BitSNAP: Dynamic Significance Compression For a Low-Energy Sensor Network Asynchronous Processor

Virantha N. Ekanayake, Clinton Kelly, IV, and Rajit Manohar
Computer Systems Laboratory
Electrical and Computer Engineering
Cornell University
Ithaca, NY 14853, U.S.A
E-mail: {viran, clint, rajit}@csl.cornell.edu

Abstract

We present a novel asynchronous processor architecture called BitSNAP that utilizes bit-serial datapaths with dynamic significance compression to yield extremely low-energy consumption. Based on the Sensor Network Asynchronous Processor (SNAP) ISA, BitSNAP can reduce datapath energy consumption by 50% over a comparable parallel-word processor, while still providing performance suited for powering low-energy sensor network nodes. In 180nm CMOS, the processor is expected to run at between 6 and 54 MIPS while consuming 152pJ/ins at 1.8V and just 17pJ/ins at 0.6V.

1 Introduction

In the energy-limited domain of untethered battery-powered sensor-network nodes, an energy efficient processing unit is essential to provide a lifetime on the order of months or years. Most commodity-off-the-shelf (COTS) microcontroller based sensor-network platforms are very energy hungry, and have documented lifespans on the order of just hours or days.

We have recently proposed a novel asynchronous processor ISA designed expressly for sensor-networks called the SNAP ISA, for Sensor Network Asynchronous Processor. SNAP was designed for two purposes: (1) To serve as the processor in a custom chip multi-processor for a hardware sensor-network simulator called Network-on-a-Chip (NoC) [1], and (2) to serve as the main processor in a low-power wireless sensor-network node called SNAP/LE (SNAP Low Energy) [2]. The processor variants based on the SNAP architecture/ISA can be seen in Fig. 1.

The SNAP/LE processor contains a 16-bit event-driven RISC core and can operate over a wide voltage range, executing instructions at a rate between 23MIPS (at 0.6V) to 200MIPS (at 1.8V) while consuming as little as 24pJ per instruction execution, which is orders of magnitude less energy than any commodity microcontroller. SNAP/LE leverages a simple ISA, event-driven execution model, custom hardware support for sensor-network nodes, and low-energy asynchronous circuit techniques to achieve extremely low dynamic power consumption at relatively high throughput. At the extremely low activity levels seen in target applications (10 events a second or less), SNAP/LE has a power consumption on the order of 16 to 58nW at 0.6V.

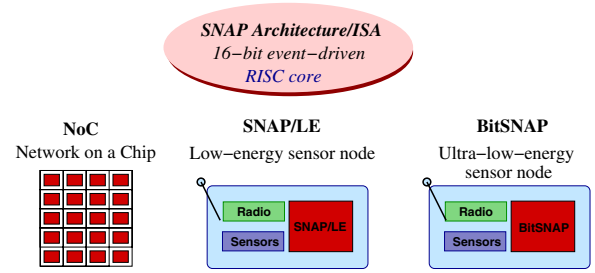

Figure 1. SNAP processor family

One key observation from current sensor-network testbeds is the relatively low performance requirements – sensor nodes mainly extract data from sensors, perform some rudimentary processing and aggregation, and forward the results to other more powerful base stations across slow (kilobits per second) radio links [3]. Most currently available sensor-network hardware have processors that operate between 4 and 12MIPS [3, 4]. Even at the limits of voltage-scaling, and using a supply voltage close to the transistor thresholds, SNAP/LE still runs at over 20MIPS, which is far faster than we require.

In this paper, we propose a new processor called BitSNAP that trades-off some of the extra performance for

Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)
1522-8681/05 $20.00 © 2005 **IEEE**

even greater power savings. BitSNAP is a logical extension of SNAP/LE, designed to leverage the *application behavior* typical of sensor-networks. We exploit the natural compressibility of operand values propagating through the processor's datapath in order to dynamically reduce the amount of switching. In addition, while SNAP/LE was designed with a 16-bit parallel datapath, BitSNAP uses a bit-serial datapath, for which asynchronous design provides an ideal match for computing on data dependent operand lengths efficiently. Using a bit-serial datapath also reduces the amount of circuit hardware required, translating to a reduction in static leakage power that is a growing concern in deep submicron processes.

This paper is organized as follows: We first define dynamic significance compression in the context of processor datapaths, and explore the benefits of utilizing it for a bit-serial processor. A discussion of BitSNAP's general architecture is then presented, along with a detailed evaluation of the novel hardware used to support significance compression. We finally evaluate BitSNAP with a representative sensor-network software workload and evaluate the energy savings and performance impact of the bit-serial compressed datapath. We conclude with comparisons against existing and proposed asynchronous designs, and a discussion of related work.

2 Significance Compression in Processors

2.1 Parallel-word architectures

Previous architectural studies [5, 6] have shown that the majority of datapath operand values require fewer bits of significance than a full 16-, 32- or 64-bit wide datapath provides. By dynamically compressing the significance of data values (suppressing leading 0s and 1s), and transmitting only the required bits, 30 to 80% of integer datapath switching energy can be saved.

Prior work by Brooks and Martonosi in the synchronous community has looked at operand-based clock gating of functional units [7]; each cycle, the operands are checked for leading zeros and the appropriate upper segment of the functional unit is turned off. Naturally, this has a latency and energy impact for the zero-detect circuit. Canal et al. in [5] presented an approach where every data operand is tagged with extra bits specifying which bytes in the data word are compressible. The byte-parallel and byte-serial datapaths studied in their paper are significantly impacted by the latching overhead for vertical control propagation and bypass mechanisms for multi-cycle byte-serial processing.

