In: "1998 IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems" ("Async98" Symposium), San Diego, CA

A Fast Asynchronous Huffman Decoder for Compressed-Code Embedded Processors

Martin Beneš
Department of EECS
U.C. Berkeley
Berkeley, CA

Steven M. Nowick*
Dept. of Computer Science
Columbia University
New York, NY

Andrew Wolfe†
S3 Inc.
Santa Clara, CA

Abstract

This paper presents the architecture and design of a high-performance asynchronous Huffman decoder for compressed-code embedded processors. In such processors, embedded programs are stored in compressed form in instruction ROM, then are decompressed on demand during instruction cache refill. The Huffman decoder is used as a code decompression engine.

The circuit is non-pipelined, and is implemented as an iterative self-timed ring. It achieves a high-speed decode rate with very low area overhead. Simulations using Lsim show an average throughput of 32 bits/25 ns on the output side (or 163 MBytes/sec, or 1303 Mbit/sec), corresponding to about 889 Mbit/sec on the input side. The area of the design is extremely small: under 1 mm^2 in a 0.8 micron full-custom layout.

The decoder is estimated to have higher throughput than any comparable synchronous Huffman decoder (after normalizing for feature size and voltage), yet is much smaller than synchronous designs. Its performance is also 83% faster than a recently published asynchronous Huffman decoder using the same technology.

1 Introduction

Embedded systems are now widely used in many consumer products, such as automobiles and portable electronics. These systems typically include a microprocessor which is embedded inside a commercial product, and which executes an application program determined by the system designer. Such systems are often limited by constraints on size, weight, power consumption and price. Therefore, a key challenge in designing cost-effective and high-volume embedded systems is to reduce the chip area of the system.

A novel approach to designing compact embedded systems has been proposed, called a **compressed code architecture** [24, 12]. In this approach, instructions are stored in compressed form in memory, then are decompressed when brought into the cache. As a result, a significant reduction in instruction memory size may be obtained. Using a Huffman encoding scheme, a compression ratio of 73% was reported for the MIPS instruction set.

The key component of this architecture is a hardware **instruction decompression circuit**. The design of this circuit is a highly-constrained problem. In particular, the circuitry must be very fast (since it is on the critical path during cache refill) but also very small (otherwise the savings in instruction memory will be lost to the area increase due to the decompression circuit).

Very recently, we introduced the first prototype design for such an asynchronous decompression circuit [1]. Instructions are compressed in memory, using a Huffman encoding scheme [10]. Huffman codes are variable-length, where the shortest codes are assigned to the most frequent symbols. In principle, a Huffman decoder is an excellent match for asynchronous design: an asynchronous decoder can be highly-optimized for common, rather than rare, codes, and thus obtain improved average-case performance. In contrast, performance of a synchronous design may be limited, due to a worst-case fixed clock rate, or the design may have a large area overhead to handle worst-case computation efficiently.

Recently, a number of asynchronous chips have been successfully designed and/or fabricated, both for microprocessors and DSPs [21, 11, 14, 15, 16, 26], including embedded processors [6, 7, 8]. Several of these designs have demonstrated the benefits of asynchronous design for average-case operation.

*This research was funded in part by by an NSF CAREER Award MIP-9501880 and by an Alfred P. Sloan Research Fellowship.

†This work was funded in part from NSF under award MIP-9408462 and by an AT&T foundation gift.

In this paper, we propose a new architecture an implementation of an asynchronous Huffman decoder. The design uses an entirely new organization, and is *83% faster* than our earlier design [1], using the same 0.8μ CMOS technology, with no increase in area. The circuit is non-pipelined, and is implemented as an iterative self-timed ring. It is largely implemented using dynamic domino dual-rail logic. It achieves a high-speed decode rate with very low area overhead.

Simulations using Lsim show an average throughput of 32 bits/25 ns on the output side, or 163 MBytes/sec (or 1303 Mbit/sec). This corresponds to a throughput rate of about 889 Mbit/sec on the input side. The area of the design is extremely small: under 1 mm^2 in a 0.8 micron full-custom layout. The decoder is estimated to have higher throughput than any comparable synchronous Huffman decoder (after normalizing for feature size and voltage), yet is 5-10 times smaller than most synchronous designs.

The paper is organized as follows. Section 2 gives background on compressed code processors, and discusses related work on Huffman decoders. Section 3 gives an overview of the functional operation of the decoder. Section 4 presents details of the architecture and the implementation of its components. Simulation results are presented in Section 5, which also discusses some remaining bottlenecks in the new design. Section 6 compares our new design with our previous asynchronous decoder design, and Section 7 presents conclusions.

2 Background

2.1 Compressed-Code Embedded Processors

An embedded system is loosely defined as one which incorporates microprocessors, or microcontrollers, yet is not itself a general-purpose computer [24, 12, 13]. Embedded systems are extremely widespread, including: controllers for automobiles, airplanes, portable consumer electronics, etc. These systems typically include a microprocessor, or microcontroller, which executes a stored program determined by the system designer. The instruction memory is either integrated on-chip, or is external [13, 25].

