datapath are best synthesized using different algorithms and techniques. Because there is no one best way to implement circuits of either type, a system tool should support a variety of choices.

Datapath implementation styles range from static logic datapaths with matching bundled delays to precharged complex gates with completion sensing. While complex gates and completion sensing techniques can sometimes reduce the delay of a computation in the average case they currently require significant manual effort to implement. Commercially available automated datapath synthesis systems are well suited to generating static standard cell datapaths with bundled delay. Techniques such as delay borrowing (explained later in this paper) and speculative completion [36] can be used to reduce the average case penalty of using bundled delay datapaths. The modularity of asynchronous circuits allows custom designed datapaths to easily replace standard gate datapath components where necessary.

Asynchronous control styles range from hardwired controllers such as asynchronous finite state machines (AFSM) and macromodules or handshake circuits, to microprogrammable control such as microengines [26]. Each of these control styles has advantages and disadvantages and as previous research has shown [25, 26] no one control style is best suited for all situations. It is therefore important to support different control styles that can be freely mixed within the same design while keeping the implementation details as transparent as possible to the design specification.

1.4 Concurrency Management

Concurrency in a system design occurs at many levels of the system, not just the lowest circuit level. As a result there are some additional issues when it comes to efficiently implementing the different levels of concurrency. At the highest level where the interaction between modules is not necessarily performance critical, macromodule control is very well suited due to its straightforward implementation and close correspondence with the original specification. Within a module there are typically several tasks operating concurrently. Such tasks often contain frequent and iterative interactions between control and datapath and thus have a more significant influence on overall system performance. Each task might therefore be better implemented using the AFSM style of control to tailor the controller more closely to the specified behavior. Within each such task a thread type of concurrency is often featured where each thread carries out a fairly autonomous sequence of dependent computations. While such fork-join concurrency could be implemented by AFSMs which allow I/O signal concurrency there is an overhead attached to having control signals go back and forth between a controller and the datapath elements it controls. First, the complexity of the AFSM goes up as more handshake signals are needed, and second, long handshake wires introduce a delay overhead. In such situations a *chained* control style may improve performance.

In a chained control style each thread is represented by a chain of computations where control cascades through macromodules which are local to each datapath element in the chain. Once the end of the chain is reached the control is sent back to the AFSM which collects completion signals from the threads. By exploiting the locality of the control and the low forward latency of specialized chaining macromodules, control overhead in such chains can be drastically reduced compared to letting a single AFSM handle all handshaking.

1.5 The ACK Approach

ACK is our high-level system design tool that describes the design specification at a procedural level and automatically compiles the specification into an interconnection of control and datapath circuits. For design specifications ACK makes use of the standard Verilog high level description language. We have written some additional Verilog code in the form of a "package" to help model asynchronous constructs such as channels that can be included if desired. Using standard synthesizable Verilog makes it possible for ACK to leverage standard simulation tools for design validation throughout the design process. Verilog also enables ACK to leverage standard synchronous tools for datapath synthesis and timing analysis.

High-level design optimizations performed by ACK are back-annotated to the original design specification to provide comprehensible feedback to the designer to make it easier to drive the design in a desired direction. For control, ACK successfully blends several methodologies into new control structures suited for high-level design. For hardwired control, ACK supports control structures consisting of *mixed* AFSM and macromodule control which allows implementation of computation chains (fork-join threads) with very low control overhead. For programmable control, ACK supports control structures in the form of highly efficient asynchronous microengines [26]. In ACK these control styles can be freely intermixed in any fashion that best fits the system currently being designed. The synthesized controller circuits can be tech-mapped to a standard cell library, or synthesized in terms of custom complex-gate CMOS circuits for higher performance [31].

In this paper we put forth our ideas on designing asynchronous *systems* rather than designing asynchronous *circuits*. We describe the ACK framework as an example of our evolving approach and as a system design framework. We put this in context with a simple system design example and use this to motivate further research in the area.

2 Related Work

In the asynchronous community there has been great interest in developing the circuit-level and logic-level tools that allow asynchronous and self-timed circuits to be built in correct and efficient ways [44, 19, 34, 43, 16]. They are reaching a level where they can be used effectively to design fine-grained asynchronous circuits. However, they are not well suited to larger system level design where the requirement to describe the system at the level of every individual signal quickly becomes overwhelming.

System level design tools in the asynchronous world are mostly descended from the Macromodules project at Washington University in the 1970's [15]. This project was the first, and is still one of the most successful asynchronous *system* design projects. A more modern VLSI version of macromodule design was described by Sutherland [41] as micropipelines. Several system design tools followed this general approach including Brunvand's Occam compiler [11, 10], the Tangram system from Philips [9], Akella's Shilpa system [1], and the Balsa system from University of Manchester [5]. Although these tools compile to very different sets of macromodules, they have in common a language-directed approach to system synthesis. Individual language constructs are translated based on their syntax to a set of macromodule circuits that implement that statement. The macromodules, although different for the various tools, share the feature that they can be highly optimized both in circuit and layout terms. While this can be a very intuitive and simple approach to system synthesis, relying on macromodules exclusively can have a performance impact on the resulting system.