In contrast, past work on dynamic significance compression in the context of asynchronous processors has focused on width-adaptive data (WAD) word architectures where

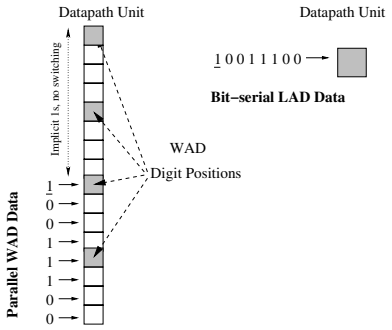

Figure 2. Width adaptive datapath(WAD) (left) of 16-bits, with one WAD digit every 4-bits. Length adaptive datapath(LAD) (right) of 16-bits, with every bit a LAD digit.

the number representation itself is self-delimiting. WAD numbers as proposed by Manohar [6] use a special number representation consisting of the digits $\{0,1,\underline{0},\underline{1}\}$. WAD digits $\underline{0}$ and $\underline{1}$ represent *delimiters* that terminate a given binary word – any bits more significant than the delimiter bit are assumed to have the same value as the delimiter. For instance, $\underline{1}011$ for an 8-bit wide datapath represents 1111 1011 in normal binary representation. This allows us to *compress* a number with leading ones and zeroes into a more compact representation, and easily lends itself to vertically pipelined control propagation in the datapath.

For reasons of efficiency in practical architectures, it becomes necessary to limit the use of WAD digits on a per block basis, shown in Fig. 2, as opposed to having all 16 bits of a datapath represented by WAD digits. Such a hybrid scheme, where some bits are normal binary digits and others are special WAD digits, was used in the QDI 32-bit register files presented in [8]. Each block consisted of 3 normal binary bits and 1 WAD digit at the most significant position, and thus allowed compression of data values on the granularity of 4-bit blocks. On the flip side, using a hybrid scheme can complicate processing of the data, because now the designer needs to accommodate both normal binary digits, as well as the WAD digits.

2.2 Bit-serial architectures

In contrast to parallel-word architectures, BitSNAP uses dynamic significance compression on a bit-serial data stream. In other words, BitSNAP is a *length-adaptive datapath* (LAD) processor as opposed to a width-adaptive datapath processor, and performs significance compression on the granularity of a single bit, as shown in Fig. 2. Instead of sending 16 bits through the datapath for each word, only the bits up to and including the delimiter bit need to be sent. An obvious advantage over conventional bit-serial datapaths, aside from the reduced switching, is that the through-

Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)
1522-8681/05 $20.00 © 2005 **IEEE**

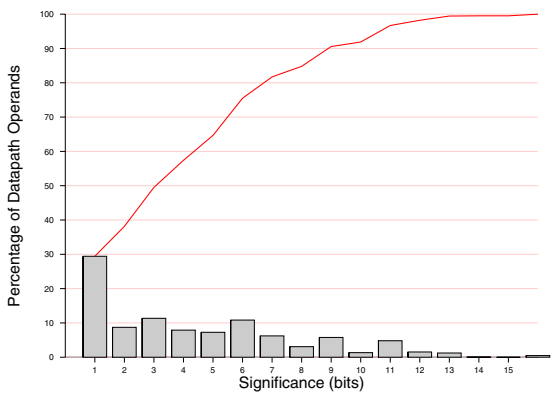

Figure 3. Operand significance averaged across representative sensor network workload (with cumulative totals)

put for operations on data words is no longer strictly linear in the architectural width (16 times the individual bit cycle time for a 16-bit architectural width), because compressed operands can complete in time proportional to the number of significant bits. A further advantage for the designer is that because every digit is in LAD representation, the units on the datapath do not have to deal with both normal binary bits and adaptive digits.

Synchronous bit-serial architectures have been widely explored, from bit-serial signal processing [9] to design compilers such as GE's Parsifal [10]. However, even simple bit-serial architectures require multiple clocks – one for the actual bit-computation unit and another to signal word boundaries. In the case of dynamic significance compression, where a word boundary could potentially occur at *any* place, the clocking and bypass scheme would quickly become unmanageable even at word sizes of 16-bits. This could be handled by processing a data end-of-word bit per cycle at additional cost in energy.

Table 1. BitSNAP Sensor Network Software Workload

Handler	Function
Packet TX	Radio packet transmission via 802.11 based MAC layer
Packet RX	Radio packet reception
AODV RR	Ad-hoc routing protocol route lookup and reply
AODV PF	Ad-hoc routing protocol packet forward
TR1000 Radio Stack	CRC and data packet encoding
Data logger	Sensor logging and running average computation

As an example of the potential energy savings inherent in processor workloads, especially in the context of BitSNAP's sensor network processor, we performed an architectural study of processor datapath operand significance (number of bits that remain after compressing leading 0s or 1s) in a typical sensor network workload. The workload, an extension to that presented in [2], is summarized in Table 1 (more details on each individual benchmark can be found in Section 6), and Fig. 3 shows the distribution of operand significance averaged across these workloads. Notice that 90% of all the data values flowing through the processor's datapath have significance less than 10 bits, and about 30% can be compressed to just 1 bit.

2.3 Length-Adaptive-Data Representation

We considered two possible options for encoding the LAD digits in our delay-insensitive pipelines:

1. For each bit, use two 1-of-2 codes, with one specifying the data, and the other marking whether this bit is a delimiter or not. Thus 00 would communicate a non-terminating 0, while 10 would communicate a $\underline{0}$ (repeating 0s beyond the current position).

2. For each bit, communicate one 1-of-4 signal, with rails 0, 1, 2, and 3 respectively communicating the WAD values 0, 1, $\underline{0}$, $\underline{1}$

Although the first is a good option for hybrid WAD datapaths such as those used in the WAD register-files, because it keeps the data information distinct from the delimiter information, we prefer the second option for LAD bit-serial datapaths. Because only one wire switches when transmitting a bit, there is no energy overhead in sending LAD data.

3 BitSNAP Architecture

In this section we provide a summary of the SNAP ISA, as well as the specific architectural decisions made in implementing the bit-serial LAD nature of BitSNAP.

3.1 Core Architecture

A high-level block diagram of the BitSNAP processor is shown in Fig. 4. BitSNAP implements an in-order, single-issue core with no speculation. The SNAP ISA is based on the MIPS design, and specifies an *architectural* word width of 16-bits with 16 registers. Instructions can be either one or two words (for instructions requiring a 16-bit immediate value). Instruction and data memory consists of two 4KB banks, with a user-writable instruction memory to allow boot-strapping of the processor from externally supplied program code at power-up.

Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)
1522-8681/05 $20.00 © 2005 **IEEE**

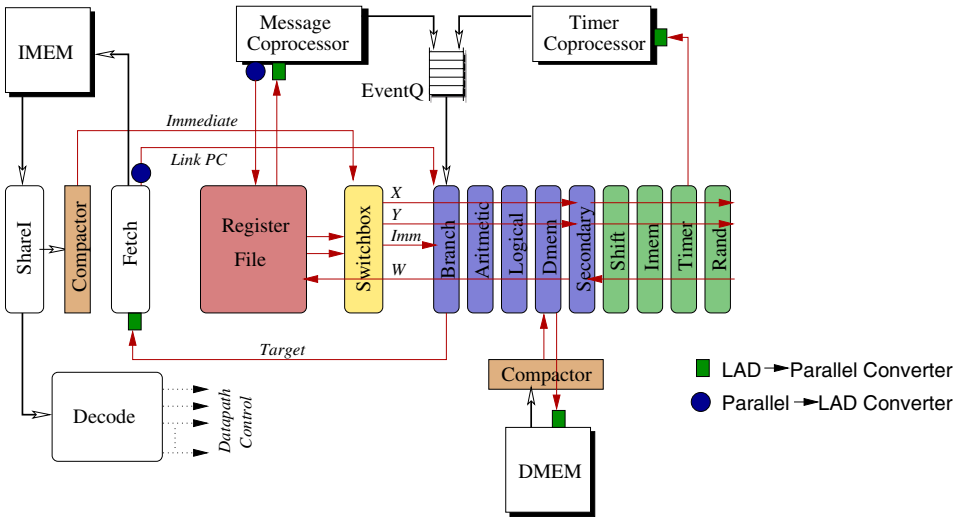

Figure 4. BitSNAP Block Diagram

The key difference between the SNAP architecture and conventional processors is that it is completely event-driven – SNAP remains idle (no dynamic switching) until either an external event (sensor reading, incoming radio message) or internal timer expiration event occurs. At this point, a token corresponding to the event will propagate through a hardware *event queue* and awaken the processor through the *PC Control* unit. In this unit, the event token is used to index into a *hardware* event table, returning the address of a specific software event handler that then gets passed on to the *fetch*, at which point the processor starts executing code. Once the processing completes, the software event handler issues a `DONE` instruction which halts the processor until another event appears at the head of the event queue.

Radio events are generated by the *message coprocessor* which communicates with an external radio chip. The message coprocessor also generates events corresponding to sensor interrupts from external sensors. All communication with the message coprocessor to the radio and sensors is handled through general-purpose register-mapped I/O, and thus require no new instructions. For more details on the SNAP/ISA and coprocessor implementation, the reader is referred to [2].

BitSNAP uses a two level bus structure to leverage the flexibility asynchronous design provides. We have a primary bus for the units on the datapath that we expect to be used more often: the arithmetic unit, logical unit, branch unit, and data memory unit. A secondary bus is then attached to the primary bus as an additional unit; this bus contains the shift unit, instruction memory access unit, timer coprocessor access unit, and pseudo-random number generator. In designing BitSNAP, we use QDI techniques that are a hybrid of the design styles used in the MiniMIPS [11] and Caltech microprocessor [12]. Whereas the Caltech microprocessor used aligned-datapaths designed with control/data decomposition, and the MiniMIPS used fine-grained pipeline stages with pipelined completion, BitSNAP varies the use of both styles based on the actual hardware block being implemented.

3.2 Bit-Serial Support

The architecture discussed so far shares much in common with SNAP/LE, our low-power sensor network node processor. Now, we present the novel features and modifications added to the architecture to enable dynamic significance compression and the conversion from bit-parallel data paths to bit-serial.

The shaded blocks in Fig. 4 represent the parts of the datapath that process bit-serial data. Notice that the entire register-file, all the functional units, and every datapath bus split and merge are implemented as bit-serial units operating on LAD digits. The thin wires represent bit-serial buses which transfer LAD data. Because we are concentrating on the main processor core design, we will be using the same bit-parallel timer and message coprocessors from SNAP/LE. However, it is certainly conceivable to see these implemented using LAD data paths in future research.

The memories still operate on 16-bit parallel words for reasons of density and efficiency. The fetch unit, because it operates every cycle and interfaces directly with the instruction memory, is also implemented using bit-parallel circuits for efficiency reasons. Otherwise, the program counter (PC) would go through a bit-serial to parallel conversion every cycle, wasting energy for the common case where the PC just increments every cycle, and does not interact

directly with the datapath. Similarly, the *ShareI* process which shares the incoming word with either the decode (in the case of an instruction word) or datapath (for immediate operands) uses bit-parallel circuits.

4 Computation on Length-Adaptive-Data

In this section we present some detailed descriptions of representative functional and bus units in BitSNAP. We will describe asynchronous circuits with CHP (Communicating Hardware Processes), whose syntax is described in the Appendix. For conciseness in expressing operations on LAD digits, we introduce some added notation: (1) The *delimiter* operator $\hat{x}$ returns true if the LAD digit x equals a delimiter value ($\underline{0}$ or $\underline{1}$), and (2) the *demote* operator $\check{x}$ returns the demoted value of x, converting from the set $\{\underline{0},\underline{1}\}$ to $\{0,1\}$.