Practical embedded systems are often limited by constraints on size, weight, power consumption and price. In particular, for low-cost and high-volume systems, the cost of the entire system is often closely tied to the total area of the chip(s). In such systems, a significant percentage of the area may be devoted to the instruction ROM, storing the program code [13]. Therefore, techniques to reduce the size of the instruction ROM are of critical importance.

Recently, Wolfe *et al.* introduced a useful approach to designing compact embedded systems, called a *compressed-code architecture* [24, 12]. In this approach, embedded programs are stored in memory in *compressed format*, then are *decompressed* in the instruction cache, and executed in standard format. This solution results in reduced memory size, with only a minimal impact on processor performance.

In more detail, Wolfe proposed that a program be compressed at development time, and stored in instruction memory. Compression is performed on small fixed-size program blocks (1 byte each).[1] At runtime, on demand, the cache refill engine locates the compressed cache line in instruction memory, decompresses it, and writes it into the instruction cache. Note that decompression is required only on a cache miss; at all other times, the processor simply executes normal uncompressed instructions, which are already present in the cache.

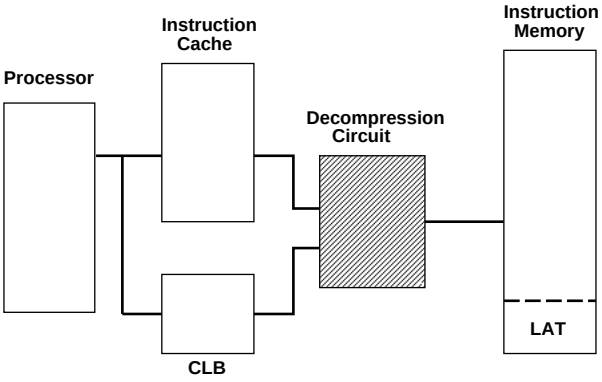

Figure 1. Block diagram of compressed-code architecture

A simple block diagram of a compressed code architecture is shown in Figure 1. The *LAT* (line address table) serves as a form of page table for the compressed cache lines, indicating the actual location of each compressed line in instruction memory. The *CLB* (cache lookaside buffer) serves as a form of TLB for the line address table; it caches the most recently accessed LAT entries, to speed up cache line refill. The overheads of LAT and CLB are very small (see [12] for details).

This compression scheme allows a substantial reduction in the size of instruction memory. Using a Huffman encoding scheme for instructions, experiments indicate that a compression ratio of 76% (*i.e.,* compressed program size/uncompressed program size) can be obtained for MIPS processors on typical applications [12]. This ratio includes the small overheads required to detect and align variable-length compressed cache lines in instruction memory. This reduction in program size can translate into lower cost, weight and power consumption for the entire embedded system. It can also allow the addition of extra program features without increasing the budget for memory size. Finally, compression can result in lower system power, since fewer bus cycles are required to fetch compressed instructions.

Related Work. Several alternative approaches have been proposed for code compression in embedded processors, each with some limitations.

A modified instruction set, called *Thumb* [19], was recently introduced for the ARM processor core, which in-

[1] An alternative scheme, of compressing the entire program as a whole, is impractical for any realistic decompression hardware.

cludes new 16-bit instructions taken from the standard 32-bit ARM instruction set. The goal is to reduce instruction bandwidth for embedded applications. However, while this approach allows for increased code density, it requires a commitment to a specialized instruction set, with an entirely new set of supporting software tools (*e.g.,* compiler, assembler, linker, etc.). In contrast, Wolfe's decompression approach is quite general: it can be used to compress instructions in any existing instruction set, without modification.

Liao, Devadas and Keutzer [13] propose software and hardware approaches to code compression, using dictionary lookup with "mini-subroutines". However, while this method is promising, it is orthogonal to Wolfe's approach: hardware compression can be used *in addition* to their approach. Furthermore, their optimization is limited to finding exact matches of entire instructions (*i.e.,* same opcode and operands); in contrast, Wolfe's method looks for matches at a finer granularity (byte-level). Finally, their method may adversely interact with existing code optimization techniques (see [13]), while Wolfe's method can be used with any existing code optimizer.

Finally, a method by Yoshida *et al.* [25] compacts instruction memory using a logarithmic-based compression scheme, along with a ROM-based decompression table. However, while significant compression can be obtained for small programs, the ROM table itself may be over 270 KBytes for large examples, rendering the approach impractical.

2.2 Related Work: Huffman Decoders

Our design is estimated to have better performance and area than existing synchronous Huffman decoders [18, 9, 4, 17]. Most of these decoders are targeted for digital video applications, and focus on the MPEG-2 VLD decoder for the DCT coefficient table. The Huffman code used in MPEG-2 has 114 code words, varying in length from 1 to 16 bits, where one of the codes is an escape sequence followed by fixed length code, extending the maximum code length to 28 bits. The structure of the code is quite simple, and the code length can be easily derived from the number of leading zeros. The complexity of this code is therefore simpler than our MIPS-based code which has 256 code words.