Another approach based more on program transformation than on syntax directed translation is Martin and Burns' CHP system from Caltech [12, 32]. Their approach applies a series of decompositions to the program before mapping into a simple set of composable circuits. This results in more flexibility at the circuit level, but requires correspondingly higher levels of signal detail at the specification level.

A somewhat more ad hoc approach is taken by the Amulet group at Manchester who have been building, without direct system design tool support, a series of large microprocessor systems based on the ARM processor [20, 22, 21]. They use essentially a macromodular approach but design the controller circuits by hand rather than from program descriptions. For their latest microprocessor design [21] a hand-designed approach is being used that involves a combination of macromodules, and even finer grained generalized C-element circuits (even more labor intensive!). High-level design using Balsa for non-critical-path parts of the system, the DMA controller in particular, is also being used [5, 4].

3 A Tour of ACK

ACK is our framework for asynchronous system level design. Following the ideas about system design from Section 1, the basic flow through the tool is shown in Figure 1. As shown in the figure, ACK is a high level synthesis tool that describes the desired system at a procedural level (including datapath specification), and automatically compiles that specification into interconnected control and datapath circuits. Apart from creating an automated path from high-level specification all the way down to layout, our recent work on ACK has concentrated on providing a designer friendly environment through the use of a standard HDL and standard validation tools as well as flexible and efficient control by offering a variety of hardwired and programmable control structures. The main features of ACK are listed below.

- Support for standard Verilog HDL specification
- Support for standard validation tools
- Back-annotated high level design optimizations
- Flexible control structures supporting both hardwired and programmable control
- Design partitioning and fast AFSM control synthesis
- Standard or complex gate technology mapping with timing optimizations

The rest of this section describes the current features and future additions planned for the ACK framework in more detail.

3.1 ACK Design Flow Overview

System Specification: As illustrated in Figure 1, standard Verilog underlies most of the ACK design process. The designer starts the design cycle by entering the design specification in a synthesizable subset of standard Verilog. Designers wishing to use channel-based communication can also use our Verilog channel package. Once entered, the initial design specification can be validated using the Verilog-XL simulator from Cadence and available formal model checkers. High level optimizations to extract parallelism and subexpression sharing are then applied to the specification and the changes are back annotated to the original Verilog code. The designer can now manually apply further optimizations to the design specification if desired.

High Level Synthesis: After the initial validation and optimization stage, the design is split and refined into separate datapath and controller parts. At this stage datapath components for computation and storage are allocated and expressed in behavioral Verilog. The control of the design is refined into AFSMs, macromodule, and microengine parts according to pragma hints from the designer. As with the datapath, control at this stage is also modeled in behavioral Verilog but at the individual signal handshake level. If required, partitioning of the AFSM controllers also takes place at this stage. After high level synthesis, the design is represented as structural Verilog code but with the actual instances still described in behavioral Verilog. The design at this stage can again be validated using the same testbenches as the original specification, and further manual tweaking of the design can be performed.

Datapath and Control Synthesis: The next step in the design process is to synthesize the behavioral instances of the structural Verilog code into actual gates. Synopsys Design-Compiler is used to synthesize the datapath portions of the design into standard gates. If desired, parts of the datapath can be substituted for complex gate structures manually derived with Cadence Layout-Synthesis. The behavioral controllers are synthesized and tech-mapped into standard or complex gate structures using tools of our own and from other universities.

Gate-Level Interconnect: The design is now represented by structural Verilog gate netlists. These netlists can again be validated using the original testbenches. At this stage timing analysis to derive bundled data delays and ensure compliance with fundamental mode constraints is performed using Synopsys Design-Analyzer.

Transistor Layout: The next stage in the synthesis process is to generate an actual layout for the design. This has so far been done using the commercial tool Epoch from Cascade, however we are currently in the process of switching to Cadence. Once a layout has been produced, final timing analyses and design validations can be performed using a post-layout extracted Verilog switch level model with Verilog-XL, or through a SPICE deck with HSPICE.

3.2 Procedural Level Description

As described in Section 1, a system level design tool should support system descriptions at a relatively high level through a standard language that supports asynchronous constructs. ACK currently uses standard Verilog to model the design at the system level because it already supports many of the features required to efficiently describe asynchronous system and circuit operation. Concurrency in Verilog can be described at a coarse grained as well as a fine grained level through the use of modules, tasks, and fork-join statements. Fork-join statements are especially important as they are used to describe characteristic asynchronous behavior involving concurrency, sequencing, and choice in