4.1 Bus Interface

A crucial set of circuits are the bus interfaces connecting the register file ports to the functional units. These interface units are quite different from their bit-parallel counterparts, because of the need to hold the bus for multiple bus read/write cycles for a single control token. We specify a bus reader that waits for a probe on the control channel *C* before reading data from the shared bus *B* and writing to the "private" channel *D* as follows:

$$
\begin{array}{l}
*[[\overline{C} \longrightarrow \;\; B?\,d;\;\; D!\,d; \\
\qquad\quad [\hat{d} \longrightarrow \;\; C? \\
\qquad\quad [] \textbf{else} \longrightarrow \;\; \textbf{skip} \\
\qquad\quad] \\
\;\;]]
\end{array}
$$

Once the delimiter bit arrives, the bus interface completes the communication on *C* and allows the next control token to appear. The handshake for this process was selected to minimize the amount of circuitry required (thereby minimizing the amount of load on the shared bus).

$$
\begin{array}{l}
*[\;\; [D^e]; [C^d]; en\uparrow; [\neg\hat{B} \longrightarrow D^d\uparrow [] \hat{B} \longrightarrow D^d\uparrow; C^e\downarrow]; \\
\qquad B^e\downarrow; [\neg D^e]; en\downarrow; (D^d\downarrow, ([\neg B]; ([\neg C^d]; C^e\uparrow), B^e\uparrow))]
\end{array}
$$

Note that this handshake does not provide any early feedback on *C* because it does not lower the enable C^e until *D* has output the entire LAD word. We use slack on the input of the process generating the bus control token *C* to decouple this reader from the decode. The bus writer process uses a similar formulation.

4.2 Alignment Process

The very nature of LAD operands implies that the values on the X and Y buses in Fig. 4 may have varying lengths.

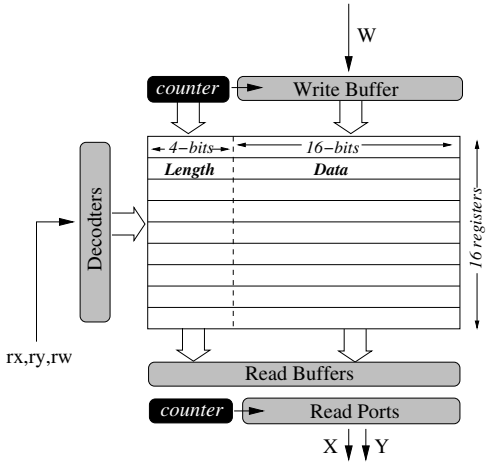

Figure 5. BitSNAP register-file

However, functions that use multiple input buses (such as the arithmetic unit) can be implemented more easily if the inputs are of the same length [6]. To address this issue we use an alignment process that pads the shorter of the two inputs to the same length as the longer one. For instance, adding $\underline{0}1$ and $\underline{1}011$ causes the first operand to be expanded to $\underline{0}001$, taking advantage of the redundancy inherent in LAD numbers. A CHP process to do the alignment is shown below:

$$
\begin{array}{l}
*[\;\; a := \overline{A},\;\; b := \overline{B}; \\
\qquad [\;\; \hat{a} \wedge \neg\hat{b} \longrightarrow \;\; X!\check{a}, Y!b;\;\; B? \\
\qquad [] \neg\hat{a} \wedge \hat{b} \;\; \longrightarrow \;\; X!a,\;\; Y!\check{b};\;\; A? \\
\qquad [] \;\hat{a} \wedge \hat{b} \;\; \longrightarrow \;\; X!(A?),\;\; Y!(B?) \\
\qquad [] \neg\hat{a} \wedge \neg\hat{b} \longrightarrow \;\; X!(A?),\;\; Y!(B?) \\
\;\;]]
\end{array}
$$

A and *B* are the unaligned input channels, while *X* and *Y* are the respective aligned outputs. Once a delimiter arrives on, for example, channel *A*, we postpone the communication on *A* until the delimiter arrives *B*. Until it does, the channel *X* outputs a non-delimited version of the value on *A* for every value that arrives on *B*.

4.3 Register File

A LAD register-file must be able to store dynamically compressed result operands exactly, without discarding the significance information. Furthermore, although datapath values may have any length from 1 to 16, the register file must be able to hold the full 16-bits of significant bits. We accomplish this by having both the normally required 16x16-bit register array, and an extra 4-bits of storage per register to record the length of the data (which also represents the location of the delimiter bit), as shown in Fig. 5.

Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)
1522-8681/05 $20.00 © 2005 **IEEE**

Writes are handled as follows: as the bit-serial data arrives at the register file on *W*, it gets stored in a write buffer. A 4-bit counter keeps count of how many bits have arrived. Once the delimiter marking the end of a word arrives, the write buffer is written to the register file, along with the value of the counter which specifies the number of significant bits.

During **reads** the 16-bit register is output to a read buffer, and the 4-bit value is read into a set of latches. A counter is then incremented while bits are shifted out of the read buffer into the register read bus, until the counter value equals the read length register.

BitSNAP's register file supports two bit-serial read ports and one bit-serial write port. Pipelined register locking is enabled, allowing non-conflicting reads and writes to progress concurrently, and allowing a following instruction to proceed while the previous is still writing back a (non-conflicting) register. However, the register file does not support direct bypass of the contents of the write buffer to the read ports; we found that the added concurrency was limited across our benchmarks, and not worth the extra complexity and energy costs of implementing a bypass mechanism. The actual register cell and word-line decoders are the same as those found in standard QDI-register files, with the exception of some extra external control to synchronize the counters.

4.4 Logical Unit

The logical unit, which computes bitwise AND, OR, NOR, and XOR, is our first example of computing on LAD digits. We assume the inputs are aligned using the previously mentioned alignment process. A bitwise logical operation on non-delimiter digits results in the same value as in normal binary operations. The extensions to support the case when both inputs are delimiter digits are shown in Fig 6. The CHP for our logical unit, which we implement using a standard PCEVFB buffer stage [8] with conditional input acknowledge on channel *F* looks like the following:

$$
\begin{array}{l}
*[\ f := \overline{F},\ \ A?a,\ \ B?b; \\
\quad [f = and \longrightarrow\ \ C!AND(a,b) \\
\quad [] f = or \longrightarrow\ \ C!OR(a,b) \\
\quad [] f = nor \longrightarrow\ \ C!NOR(a,b) \\
\quad [] f = xor \longrightarrow\ \ C!XOR(a,b) \\
\quad]; \\
\quad [\hat{a} \longrightarrow\ \ F?\ [] \textbf{ else } \longrightarrow\ \ \textbf{skip}] \\
\ ]
\end{array}
$$

4.5 Arithmetic Unit

The arithmetic unit handles normal addition/subtraction, as well as addition/subtraction with carry to allow synthesis of software 32-bit addition. In order to simplify the circuits, the arithmetic unit expects aligned inputs.