For example, the decoder by Park *et al.* [17] has an area is 3.5 mm^2 in a 0.65μ CMOS process, compared with only 0.75 mm^2 for our design in a 0.8μ process. While the authors claim a peak performance of 680 Mbit/sec, this is based on the decoding of 17-bit codewords at 40 MHz. Since the worst possible 114 symbol Huffman code has an average symbol length of 7 bits,[2] and a more typical code would have an average symbol length of no more than 5 bits, a more realistic sustained performance is in the 200–250 Mbit/sec range.

One of the fastest designs is a recent Toshiba chip [5], using 0.5μ nFets, which was clocked at 200 MHz output rate. The design uses aggressive but area-expensive cricuit techniques like differential amplifying logic, which we have not used. The 28k-transistor VLD macro occupies about 5 mm^2 in a 0.8μ process (0.5μ nFETS are used), and operates at 3.3 V. In contrast, our design uses 6100 transistors in a total area of 0.75 mm^2. After scaling to our conditions (5 V and 0.8μ process), their normalized rate is estimated at 150 MHz.[3] In comparison, our asynchronous design has a rate of 163 MByte/sec.: our design is faster, and is also roughly 5 times smaller.

3 Huffman Decoder: Functional Overview

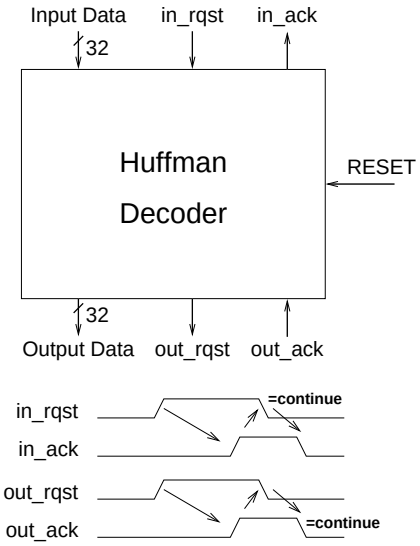

Figure 2. Decoder overview

A simple high-level diagram of the Huffman decoder is shown in Figure 2. Input data is fetched from memory using a simple 4-phase asynchronous handshaking protocol [3]. This data consists of a single compressed cache line, loaded 32-bits at a time. The compressed line itself contains a sequence of variable-length input symbols, each of which is a Huffman code for 1 byte of an instruction. The decoder then parses each individual input symbol and translates it into the corresponding output symbol (1 byte). As output, the uncompressed cache line is delivered to the instruction cache using a 4-phase protocol. Existing logic within the processor generates memory addresses. The cache refill logic counts the 32-bit data words produced by the decoder, and it resets the decoder after 8 words have been produced, which correspond to an entire uncompressed cache line.

[2] Given a set of 114 symbols, the corresponding Huffman code depends on the assumed frequency of the symbols. In the worst case, all symbols have the same frequency, therefore all codes are 7 bits: the weighted mean code length (dynamic average) is 7 bits and the maximum code length is 7 bits. For details on Huffman codes, see [10].

[3] However, since the chip includes both VLD and IDCT blocks, it is unclear whether the speed is limited by the VLD critical path.

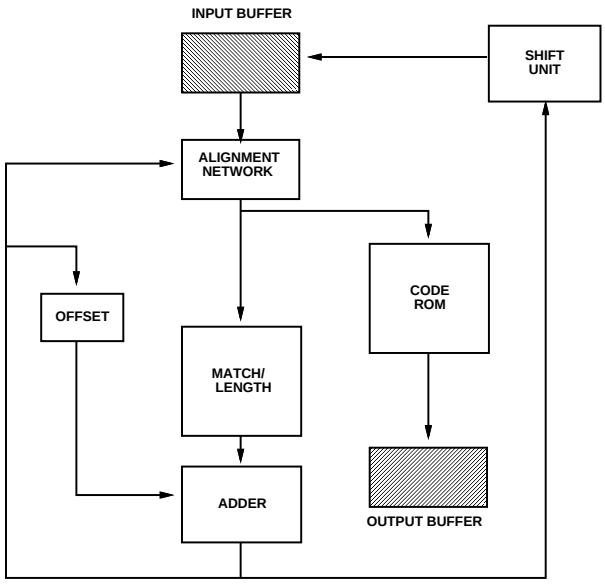

Figure 3. Functional block diagram of decoder

A functional block diagram is shown in Figure 3, indicating the flow of basic operations. The Input Buffer holds the current input data, which is supplied to the decoder. The Alignment Network (discussed shortly) holds the current aligned version of the buffer, corresponding to the start of the current symbol.

In each decode cycle, two operations occur in parallel. (i) *Symbol Decoding:* Using a lookup table, the aligned current input symbol is translated into an output symbol. This table is implemented by an optimized Code ROM. (ii) *Updating Input Stream:* The decoder determines the length of the current symbol and shifts the inputs appropriately, to prepare the next input symbol. A Match/Length unit is used to compute the length of the current symbol. This length is used to compute the shift amount, in order to retire the current symbol from the input stream. To reduce hardware overhead, a 2-step shift is used. The Shift Unit shifts the Input Buffer an integral number of bytes: 0, 1 or 2. Since a residual misalignment (1-7 bits) may still occur, the Alignment Unit, serving as a small barrel shifter, adjusts the Input Buffer by the remaining bit offset. The previous alignment Offset is saved, and added to the current symbol length, to compute the final shift amount; it is fed to both the Shift Unit (0, 1 or 2 byte shift) and the and the Alignment Network (0-7 bit shift).