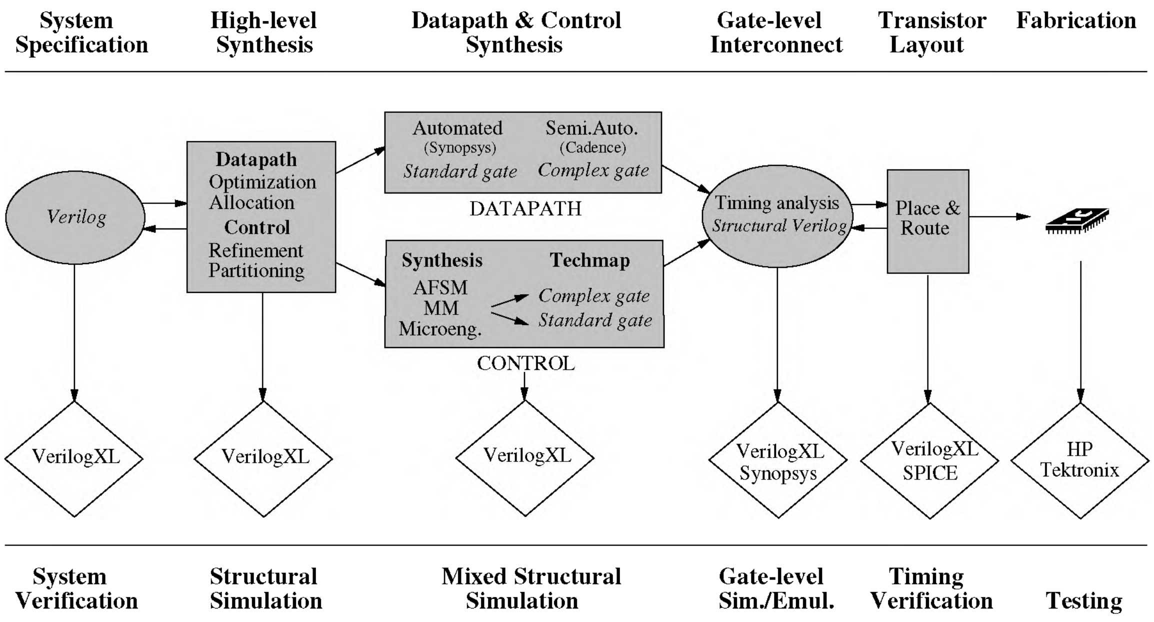

Figure 1. ACK tool flow

a succinct manner. Fork-join threads can be used to describe concurrent actions on individual signals (e.g., a signal burst in a burst-mode AFSM) as well as higher level expressions of concurrency. Verilog also supports the notion of events on signals which greatly simplifies description of asynchronous handshake protocols.

Standard Verilog does not, however, include the high-level concept of an asynchronous communication channel. While the behavior of channels can, of course, be described by the designer using explicit data and control signals, this is a common enough construct in asynchronous design that we have developed a standard channel package in Verilog. We are currently also looking into ways of supporting user defined datatypes in Verilog that are more complex than the array of bits currently supported.

Because high-level specifications are often quite general in nature, hints from the designer about which optimizations or control structures should be used can help the tool produce better circuits. As these hints have no effect on the high level behavioral simulation of the design, ACK supports such designer hints through the use of pragmas. Through these pragmas the designer can specify what type of control structure to use (AFSM, macromodule, or micro-engine), if chaining should be applied, and what parts of the code should be optimized for sequential or parallel execution. Interface timing is also currently specified through pragmas. The importance in using pragmas for hints to the synthesis tools is that different target implementation and circuit structures can be evaluated without having to alter a single statement of the behavioral specification.

3.3 Validation

Being able to simulate and verify a design from first specification to final layout is an essential part of any synthesis framework to ensure the correctness of the design. Targeting a single language for the description of the design through all phases of the synthesis process greatly simplifies validation. Using standard languages also has the benefit of mature validation tools being available. Since ACK targets standard Verilog as design specification language and also uses it to represent the intermediate structural forms of the design, testbenches can be reused at all levels of the synthesis process to ensure that the translations and optimizations of compiler and layout tools preserve important properties of the design. ACK leverages the Verilog-XL simulator from Cadence to gain confidence in the functional correctness of the design at the behavioral level. At the post-synthesis structural levels of the design, properties specific to asynchronous circuits, such as the implementation meeting bundled data and fundamental mode timing constraints, must be ensured. Back annotated switch-level timing simulation using Verilog-XL is well suited for these types of timing checks at the circuit level.

3.4 High-level Synthesis

While very small designs may be best optimized by the designer, opportunities for optimization quickly become hard to discover as the size of the design grows. As the specification grows in complexity the designer may have to express the design in an easily understandable fashion rather than what yields the best performance. Designs are therefore typically described with simplistic sequential computations to more easily gain confidence in their correctness. Manually translating these specifications to more efficient structures is an error prone and time consuming process. Automated optimizations play an important role in this process. Although these automated optimizations are important, the designer may still want to understand the changes that are made to the design. The designer may also want to perform additional optimizations by hand that may be difficult for a compiler to detect. In these situations feedback to the designer on what optimizations have been performed is essential.

ACK supports this feedback by back-annotating the result of all optimizations to the original specification. The designer can thus see directly the changes that have been made to the design and use this information to select compiler options to steer the optimization closer to what is desired. The ability to freeze parts of the specification so that no further optimizations are carried out on that section of code allows the designer to concentrate further optimizations without interfering with already finalized parts of the design. The ACK compiler supports many different optimization switches allowing the designer to closely guide the optimization procedure if so desired. ACK currently supports standard compiler optimizations such as loop-unrolling, sharing of common sub-expressions, and dead-code elimination, and is being extended with methods to automatically extract thread parallelism and detect chaining opportunities. Optimizations are further guided by pragmas in the specification language.