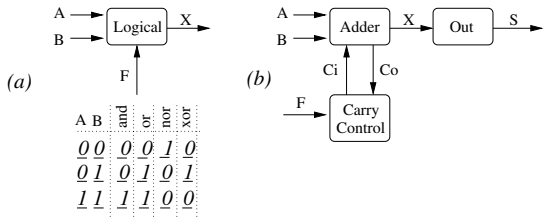

Figure 6. (a) Logical unit (b) Arithmetic unit

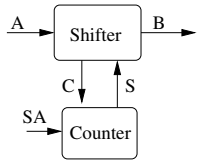

Figure 7. Shift unit with counter

The function logic remains the same as that for a binary full-adder for non-delimiter bits. Table 2 shows the addition for the delimiter bits. The slightly more complicated cases occur in rows 2 and 5, where the delimiter bit "overflows" into the next higher bit.

Table 2. Truth table for addition

a	*b*	*cin*	*sum*	*cout*
0	0	0	0	0
0	0	1	01	0
0	1	0	1	0
0	1	1	0	1
1	1	0	10	1
1	1	1	1	1

We use three processes to implement the adder as shown in Fig. 6b. The *Adder* does the actual logic, while the *Carry Control* block generates the carry-in for the adder, based on the last carry and operation to be performed (add/sub on *F*). The *Out* process handles the special cases in the delimiter table by outputting the two consecutive values on *S*.

4.6 Shifter

In parallel-word architectures, shifting can usually be accomplished by a multi-stage logarithmic shifter, which relies on hard-wired shift stages. In contrast, with bit-serial data, shifting requires a more active approach, either dynamically *consuming* input values in the case of a right shift, or inserting 0s in the case of a left shift.

Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)
1522-8681/05 $20.00 © 2005 **IEEE**

The shifter consists of a function block and a counter which stores the shift amount (Fig. 7). The counter can receive two commands on channel *C*: (1) *LOAD* to read in the shift amount on *SA*, and (2) *DEC* to decrement the counter. On every command, the value of the counter is compared to zero and the result sent to the shift unit on *S*. The data comes in from the bus on channel *A* and the shifted value is output on *B*.

We will discuss implementing the right shift, the more complex case because of the need to ensure that the delimiter bit does not get shifted out. The following CHP for an arithmetic shift right (*C* is assumed to reset with a *LOAD* token):

$$
\begin{array}{l}
SRA \equiv \\
\quad *[\ (s := \overline{S});\ \ A?a; \\
\quad\quad [s = SHIFT \longrightarrow \\
\quad\quad\quad\quad [\hat{a} \longrightarrow\ \ B!a,\ \ S?\ \ , C!LOAD \\
\quad\quad\quad\quad [] \neg\hat{a} \longrightarrow\ \ S?,\ \ C!DEC \\
\quad\quad\quad\quad] \\
\quad\quad [] s = STOP \longrightarrow \\
\quad\quad\quad\quad [\hat{a} \longrightarrow\ \ B!a,\ \ S?,\ \ C!LOAD \\
\quad\quad\quad\quad [] \neg\hat{a} \longrightarrow\ \ B!a \\
\quad\quad\quad\quad] \\
\quad]]
\end{array}
$$

Notice that as soon as the delimiter on the input is encountered, the output can be completed by copying it to the output, and terminating the shift (regardless of how many bits were actually shifted).

The operation for a left shift is similar, except instead of consuming the input bits, we insert zeros corresponding to the shift amount. Naturally, this may generate a number that is more than 16-bits long; we account for this case at the write port of the register file which uses its counter to discard bits above the 16-bit threshold.

5 Managing Length-Adaptive-Data

This section will describe the special hardware used in BitSNAP to handle the interface between LAD bits and bit-parallel components. We will also present some special circuits needed to optimize the handling of LAD data. BitSNAP is only as effective in saving energy in so far as it can actually keep all the operands maximally compressed as shown in Fig. 3. For instance, operands coming from memory need to be compressed before entering the datapath. Another cause of concern is that certain operations on LAD numbers tend to decompress the representation away from the optimal — $\underline{0}111 + \underline{1}000$ results in $\underline{1}111$ using our hardware. Ideally, the datapath should compress this to $\underline{1}$, but attempting full compression on every operand introduces unacceptable energy and throughput costs. We will describe some practical compaction techniques that alleviate the decompression seen in operands.

5.1 LAD/Bit-Parallel Conversions

The boundaries between LAD to bit-parallel in Fig. 4 require conversion circuit blocks. We discuss the two main converters in this section, the conversion from LAD to parallel words, and the *compactor* block.

5.1.1 LAD to Bit-Parallel

When sending an address to the data memory, or a data value to the message coprocessor, the communicating bus needs to convert from a LAD stream (that is of variable length) to a 16-bit wide value. We use a counter, and a set of 16 1-bit registers controlled by a demultiplexor driven by the value of the counter. The LAD stream then writes each register in turn; when the delimiter arrives, the LAD bit is not acknowledged until the counter completes counting.

5.1.2 Compactor for Bit-Parallel to LAD

BitSNAP requires this conversion mainly on the data words coming from the memories (immediates and load values). Observe that we have no significance information stored in the memories; the bit-parallel words all effectively have the maximum significance of 16. We need to perform *compaction* on these words to compress the leading 0s and 1s. Fig. 8 shows the effects of different methods of achieving maximal compression. The *maximum compression* line (equivalent to that from Fig. 3) shows the maximum each operand can be compressed (i.e. the true significance of the operands), and represents the best any compaction method can achieve. The *full memory load compaction* shows what happens to the operand significance if we perform *full* compaction on the data values coming from memory (immediate values, and load values). We can get to within 5% of the optimal up to 8 bits, and within 1% above that (other side-effects such as decompression due to operations such as adds and subtracts prevent us from achieving the maximum).