Details of the management of the input and output buffers and their interaction with the overall system, such as requesting the next input word, signaling a completed output word, and indicating decode completion, have been described in [1], and will not be discussed further here.

4 Huffman Decoder: Architecture and Implementation

This section presents an overview of the new decoder architecture, as well as details on the implementation of its components.

4.1 Overview

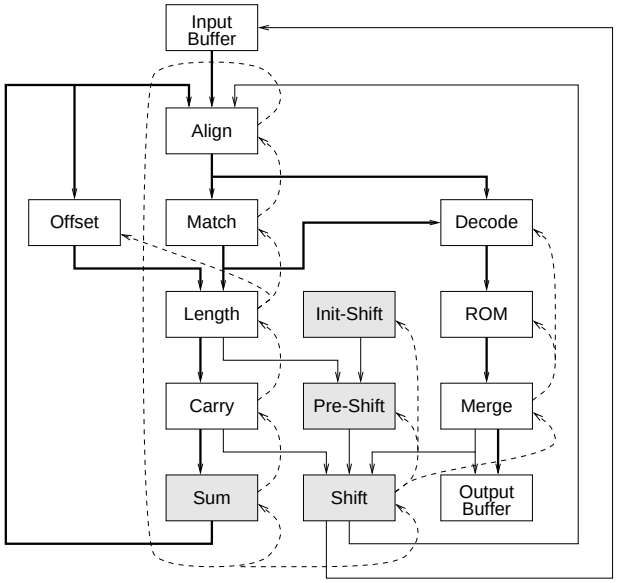

Figure 4. Structural block diagram: self-timed ring architecture

A structural diagram of the new decoder architecture is shown in Figure 4. The decoder is organized in the form of a self-timed ring. These rings have been studied in depth, and applied to a number of designs such as a high-speed self-timed divider chip [22, 23]. In particular, Williams introduced a novel *zero-overhead* design style for self-timed rings, where control operations (*e.g.,* precharging in a dynamic implementation) are done in background mode while other stages are computing. Therefore, a token can effectively propagate through each logic stage with no control overhead. Such structures have been extended to multi-rings as well [20]. Zero-overhead rings have been highly effective for the implementation of iterative computations, where latency is paramount.

Therefore, the core of our new architecture is a self-timed ring, generalized to allow parallel computation threads. As shown in Figure 4, most of the stages are dynamic (shown in white); the remainder are static (shaded gray). Most of the dynamic circuits are implemented in precharged domino dual-rail logic. The dotted lines in the figure represent handshaking communication, *i.e.,* synchronization between stages, and will be discussed later.

There are 2 main goals of the architecture:

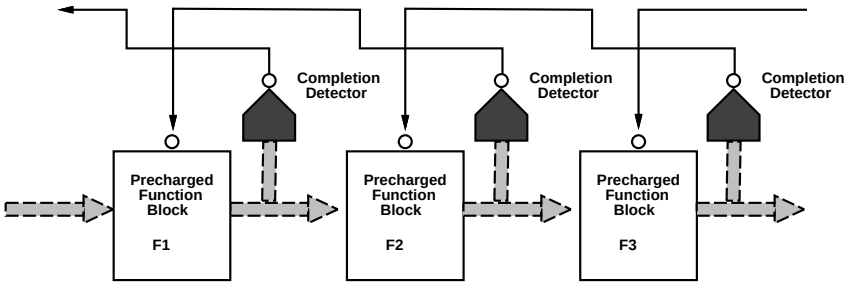

Figure 5. PS0 synchronization scheme

1. *balanced stages:* The goal is to reasonably balance stage delays, so that control overhead can be nearly eliminated during computation. However, a precise balance is not possible, since a few stages have highly-variable data-dependent delays (*e.g.,* Match, Pre-Shift, Shift).
2. *optimal scheduling:* The goal is to have as much parallel computation as possible, and to schedule computations as early as possible, to improve total latency.

Below is a short overview of the new architecture (refer to Figure 4). There are basically 3 parallel computation threads: (i) Align through Sum; (ii) Length through Shift; and (iii) Align through Merge (to Output Buffer).

- *Align through Sum* has 5 stages. The Match/Length Unit of Figure 3 is implemented as two stages: Match and Length. Match produces a 1-hot output, indicating the *match class* to which the current input symbol belongs. A symbol may belong in one of several classes of Huffman codes. A Length ROM then translates this class into the actual token length.

  The Adder of Figure 3 is divided into a Carry stage followed by a Sum generation stage. Once the sum is computed, it is passed as select bits to the Align stage, indicating the new 0-7 bit shift, and is also passed to the Offset stage. Offset is a dynamic identity function (*i.e.,* buffer), which acts like a register; it stores the current offset for the next computation.
- *Length through Shift.* This thread implements byte-shift operations: 0, 1, or 2 byte shifts. Init-Shift is used only for an initial load, at the start of decoding.