The current ACK optimizations are general in nature and can be used with advantage in both hardwired and programmable control. We recognize the need for optimizations targeting a specific control structure, e.g., AFSMs versus microengines, and are working on identifying what effect more specialized optimizations have on control overhead.

3.5 Control Synthesis

For a system level design framework it is important to provide flexible control that can be used efficiently in many different situations as one single control style seldom provides the best solution when large system designs are considered. ACK successfully blends several hardwired as well as programmable methodologies into new control structures suited for high-level design.

Hardwired control

ACK is biased towards using macromodules as the top-level control in the design hierarchy. This makes for a nicely transparent translation from specification program to circuit, and provides sufficient performance for most top level control and communication activities. The use of macromodules can be overridden by pragma hints from the designer where performance is critical. In the individual modules of the design hierarchy where performance is usually more important the control is implemented as burst-mode AFSMs to tailor the controller more closely to the specified behavior. Although small controllers of this type are very fast, their complexity grows quickly as the controller size increases. ACK provides a partition methodology [30] for AFSMs that divides large controllers into smaller interacting subcontrollers that can be implemented more efficiently.

To model short sequences of frequently used sequential computations even more efficiently than AFSMs, these sequential actions can be realized as chains [24]. The control for each computation chain is built out of highly specialized macromodule controllers that are specifically designed to provide a low forward latency. A request signal can thus cascade through a chain of such macromodules with very low control overhead. The macromodule control within a chain supports fork and join structures and also conditional execution to handle simple choices. More elaborate choice structures still reside in the AFSMs where they are more efficiently implemented. Our experiments show that chaining can deliver a performance increase of well over 20% for control dominated designs as compared to an AFSM-only solution.

In order to use a tool like ACK effectively to explore a design space that includes AFSM controllers, the synthesis step for these controllers (including handshake reshuffling, state assignment, logic minimization, etc.) must be as fast as possible to give the tool an interactive feel. In ACK we leverage existing tools wherever possible for this phase of the synthesis. The AFSM state assignment, for example, can be accomplished using a variety of burst-mode synthesis engines like Yun's 3D [44] or Nowick's Minimalist [19] system. With timing information provided by the designer and derived from the high-level synthesis, timed approaches like ATACS [34] can also be used. While macromodule synthesis is performed quickly, the complex logic minimization step of existing AFSM synthesis algorithms has been too slow to enable the interactive and iterative design exploration necessary to find good implementations of large controllers. We have therefore developed a new methodology for exact logic minimization of burst-mode AFSMs [27]. Using our new tool, even the largest burst-mode benchmarks available to date [35, 44] can be minimized exactly in less than one second, as compared to thousands of seconds for existing minimizers.

Programmable control

In addition to efficient hardwired control, it is important to provide the flexibility of programmable control at different levels of the system design. Many system designs are targeted to specific applications that, although specialized, require some flexibility in their run-time control. To be useful in this context, the programmable control must offer high performance and low overhead. Existing asynchronous general purpose microprocessors [22, 33, 39], and even asynchronous microcontrollers [23, 38] can be too coarse-grained with too much overhead for the task. Microprocessor cores also do not provide the flexibility in their control structure required to integrate them efficiently as a piece of a mixed-control style system design that also includes AFSM and macromodule sections.

We have investigated asynchronous microengines for ACK which offer both high performance and fine-grained programmability for domain-specific applications [26]. Our microengine architecture uses customized VLIW microinstructions that offer very fine-grained control over the datapath programmability resulting in compact microcode and high performance. The microengine allows implementations of domain specific applications that can directly compete with hardwired control in terms of performance (and area if ROM is used). By using standard macromodules for the local control of datapath units, the datapath is kept completely modular, and control is easily programmable and implementable in a standardized fashion. A major part of the microengine's high performance comes from its ability to dynamically schedule computation units in parallel and serial clusters, or chains, to best suit the current situation. Forming such serial clusters dynamically is very hard to do efficiently in synchronous microengines because the propagation delays of all computations must add up to an integral multiple of the clock period. In addition to offering high performance programmable control, the microengine architecture supports standard two and four phase handshake protocols with bundled data assumptions and is thus very easily integrated with other asynchronous components at any level in the design hierarchy.

3.6 Datapath Synthesis

At the moment ACK uses standard off-the-shelf synchronous-style datapath synthesis using Synopsys Design-Compiler combined with bundling delays. In order to make this as efficient as possible, considerable timing analysis is performed to overlap the datapath bundling delays with control delays whenever possible using what we call *delay borrowing*. Through delay borrowing the control overhead can in many cases be significantly reduced. Given the new ability of ACK to efficiently exploit thread level concurrency through chaining however, the option of more aggressive completion detecting or completion sensing data paths [36, 13] is an intriguing option that we intend to explore further.