Unfortunately, a full compaction circuit can be expensive both in terms of throughput and energy [6]. After performing some simulations, we decided the following scheme would achieve comparable results for less circuit complexity: For every word requiring compaction, we only attempt to compact the upper 12 bits: (1) The upper 8-bits are checked for all 1s or 0s, (2) In parallel, the next lower 4-bits are checked for all 1s or 0s, (3) If both tests are true, and both blocks have the same value, we compact the upper 12 bits into one LAD bit. If just the upper 8-bits are compressible, then we compact just that block.

This method has the added advantage that the 4 least significant bits can be sent bit-serially while the compaction is taking place. In addition, in between this compaction stage and the process on the BitSNAP datapath that will use this

Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)
1522-8681/05 $20.00 © 2005 **IEEE**

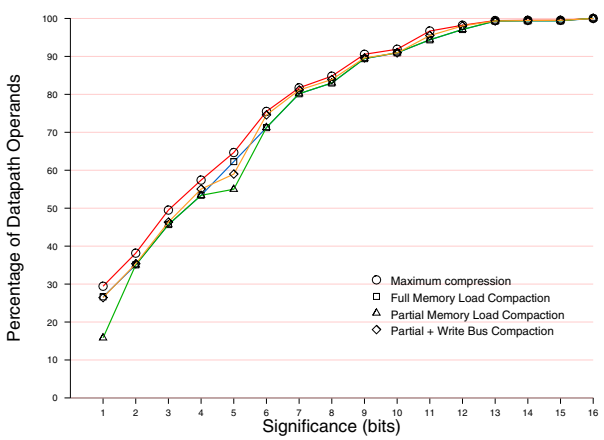

Figure 8. Operand significance using different compaction options

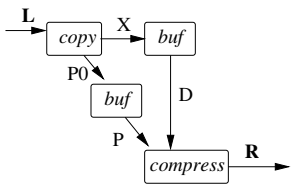

Figure 9. One-place compaction block

value, we insert several one-step compaction blocks (described more fully in the next section) that can compact by one bit: thus, 0001 will get compressed to 01. We found that three of these blocks provided the best tradeoff between latency and compaction effectiveness. The results of this method can be seen in Fig. 8 by the *partial memory load compaction* graph, which closely follows the full compaction results except for a large difference of more than 10% at 1 bit. Although this is a significant departure from ideal, the next optimization we describe will fix this suboptimal behavior.

5.2 One-Step Compaction

We use this process in both the immediate and load value compaction process described in the previous section, as well as on the write bus leading to the register file to tackle the decompression arising from computation on LAD operands.

A diagram of the processes required is shown in Fig. 9. The input arrives on channel L and a compressed output appears on channel R; the *copy* and *buf* processes are simple PCEVFB copies and buffers, while the *compress* process is given by the following (D is assumed to initialize with a null token):

$$
\begin{array}{l}
*[D?d; P?p; \\
\quad [\hat{d} \longrightarrow R!d \\
\quad []\neg\hat{d} \wedge \hat{p} \wedge p = d \longrightarrow \textbf{skip} \\
\quad []\neg\hat{d} \wedge \hat{p} \wedge p \neq d \longrightarrow R!d \\
\quad []d = null \longrightarrow \textbf{skip} \\
\quad []\textbf{else} \longrightarrow R!d \\
\quad]]
\end{array}
$$

The effectiveness of adding a single one-step compaction stage on the write bus (combined with the partial memory compression described previously) can be seen in Fig. 8 by the *Partial + Write Bus Compaction* graph. The compression is now closer to the maximum possible, and even better than just using full compaction on the memory buses.

6 Evaluation

We have performed an energy and performance comparison of the more commonly used LAD datapath units in BitSNAP versus bit-parallel SNAP/LE. For SNAP/LE, for which we have production quality datapath layout, we used Nanosim (a fast circuit simulator from Synopsys similar to Hspice in accuracy) to gain energy/performance numbers from extracted netlists with post-layout parasitics. For BitSNAP, we again used Nanosim, but this time on a pre-layout netlist with some parasitic estimates. This netlist was not optimized; most gates use a fixed nfet width of 4λ and pfet width of 8λ. We assume a TSMC 180nm process using SCMOS rules.

The energy and performance results are shown in Table 3 for BitSNAP and SNAP/LE running at 1.8V. In terms of energy consumption, as expected, LAD bit processing is somewhat more expensive than the per bit energy cost from bit-parallel SNAP/LE, due to the control circuitry overhead every cycle. For the bus interfaces, we break down the cost for reading/writing the bus (average of the two), the one-step compaction on the write bus, and the alignment process used in operations requiring two operands. Thus, for an addition, the total bus energy consumption per bit will be the sum of 0.43x3pJ (two read/one write operations), 0.71pJ (write-bus compaction), and 0.51pJ (alignment). The equivalent energy in SNAP/LE would be 8.1x3pJ per 16-bit operation. The register file energy in BitSNAP consists of a fixed cost for accessing the entire 16-bits plus 4-bit delimiter location (average of reads and writes), as well as a per bit cost for accessing the one-bit input/output bus. Notice that the fixed cost for the register-file is much less than an equivalent access for SNAP/LE. This is because the throughput of the register-file in SNAP/LE must meet the overall throughput of the processor (on the order of 200MHz). For BitSNAP, however, what matters is the cycle time of getting bits in and out from the serial bus, whose

Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)
1522-8681/05 $20.00 © 2005 **IEEE**

Table 3. Energy/Performance for BitSNAP vs. SNAP/LE

Structure	SNAP/LE		BitSNAP			
	Energy per word (pJ)	Total gate width μm	Energy per bit (pJ)		Freq. (MHz)	Total gate width μm
Bus Interface	8.1	473	Reader/Writer 1-step Comp. Align	0.43 0.71 0.51	334 235 257	46 158 77.3
Arithmetic	11.7	1559	1.3		248	359
Logical	5.4	788	0.37		322	168
Register File	31.2	16.9K	Core (20-bits) 1-Bit Ports	10.5 1.1	142 263	10.6K
Shifter	11.6	6K	1.4		238	515

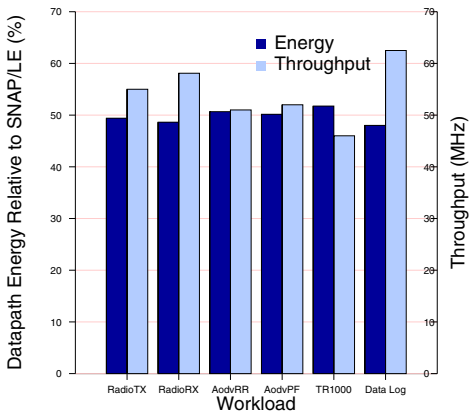

Figure 10. BitSNAP: Relative dynamic energy savings and throughput (at core voltage of 1.8V) for each workload

multiple cycles can effectively hide the slower cycle time of the parallel read and write to the register core.