  While the functional diagram in Figure 3 suggests that shift occurs *after* add is complete, our actual architecture uses two optimizations. First, in some cases, the upper bit from Length ROM immediately indicates that a 1-byte shift is needed (Pre-Shift). Second, any additional 1-byte shift is indicated by Carry: there is no need to wait until Sum is computed.
- *Align through Merge.* In this thread, the current input symbol is decoded and written into the Output Buffer. The Code ROM of Figure 3 is now implemented by 3 stages: Decode, ROM, and Merge. These stages implement a compact lookup table, which uses a 2-way decoding process. Each match class (indicated by Match stage) provides a unique enable signal to a corresponding portion of the Decode stage. However, simultaneously with the processing of the Match stage, the aligned input data itself is forwarded to the ROM decoders, to identify one *potential* word line to enable for each match class. When the match signal finally arrives, it enables the last stage of exactly one ROM decoder, and the ROM generates the correct output symbol.

All 3 threads are joined at a *single* synchronization point: the Align stage. More precisely, the third thread (Align through Merge) must complete before Shift can complete, then the remaining two threads are joined at the Align stage.

The remainder of this section gives details on the implementation of the stages. Note that some of the stages are the same as in the previous prototype [1]; others are new. However, the overall architecture, scheduling, and inter-stage synchronization schemes are new. We will present both new and old stages (old in less detail) for coherence. A final comparison of the new architecture with the earlier prototype appears in Section 5.

4.2 Inter-Stage Synchronization

To synchronize between stages in the ring, we use an adaptation of a scheme called **PS0** by Williams [22], as shown in Figure 5. In **PS0**, precharged dual-rail function blocks are organized into a ring. Each function block has a completion detector, which controls the previous stage. As an example, once F_2 has evaluated, completion of evaluation is detected, which then enables F_1 to precharge. Once F_2 precharges, completion of precharge is detected, which then releases precharge of F_1.

Thus, the configuration enforces two synchronization requirements: (i) stage F_i precharges only after F_{i+i} has evaluated; (ii) stage F_i can evaluate a new token (precharge deasserted) only after F_{i+1} has finished precharging. An additional requirement — that F_i is done precharging before F_{i+1} begins precharging – must be insured by delay assumptions. This scheme has been effectively used in zero-overhead implementations, where each stage is ready to evaluate as soon as data arrives.

For the new architecture, we use a modified version of **PS0**, as shown in Figure 6. A precharged *matched*

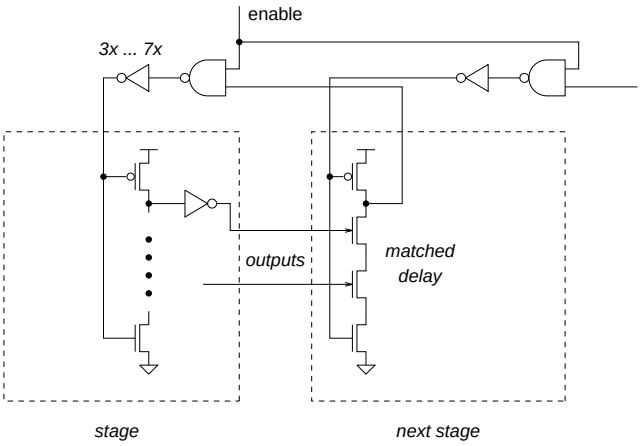

Figure 6. Huffman Decoder synchronization scheme

delay, instead of a completion network, is used in most stages of our design. The figure shows the matched delay in "Next Stage"; the delay matches the worst-case path through the function block, and indicates to the previous stage when the block is done.

Also, we include a global *enable* signal. When deasserted, this signal resets all dynamic stages to the precharge state (*e.g.,* between decoder requests).

In Figure 4, inter-stage synchronization is indicated by dotted arrows.

4.3 Huffman Encoding and Match

We use a variant of Huffman encoding, introduced in [1], to optimize the decoder implementation. On the one hand, the *length* of each Huffman code is precisely determined by the frequency distribution of the original input alphabet. On the other hand, the actual *binary code* assigned to each input symbol is flexible, as long as the prefix property and the code length requirement are maintained.

We therefore cluster Huffman codes into *match classes*. All codes in a given match class have the same length, but there may be more than one match class for each length. All codes in each class are assigned the same unique binary prefix; they differ only in a small number of suffix bits, used to enumerate the different members of the class. For example, in Table 7, there is a single match class, 0, of 2-bit codes, containing 1 member: code 00; there are no enumerating bits. For 5-bit codes, there is a single match class, 1, which contains 4 distinct codes; the enumerating bits are indicated by two dots. Each code has the same prefix: 010 (a −, indicates a don't-care, assuming a match higher in the table did not occur). The remaining 2-bit suffix is used to enumerate the codes in the class: 01000, 01001, 01010, 01011.