Generalized C-element implementations with timing have been used effectively in data path portions of other high-performance asynchronous circuits [45, 40]. The same technology mapping options that we are exploiting for fast gC-based control circuits can be used for datapath synthesis. Allowing the specification of input timing and output constraint timing, as we do with control tech-mapping, fits well with datapath synthesis where some data may arrive before other data and some outputs must be generated earlier than others.

3.7 Circuit Implementation

Once the circuits have been synthesized, they must be realized into specific implementations. ACK performs a variety of low-level circuit optimizations at this point to get more performance out of a given high-level organization. Our implementation phase allows the controllers to be realized as custom complex CMOS gates [31, 46], and we are currently developing specific technology mapping techniques to optimize the performance of these gates at the transistor level. These optimizations will also become important in datapath synthesis.

One of the main advantages of the tech-mapping tool being developed by our research group is that it allows specification of input time separation and maximum timing bounds on output generation. These timings are derived automatically during the high-level and datapath synthesis steps of ACK. Input timing allows average case performance tech-mapping to be explored much more accurately than existing methods that are based only on probabilities [6, 28]. Output timing constraints allow setting a maximum bound on the time the controller is allowed to take to produce an output in response to an input. These output constraints are very useful in favoring a less frequently, but timing critical, operation over the average case. Such output constraints are essential for system level design where rigid timing bounds are often put on module interfaces, especially when interfacing to synchronous designs.

The circuits, once cast into realizable form, are assembled for fabrication using commercial place and route software. Until recently we used the Epoch tool from Cascade, but are in the process of switching to Cadence as the backend physical assembly engine.

4 Design Example: An Error Decoder for the CD-Player

To give a demonstration of a design implemented with ACK and the performance achievable using our proposed control structures, this section presents an error decoder for the CD-player [29] as a design example. This example is implemented both as a microengine and as hardwired control as generated by ACK. Although this is a fairly small example by system-description standards, it should be large enough to show some of the system design features in ACK and the potential for a system-level approach to synthesis, while being small enough to present enough detail to be interesting.

The error decoder circuit implements error-detection on the audio information recorded on Compact Discs using a syndrome computation algorithm. Figure 2 illustrates the

```
'include "channel.pkg"
module CD_Player_Error_Decoder (reset, start, tw, cw, sw, ew, lw);
  input    reset, start;
  input    tw;
  input    [7:0] cw;        reg  [31:0] syn;
  output   sw;              reg  [7:0] e, s;
  output   [7:0] ew;        reg  [5:0] n;
  output   [5:0] lw;        reg  t, stat;

  channel #(1)  S(sw);
  channel #(8)  E(ew);      channel #(1)  T(tw);
  channel #(6)  L(lw);      channel #(8)  C(cw);

  function [31:0] Horner;
    input  [7:0] s;
    input  [31:0] syn;
    begin
      Horner[7:0]   = GFadd(s, syn[7:0]);
      Horner[15:8]  = GFadd(s, Alpha(syn[15:8]));
      Horner[23:16] = GFadd(s, Alpha(Alpha(syn[23:16])));
      Horner[31:24] = GFadd(s, Alpha(Alpha(Alpha(syn[31:24]))));
    end
  endfunction

  function [7:0] GFadd;
    input  [7:0] s, syn;
    begin  GFadd = s ^ syn;  end
  endfunction

  function [7:0] Alpha;
    input  [7:0] syn;
    begin
      Alpha[7:5] = syn[6:4];   Alpha[1:0] = {syn[0],syn[7]};
      Alpha[4:2] = syn[3:1] ^ {syn[7],syn[7],syn[7]};
    end
  endfunction

  function [31:0] Shuffle;
    input  [31:0] syn;
    begin
      Shuffle[7:0] = syn[15:8];  Shuffle[15:8] = syn[31:24];
      Shuffle[23:16] = syn[7:0];  Shuffle[31:24] = syn[23:16];
    end
  endfunction

  always @(reset) begin
    if (reset == 0)  begin  T.reset; C.reset; S.reset; E.reset; L.reset;  end
    else begin
      @ start;
      forever begin
        fork
          begin  T.recv(t);  if (t == 0) n = 27;  else  n = 32;  end
          begin  syn = 0;  end
        join
        while  (n[5] != 1)  begin
          fork
            begin  n = n - 1;  end
            begin  C.recv(s);  syn = Horner(s,syn);  end
          join
        end
        fork
          begin  if  (t == 0) n = 27;  else  n = 32;  end
          begin  e = syn[7:0];  end
        join
        syn = Shuffle(syn);
        syn = Shuffle(syn);
        // pragma: PARTITION
        while  (~((n[5] == 1) || (syn[7:0] == syn[15:8])))  begin
          fork
            begin  n = n - 1;  end
            begin  syn = Horner(0,syn);  end
          join
        end
        stat = n[5];
        syn = Shuffle(syn);
        stat = (stat || (syn[7:0] == syn[15:8]));
        syn = Shuffle(syn);
        stat = (stat || (syn[7:0] == syn[15:8]));
        fork  S.send(stat); E.send(e); L.send(n);  join
      end
    end
  end
endmodule
```

Figure 2. Error decoder for the CD-player - specification in standard Verilog.

behavioral Verilog language specification of the error decoder design. Figures 3 and 4 show the structure of the hardwired and microengine implementations as derived by ACK. The error 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 error decoder can be found in [29].