On average, we found that a LAD bit takes approximately 10% of the energy for an equivalent 16-bit parallel word. Thus, any operation that can be compressed to 10 bits or less will save energy in BitSNAP, while any operation that uses more bits will end up expending more energy. We used the energy estimates in our architectural BitSNAP simulator to analyze the actual savings over SNAP/LE; across the sensor network workload presented earlier, the datapath energy consumption of BitSNAP ranges from 48% to 52% of the SNAP/LE's as shown in Fig. 10.

From a performance perspective, the throughput of most BitSNAP units exceed that of SNAP/LE, mainly because of the lack of wide completion trees. In a conventional bit-serial processor design, the overall execution time would slow down to about 1/16th the bit-parallel execution time, but because we use significance compression, we only suffer a slowdown proportional to the actual amount of bits we process. We show in Fig. 10 the expected throughput of BitSNAP on each of the benchmarks, as well as the estimated datapath power savings over SNAP/LE. *RadioTX* has the 802.11 based MAC layer taking a message from the application and transmitting it byte-by-byte across the radio interface, while *RadioRX* receives a message packet and hands it up to the application layer. In *AODVRR*, the MAC receives a route request which gets passed to the AODV-based ad-hoc routing layer that then performs a lookup in the node's routing table. The lookup is then passed back to the MAC for transmission. In *AODVPF*, the MAC receives a data packet meant for another node, that then gets forwarded via the next-hop found in the node's routing table. *TR1000* implements the radio stack for the TR1000 radio chip, and performs SEC-DED error coding and packet CRC. *DataLog* periodically samples a sensor reading and computes a running average. Across our benchmarks, BitSNAP executes at an average throughput between 46MHz to 62.5MHz, which corresponds to a slowdown of just 4.3 to 3.2 over SNAP/LE.

We have also provided the total transistor gate width breakdown for each circuit block in Table 3. As expected, BitSNAP has a considerably lower number of devices. SNAP/LE contains a total of 62K transistors (39,694μm total gate width) in the datapath, and we estimate that, based on the trends seen for the units in Table 3, BitSNAP will have a total of 23.5K transistors (13,648μm width). The total savings in gate width leads directly to static leakage energy reduction of 62%. For a sensor node processor that spends much of its time idle, with no dynamic switching, reducing the static energy can be a key factor in extending the lifetime of the node.

In terms of total energy consumption, we estimate that BitSNAP will incur about 70% of the dynamic energy consumption of SNAP/LE after accounting for the memories, fetch and decode. This translates to 152pJ per instruction at 1.8V and 17pJ per instruction at 0.6V. Table 4 compares the energy consumption of BitSNAP with other current and

Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)
1522-8681/05 $20.00 © 2005 **IEEE**

Table 4. Energy/Performance comparison with asynchronous processors

Processor	Process	Datapath Width	Voltage (V)	Throughput (MIPS)	Energy (pJ/ins)	Energy/bit (pJ/ins/bit)
ASPRO-216 [13]	$0.25\mu m$	16	1	25	1,000	63
AMULET3 [14]	$0.35\mu m$	32	3.3	120	1,291	40
Lutonium [15]	$0.18\mu m$	8	1.8	200	500	63
			0.5	4	43	5
MiniMIPS [11]	$0.6\mu m$	32	3.3	180	22,000	688
			1.5	60	3,700	116
Philips 80C51 [16]	$0.5\mu m$	8	3.3	4	2,250	281
SNAP/LE	$0.18\mu m$	16	1.8	200	218	14
			0.6	23	24	1.5
BitSNAP	$0.18\mu m$	16	1.8	54	152	10
			0.6	6	17	1.1

proposed asynchronous processors. As can be seen, BitSNAP's energy conscious design and dynamic significance compression yields much lower power consumption than other designs. We can also translate these energy numbers into the active power we would expect to observe in an actual sensor network. We noted that the workloads used to benchmark BitSNAP typically have between 70-250 dynamic instructions. In a typical sensor network scenario with extremely low activity levels of less than 10 events per second [2], this corresponds to an active power consumption in the realm of just 12nW-43nW at 0.6V.

7 Related Work

Dynamic significance compression has been widely proposed for reducing both static leakage and dynamic switching in cache or memory structures [17, 18] by turning off bytes of all zeros and ones. For processor cores, fixed width significance compression as proposed by Brooks and Martonosi clock-gated the upper 48-bits of a 64-bit integer unit if those input bits were zero [7]. They demonstrated 54.1% to 57.9% energy savings for that particular unit, but did not explore the cycle time or energy impact of doing a 48-bit zero detect every cycle, nor was the compression generalized to other parts of the architecture. Because the significance information is not propagated across cycles, and not stored in the register-file, we believe this scheme has limited usefulness.

Canal, González, and Smith [5] proposed tagging data values with bits specifying which bytes in a 32-bit datapath could be compressed. The instruction and data caches were also assumed to be modified to hold compressed data, with compression happening on a cache line fill. They demonstrated 30-40% activity factor reduction on both 32-bit datapaths and byte-serial datapaths, at a cost of increasing the CPI by 24-79%. However, because of the higher-level architectural nature of the study, there were no energy estimates for the resulting architecture. We believe there is a significant energy/area overhead of supporting caches with compression, generating the control, and handling the bypass paths in multi-cycle byte-serial synchronous implementations.