The benefit of using clustered Huffman codes, partitioned into a moderate number of match classes and each with a few enumerating bits, is to simplify the implementation of 2 stages: Match and ROM Decode. Given that we are encoding each byte of instruction into a Huffman code, the requisite 256 codes can be partitioned into 31 distinct classes, shown in Table 7, along with their qualifying bit patterns. The matching process starts from the top and examines each bit pattern until a match is found.

class	bit pattern	length
0	00	2
1	0-0..	5
2	0 ...	6
3	-000..	
4	-00 ...	7
5	-0-00..	
6	-0-0-0.	
7	-0-0--0	
8	-0	8
9	--00....	
10	--0-00..	
11	--0	9
12	---00....	
13	---0-0...	
14	---0--00.	
15	---0--0-0	
16	---0	10
17	----0.....	
18	-----00000	
19	-----0.....	11
20	------00...	
21	------0-00.	
22	------0-0-0	
23	------0	12
24	-------0....	
25	--------0...	
26	---------0...	13
27	----------0..	
28	-----------0.	
29	------------0	
30	.	14

Figure 7. Match Classes

The *Match Logic* in Figure 8 performs the matching task. Each block represents the 4-transistor circuit shown on the bottom. Both phases of the single input bit, b_i (output of the Align stage), are used at each level of the match tree. The row of cells labeled *b5*, for example, is connected to the two phases of the input bit 5. The single *decode_in* input signal is driven by a sequence of prior stages that represents a value of the preceding bits.

The outputs of the circuit are its leaves, which are labeled by number (0-30). Each output corresponds to a unique match class, and is a *decode_out0* or *decode_out1* signal from the indicated stage. Each such output (n) is inverted, to produce the final match signal ($m(n)$) (as suggested by the inverter at top). In some cases, several outputs are connected in a wire-or circuit to indicate a length class. Since only one class is detected, the circuit's output is 1-hot, and therefore any of the 31 outputs going low indicates completion to the following stages. Note that this circuit is constructed such that the shortest, and thus most

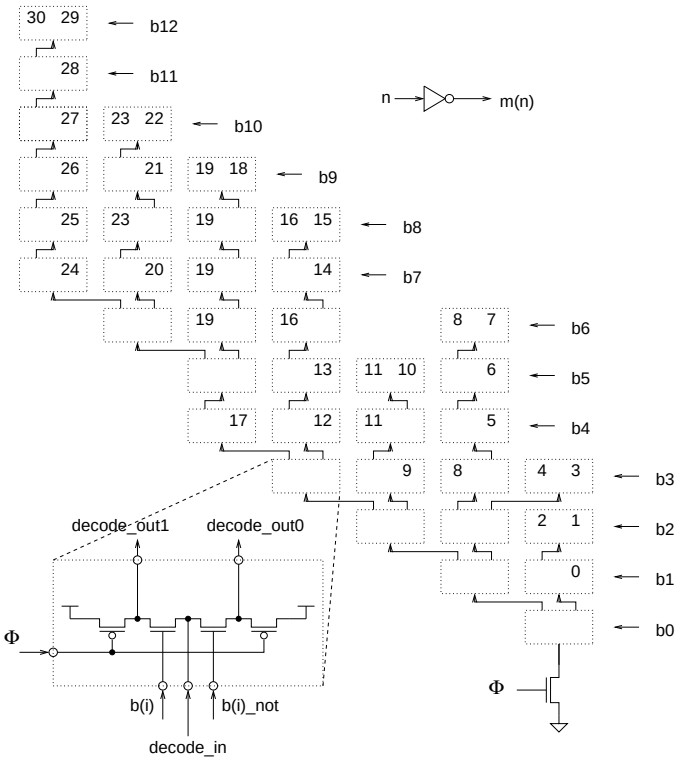

Figure 8. Class Matching Logic

common, codes are matched using the fewest levels of logic; therefore, the average response time is much faster than the worst-case.

Note that a deep N-channel stack is used, to detect matches with the longest codes. To improve performance, the bottom three transistors in the stack are somewhat widened (2-4x). While worst-case detection is very slow (nearly 3.5 ns), average-case detection is much faster (roughly 1.5 ns).

The completion generation for the Match stage is a challenge: because of the highly data-dependent operating speed, it is not practical to use a fixed matched delay. On the other hand, generation of a true completion signal, using a completion detection tree, would have excessive overhead (combining 31 distinct outputs). Our solution is to *defer* completion generation for Match to the subsequent stage: Length ROM (see below).

4.4 Length ROM and Adder

Figure 9 gives greater detail on the implementation of several components, for the following subsections.

Length ROM. A small Length ROM translates the Match outputs into the 4-bit binary length of the current input symbol. The inputs to the ROM are the 31-wire one-hot outputs of Match. Each input corresponds to one match class, and serves as a word line in the ROM. The ROM then produces a 4-bit dual-rail output (8 wires), which is the length of the input symbol. As an optimization, the P/K/G Generation unit of the subsequent adder is merged directly into the Length ROM, to form a single stage, which we call *Length*.