The control structure of the hardwired implementation is based on burst-mode finite state machine synthesis. Since the control is too large to synthesize as a single controller with currently available synthesis tools, it must be partitioned (while our new logic minimizer [27] can easily handle such large controllers, currently available state assignment tools cannot). The designer specified partition statement is illustrated as a pragma in the Verilog code in Figure 2. The ACK framework then automatically partitions and refines the control logic into two sequentially executing burst-mode sub-controllers. To allow thread level concurrency using only burst-mode AFSM controllers, the fork-join statements must be split into separate controllers. The execution of these thread controllers is initiated by a handshake from the main controller partitions. Note the inability of an AFSM-only implementation to exploit efficient chaining for the fork-join threads. The datapath is implemented using standard gates with bundled control delays.

As an alternative to the hardwired implementation, the design is also implemented in the form of our programmable asynchronous microengine architecture [26]. In the microengine implementation, a microcode memory holds the VLIW micro instructions that control the execution of the datapath. The instruction bits directly control the operation mode of the RAS (Request, Acknowledge, and Sequencing) macromodule control blocks, as well as mux and operation mode settings of the datapath units. The threads of the fork-join statements in Figure 2 are implemented in the microengine as efficient computation chains. This exploitation of VLIW instructions and chaining allows rolling many actions into a single instruction and helps to significantly reduce control related overhead. The possible sequential chains are easily identified in Figure 4 by the horizontal arrows connecting the corresponding RAS blocks. The exact same datapath as for the hardwired implementation is also used for the microengine.

4.1 Result Comparison

The error decoder for the CD-player has been designed using ACK for a 3V, 0.6μ CMOS technology, and uses a single-rail bundled datapath and a four-phase handshake protocol. The direct comparison is with the same system designed using the Philips Tangram tool [29]. The Tangram circuit was, however, designed for a 5V, 1.2μ CMOS technology using a double-rail datapath, but did also use a four-

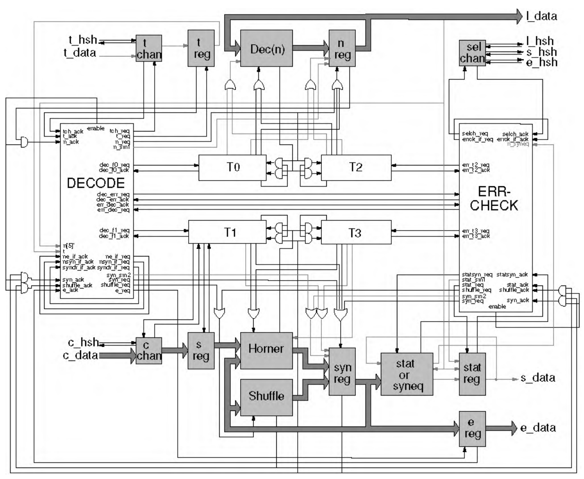

Figure 3. Error decoder for the CD-player - hardwired AFSM structure.

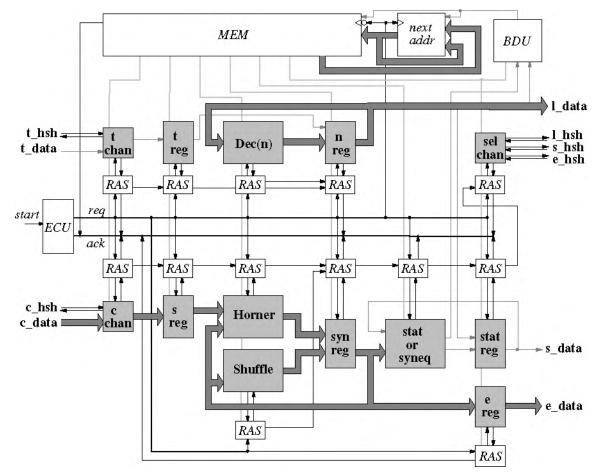

Figure 4. Error decoder for the CD-player - programmable microengine structure.

phase signaling protocol. The original Tangram circuit was reported to use a core area of 2.0mm^2, and have an approximate worst-case execution time of 20 μs per decoding sequence, where a decoding sequence decodes 32 8-bit words from the Compact Disc. According to van Berkel [8] when a similar, but more complex design for an error corrector for the DCC-player was scaled, a factor of 1.5 in performance improvement and a 40% smaller area was attributed to single-rail over double-rail. With feature size scaling under a constant field assumption [42], except for voltage, a single-rail Tangram implementation of the error decoder for the CD-player in the same 3V 0.6 micron technology that we targeted for the ACK-designed circuit could therefore be expected to have a decoding sequence time of about 5 μs and an area of 0.3 mm^2.