Nielsen and Sparsø demonstrated an asynchronous FIR filter bank [19] that used a two way compression scheme: data values are divided into two *slices*, a lower-order set of bits for small numbers, and a conditionally activated higher-order set of bits. The lower slice was tagged with a bit denoting whether the upper slice was used or not. This tag bit then conditionally activated the processing units and memories. They attributed about 30% of their power reduction over a similar synchronous implementation directly to the use of their slice based significance compression

Width-adaptive datapaths (WAD) for processors were first proposed by Manohar for dynamic compression on asynchronous parallel-word datapaths [6]. The WAD design was used by Fang and Manohar in QDI register file design, for compressing operands at a 4-bit block granularity using one WAD bit per block [8]. The vertically pipelined and block-skewed design showed an energy overhead of 25% over non-WAD register-files using pipelined completion [11] per block, but had an overall energy savings due to a 60% reduction in the activity factor.

8 Summary

In this paper, we have demonstrated a novel datapath design that applies dynamic significance compression to bit-serial data in the context of a low-power sensor network processor. By operating on variable length data, our length-adaptive datapath (LAD) BitSNAP processor can reduce the datapath energy consumption by roughly half while running at 20-25% the peak throughput of a comparable parallel-

Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'05)
1522-8681/05 $20.00 © 2005 **IEEE**

word processor. We estimate that BitSNAP will operate at 0.6V at 6MIPS while consuming just 17pJ/ins.

9 Acknowledgments

This work was supported in part by the Multidisciplinary University Research Initiative (MURI) under the Office of Naval Research Contract N00014-00-1-0564, and in part by NSF grants ITR 0428427 and NETS 0435190.

A Summary of CHP Notation

The CHP notation we use is based on Hoare's CSP [20]. A full description CHP and its semantics can be found in [21]. What follows is a short and informal description.

- Assignment: $a := b$. This statement means "assign the value of b to a." We also write $a\uparrow$ for $a := true$, and $a\downarrow$ for $a := false$.
- Selection: $[G1 \rightarrow S1 \; [] \ldots \; [] Gn \rightarrow Sn]$, where Gi's are boolean expressions (guards) and Si's are program parts. The execution of this command corresponds to waiting until one of the guards is $true$, and then executing one of the statements with a $true$ guard. The notation $[G]$ is shorthand for $[G \rightarrow skip]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]"
- Repetition: $*[G1 \rightarrow S1 \; [] \ldots \; [] Gn \rightarrow Sn]$. The execution of this command corresponds to choosing one of the $true$ guards and executing the corresponding statement, repeating this until all guards evaluate to $false$. The notation $*[S]$ is short-hand for $*[true \rightarrow S]$.
- Send: $X!e$ means send the value of e over channel X.
- Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
- Probe: The boolean expression $\overline{X}$ is $true$ iff a communication over channel X can complete without suspending. In the event channel X communicates data, $x := \overline{X}$ assigns the data to variable x without completing the communication on X (value probe). This action suspends until data is available on X.
- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$ or S, T.
- Simultaneous Composition: $S \bullet T$ both S and T are communication actions and they complete simultaneously.

References

[1] Clinton Kelly IV, Virantha Ekanayake, and Rajit Manohar. SNAP: A Sensor-Network Asynchronous Processor. In *Proc. International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2003.

[2] Virantha Ekanayake, Clinton Kelly IV, and Rajit Manohar. An ultra low-power processor for sensor networks. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[3] P. Levis et al. The emergence of networking abstractions and techniques in TinyOS. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[4] J. Hill et al. System architecture directions for network sensors. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[5] R. Canal, A. González, and J.E. Smith. Very Low Power Pipelines using Significance Compression. In *Proc. International Symposium on Microarchitecture*, December 2000.

[6] Rajit Manohar. Width-Adaptive Data Word Architectures. In *Proc. International Conference on Advanced Research in VLSI*, March 2001.

[7] D. Brooks and M. Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[8] D. Fang and R. Manohar. Non-uniform access asynchronous register files. In *Proc. International Symposium on Asynchronous Circuits and Systems (ASYNC)*, Apr 2004.

[9] P. B. Denyer and David Renshaw. *VLSI Signal Processing; A Bit-Serial Approach*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[10] Richard I. Hartley and Peter F. Corbett. A digit-serial silicon compiler. In *Proceedings of the 25th ACM/IEEE conference on Design automation*, pages 646–649. IEEE Computer Society Press, 1988.

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

[12] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Proc. International Conference on Advanced Research in VLSI*, pages 351–373, 1991.

[13] M. Renaudin, P. Vivet, and F. Robin. ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessr. In *4th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '98)*, pages 22–32, 1998.

[14] S. B. Furber, D. A. Edwards, and J. D. Garside. AMULET3: a 100 MIPS asynchronous embedded processor. In *Proc. International Conf. Computer Design (ICCD)*, 2000.

[15] Alain J. Martin, Mika Nystrm, and Catherine G. Wong. Three generations of asynchronous microprocessors. *IEEE Design and Test of Computers, special issue on Clockless VLSI Design*, Nov/Dec 2003.

[16] Hans van Gageldonk, Kees van Berkel, Ad Peeters, Daniel Baumann, Daniel Gloor, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. In *ASYNC '98: Proceedings of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 0096. IEEE Computer Society, 1998.

[17] Luis Villa, Michael Zhang, and Krste Asanovic. Dynamic zero compression for cache energy reduction. In *Proc. International Symposium on Microarchitecture*, Dec 2000.

[18] Nam Sung Kim, Todd Austin, and Trevor Mudge. Low-energy data cache using sign compression and cache line bisection. In *2nd Annual Workshop on Memory Performance Issues*, May 2002.

[19] Lars S. Nielsen and Jens Sparsø. Designing asynchronous circuits for low power: An IFIR filter bank for a digital hearing aid. *Proceedings of the IEEE*, 87(2):268–281, February 1999.

[20] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, pages 666–677, 1978.

[21] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Straunstrup, editor, *Formal Methods for VLSI Design*, pages 237–283. North-Holland, 1990.

1522-8681/05 $20.00 © 2005 **IEEE**