The ROM also has a 9th output which is a matched delay. This output is an extra bit line, enabled by every input, which is used to generate a completion signal. As indicated earlier, it is difficult to build a simple done signal for the preceding stage, Match, which has 31 outputs. Therefore, we use this 9th output of Length ROM as a simply-implemented, and deferred, "done" signal for Match. In addition, a second, delayed version of this 9th output is used as the done signal for the entire Length stage, including P/K/G Generation.

Adder. A small carry-lookahead adder is used, to compute the total shift amount. The adder is broken into 3 stages: *P/K/G Generation* (merged into the Length stage), *Carry Generation*, and *Sum*.[4] Each of these stages corresponds to a Williams-style pipeline stage as in Figure 5. Dual-rail signals are indicated by thick lines; individual wires are indicated by thin lines.

The *P/K/G Generation* unit takes two 3-bit operands: the current bit-offset of the Alignment network (stored in *Offset*) and the length of the current input symbol (indicated by *Length ROM*). Each of the operands is dual-rail. The unit computes whether each bit is a propagate, kill, or generate bit; the result is a 1-hot code for each bit, using 3 wires.

The *Carry* stage computes the final dual-rail carries, which are used to generate the final sum. The stage transforms the three-wire P/K/G inputs into propagate bit (Pi) and carry-in bit (cin) outputs. Each Pi output is dual-rail, as is each cin output. The two rails of each Pi output are generated by ORing K+G inputs (no propagate) and by passing the P input (propagate), respectively. The dual-rail outputs of the Carry stage are fed to the Sum stage. The upper carryout bit (rightmost output of Carry stage) is used to control the Shift stage and will be discussed later.

The *Sum* stage computes the final 3-bit sum, which is the new offset (0-7 bits) for the Alignment Network. Any larger shift, of 1- or 2-bytes, is implemented by the Shift stages.

4.5 Shift and Align

In each iteration, an input symbol is both decoded and retired from the input stream. To retire the input, the Input Buffer must be shifted by the length of the current symbol, which is computed by the Length ROM. To simplify the hardware, we use a 2-step shift process: *shift* and *align*.

Shift. The Shift Unit shifts the Input Buffer an integer number of bytes: 0, 1 or 2. Since a residual misalignment (1-7 bits) may still occur, the Alignment Unit, serving as a small barrel shifter, adjusts the Input Buffer by the remaining bit offset. The *new shift amount* is computed by adding the current offset in the Offset stage (indicating current bit alignment of the Align stage) to the current input

[4]Note: to simplify the topology of Figure 9, the msb is shown on the right-side, and the lsb on the left side.

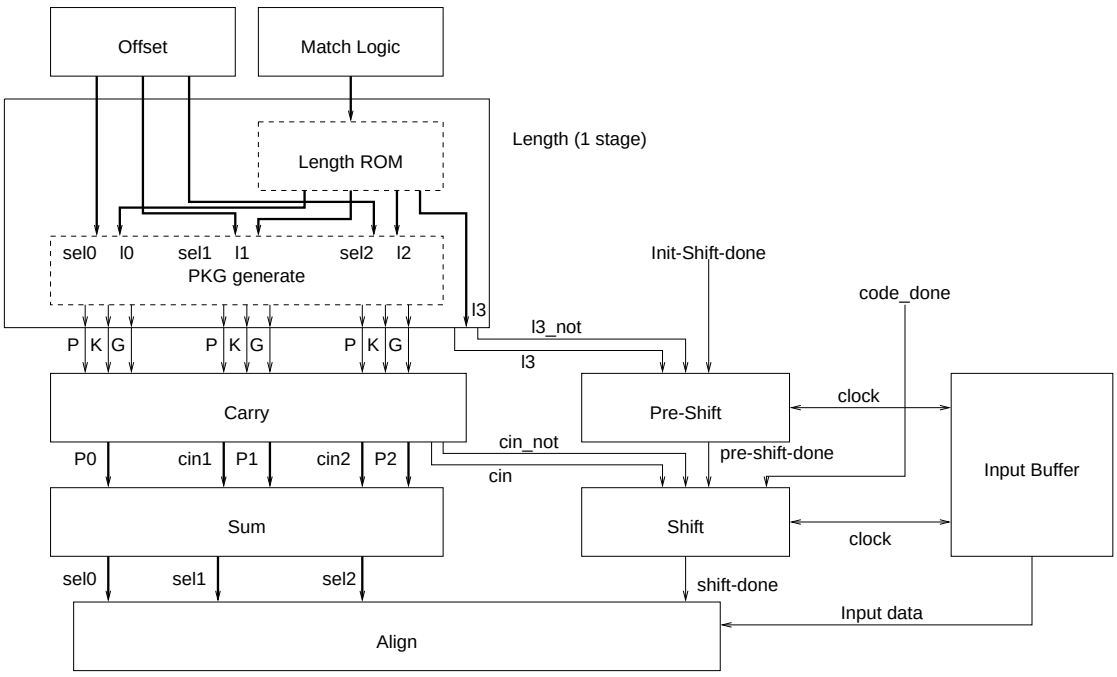

Figure 9. Detailed block diagram (partial): self-timed ring architecture

symbol length (provided by Length ROM). In our implementation, the maximum shift amount is 21 bits: given a maximum-length 14-bit Huffman code and a 7-bit current Offset.