Table 1 shows the performance of the scaled Tangram single-rail implementation of the error decoder for the CD-player along with the performance of the same error decoder as generated by ACK. The ACK circuit's performance numbers were obtained through post-layout SPICE simulation using worst case transistor models and temperature for two different implementations: one with hardwired AFSM control and no chaining, and one with programmable microengine control and thread-level chaining used to improve performance. Both ACK-generated circuits used the same datapath. As can be seen in the table, the ACK circuit with hardwired control had a worst case decoding sequence time of 1.58 μs with an area of 0.25 mm^2. The ACK circuit with microengine control had a decoding sequence time of 1.46 μs and an area of 0.20 mm^2 when a ROM-based memory was used, and an area of 0.46 mm^2 when RAM was used. The speedup figure shown in the table is for speedup of the microengine version of the circuit compared to the hardwired version.

Although the Philips design was targeted for low-power, both the Phillips and ACK circuits used the same degree of parallelism in the specification. The 3X performance advantage of the ACK circuit probably cannot be explained solely because of the low-power emphasis of the Tangram circuit. We believe that a significant portion of the performance difference can be attributed to the efficient implementations of partitioned burst-mode controllers and chained computations that ACK allows. As illustrated by the performance numbers of the microengine implementation, our approach to programmable control can also compete directly with hardwired control with respect to performance. A large part of the microengine's power comes from its ability to chain computations with very little control overhead. To achieve good performance it is therefore desirable to have designs which allow long chains to be formed.

Two other ACK-generated circuits are represented in Table 1: an implementation of Beerel and Yun's differential equation solver [45], and a barcode reader circuit [37]. These examples represent different levels of opportunity to exploit thread-level chaining. When chaining is a possibility, as in the error decoder for the CD-player and differential equation solver, the chained microengine version has a performance edge. When the opportunities for chaining are not as pronounced, as in the barcode reader, the hardwired control is faster. Our initial gate-level simulations indicate that ACKs hardwired approach could gain about 10-20% in performance if chained control is used[1]. As illustrated in the Table, hardwired and programmable control have their strengths and weaknesses, both in terms of area and performance, and it is important for a high-level synthesis framework to support both.

It should be noted that both types of design (hardwired

[1]Due to recent infrastructure changes (the Epoch layout tool is no longer available to us) we have not yet been able to layout a hardwired version of the error decoder for the CD-player that exploits chaining.

Design	Hardwired		Microengine		
	Time(μs)	Area(mm^2)	Time(μs)	Arom/ram(mm^2)	Speedup
Scaled Tangram CD-player error decoder	5	0.3	—	—	—
ACK CD-player error decoder	1.58	0.25	1.46	0.20/0.46	+8.0%
ACK Diff-eqn solver	1.75	0.26	1.69	0.28/0.47	+3.5%
ACK Barcode reader	3.30	0.10	3.61	0.10/0.25	−9.5%

Table 1. Design comparisons of hardwired vs. microengine implementations.

and microengine) were implemented in ACK without using any explicit timing based optimizations. Better results are to be expected for both types of designs when timing optimizations are applied to hide control overhead. The ACK designs were synthesized to a gate-level representation with bundled data delays obtained via process corners gate-level timing analysis using Synopsys Design-Analyzer tool. This timing analysis is, in our experience, very accurate allowing the use of relatively small safety margins. The gate-level circuits were placed and routed using the Cascade Epoch tool, which was also used to generate post-layout area numbers and SPICE models.

5 Conclusions

We have described our views on how high-level system design should be approached, and presented ACK as our current research vehicle to explore these ideas. We feel that system design has a different set of challenges than small controller design and requires a different set of tools. Specifically, the ability to chain computations by using highly optimized macromodules, together with specialized AFSM control, and the possibility of high-level programmable control seems to be well suited to the demands of system design. Combined with the standard procedural-level Verilog HDL as input language this approach is a powerful method for designing systems that are too large to be expressed in signal transition graphs, but too small or specialized to benefit from general microprocessor implementation. Our initial results are quite encouraging and we plan to continue to expand the abilities of the ACK tool to become even more useful at the system level.

Acknowledgements. Our thanks go to Al Davis for helpful discussions, and to Junglin Yang, Eric Kuehne, Johan Nyblom, Samuel Larsen, Robert Thacker, and Eric Peskin for helping with the development of the ACK framework.

References

[1] V. Akella and G. Gopalakrishnan. Shilpa: A high-level synthesis system for self-timed circuits. In *ICCAD*, pages 587–594, Nov. 1992.

[2] T. Anderson, D. Culler, and D. Patterson. A case for networks of workstations: NOW. *IEEE Micro*, February 1995.

[3] W. Athas and C. Seitz. Multicomputers: Message passing concurrent computers. *IEEE Computer*, 21(8), August 1988.

[4] A. Bardsley. Balsa, a case study of a DMA controller. In M. Josephs and A. Yakovlev, editors, *ACiD Working Group Workshop*, January 1999.

[5] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *CHDL-97*, pages 89–91, Apr. 1997.

[6] P. A. Beerel, W.-C. Chou, and K. Y. Yun. A heuristic covering technique for optimizing average-case delay in the technology mapping of asynchronous burst-mode circuits. In *EURO-DAC*, Sept. 1996.

[7] P. A. Beerel, C. J. Myers, and T. H.-Y. Meng. Covering conditions and algorithms for the synthesis of speed-independent circuits. *IEEE Trans. on CAD*, Mar. 1998.

[8] K. v. Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schalij, and R. van de Wiel. A single-rail re-implementation of a DCC error detector using a generic standard-cell library. In *Asynchronous Design Methodologies*, pages 72–79. IEEE Computer Society Press, May 1995.

[9] K. v. Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *EDAC*, pages 384–389, 1991.

[10] E. Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[11] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *ICCAD*, pages 262–265. IEEE Computer Society Press, Nov. 1989.

[12] S. M. Burns and A. J. Martin. Synthesis of self-timed circuits by program transformation. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 99–116. Elsevier Science Publishers, 1988.

[13] W. Chou, P. A. Beerel, R. Ginosar, R. Kol, C. J. Myers, S. Rotem, K. Stevens, and K. Y. Yun. Average-case optimized technology mapping of one-hot domino circuits. In *ASYNC98*, pages 80–91, 1998.

[14] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[15] W. A. Clark. Macromodular computer systems. In *Spring Joint Computer Conference*. AFIPS, April 1967.

[16] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, Nov. 1996.

[17] W. J. Dally and P. Song. Design of a self-timed VLSI multicomputer communication controller. In *ICCD*, pages 230–234, 1987.

[18] R. Felderman, A. DeSchon, D. Cohen, and G. Finn. Atomic: A high speed local communication architecture. *Journal of High Speed Networks*, 3(1), 1994.

[19] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist, an environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report CUCS-020-99, Columbia University, Computer Science Dept., July 1999.

[20] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In *COMPCON*, pages 476–485, Mar. 1994.

[21] S. B. Furber, J. D. Garside, and D. A. Gilbert. AMULET3: A high-performance self-timed ARM microprocessor. In *ICCD*, Oct. 1998.

[22] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, Feb. 1999.

[23] H. v. Gageldonk, D. Baumann, K. van Berkel, D. Gloor, A. Peeters, and G. Stegmann. An asynchronous low-power 80c51 microcontroller. In *ASYNC98*, pages 96–107, 1998.

[24] D. Gajski. *Principles of Digital Design*. Prentice Hall, 1997.

[25] G. Gopalakrishnan, P. Kudva, and E. Brunvand. Peephole optimization of asynchronous macromodule networks. *IEEE Trans. on VLSI*, 7(1):30–37, Mar. 1999.

[26] H. Jacobson and G. Gopalakrishnan. Application-specific programmable control for high-performance asynchronous circuits. *Proceedings of the IEEE*, 87(2):319–331, Feb. 1999.

[27] H. Jacobson, C. Myers, and G. Gopalakrishnan. Fast and exact logic minimization for extended burst-mode controllers. Technical Report UUCS-99-012, Department of Computer Science, University of Utah, U.S.A., 1999.

[28] K. W. James and K. Y. Yun. Average-case optimized transistor-level technology mapping of extended burst-mode circuits. In *ASYNC98*, pages 70–79, 1998.

[29] J. Kessels, K. van Berkel, R. Burgess, M. Roncken, and F. Schalij. An error decoder for the compact disc player as an example of VLSI programming. Technical report, Philips Research Laboratories, Eindhoven, The Netherlands, 1992.

[30] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[31] P. Kudva, G. Gopalakrishnan, H. Jacobson, and S. M. Nowick. Synthesis of hazard-free customized CMOS complex-gate networks under multiple-input changes. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[32] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[33] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. Penzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, Sept. 1997.

[34] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis and verification of gate-level timed circuits. Technical Report CSL-TR-94-652, Stanford University, Jan. 1995.

[35] S. M. Nowick, M. E. Dean, D. L. Dill, and M. Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. *Integration, the VLSI journal*, 15(3):241–262, Oct. 1993.

[36] S. M. Nowick, K. Y. Yun, and P. A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *ASYNC97*, pages 210–223. IEEE Computer Society Press, Apr. 1997.

[37] P. R. Panda and N. Dutt. 1995 high level synthesis design repository. Technical Report 95-04, University of California, Irvine, U.S.A., 1995.

[38] M. Renaudin, P. Vivet, and F. Robin. ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor. In *ASYNC98*, pages 22–31, 1998.

[39] W. F. Richardson and E. Brunvand. Architectural considerations for a self-timed decoupled processor. *IEE Proceedings, Computers and Digital Techniques*, 143(5):251–257, Sept. 1996.

[40] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev. RAPPID: An asynchronous instruction length decoder. In *ASYNC99*, pages 60–70, Apr. 1999.

[41] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[42] N. H. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison Wesley, 1992.

[43] C. Ykman-Couvreur, B. Lin, and H. de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, Sept. 1994. User and Tutorial manual.

[44] K. Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems*. PhD thesis, Stanford University, Aug. 1994.

[45] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Trans. on VLSI*, 6(4):643–655, Dec. 1998.

[46] K. Y. Yun and D. L. Dill. Automatic synthesis of extended burst-mode circuits: Part I (specification and hazard-free implementation). *IEEE Trans. on CAD*, 18(2):101–117, Feb. 1999.