Figure 10 shows the new shift control, which incorporates 3 stages: Init-Shift, Pre-Shift and Shift. Init-Shift is used only for an initial load, at the start of a new decode request (see [1]).

As an optimization, a *Pre-Shift* stage is used to implement an early byte-shift. In particular, whenever the current (Huffman-encoded) input symbol has a length of 8 bits or greater, at least 1 byte-shift is required. In this case, the upper bit (l3) from Length ROM itself indicates that a 1-byte shift is needed, *i.e.,* the new token length is at least 8.

The *Shift* stage is used to implement a normal byte-shift, once the current Offset has been added to the new symbol length. As an optimization, this byte-shift is requested by an upper carryout from the Carry stage: there is no need to wait until Sum is computed.

Interestingly, if a 1-byte shift is required, it may be initiated either by Pre-Shift or Shift, depending on the current input and offset. However, whenever a 2-byte shift is required, a 1-byte shift is always initiated by Pre-Shift; therefore, the Shift stage only needs to initiate a 1-byte shift, in all cases.

Figure 10 shows a more detailed implementation of the shift controller; it integrates the Init-Shift, Pre-Shift, and Shift blocks into one circuit. Each shift block consists of two separate gates. One generates the "up" signal (u0, u1, and u2), and the other generates the "down" signal (d0, d1, d2). The "u" and "d" signals are then merged into *in_clk* (input buffer clock), through a simple clock generation network, which is used to clock a 1-byte shift in the Input Buffer. A delayed version of the asserted *in_clk* is also fedback as an input through an RS latch, which in turn drives *in_clk* low. The RS latch is used to eliminate potential hazards.

Each of the 6 gates evaluates only after appropriate conditions have been met. For example, during a pre-shift, the "Pre-Shift" gate will set u1 high as soon as l3 goes high (from Length ROM). This, in turn, will drive *in_clk* high and thereby drive *clk_up* high. The next gate, once it sees that both u1 and *clk_up* are high, sets d1 high, which in turn sets *in_clk* low, and thus completes the pre-shift operation.

Each gate may evaluate under several different conditions, and the whole shift unit is thus able to support all possible combinations of shift and pre-shift requests.

The last stage also serves as a synchronization point of the shift and ROM threads: *shift_done* will not go high until symbol decode completes (as indicated by *code_done*).

Align. The *Align* stage effectively implements 3 stages of 2-1 multiplexers in hazard-free precharged domino logic. The stage has been described in [1]. The 3 select bits, from the Sum stage, indicate the desired shift amount (0-7 bits). An important new feature of the Align stage, in the self-timed ring, is that it is the *unique synchronization point*. That is, Align initiates a new cycle; it can proceed only when all 3 parallel threads are completed: (i) sum generation, and (ii) shift (which in turn depends on symbol decode: Code ROM + Merge).

Align is implemented as one complex precharged dynamic gate. The Sum output bits (thread (i)) are dual-rail,

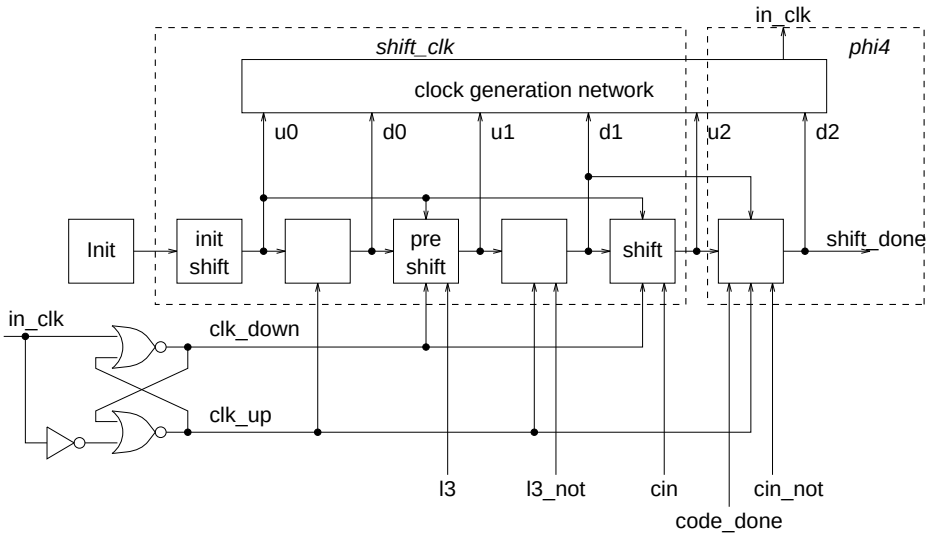

Figure 10. Shift Control: Init-Shift, Pre-Shift, and Shift

so Align will not evaluate until they arrive. A transistor controlled by the *shift_done* signal (thread (ii)) is added to the pulldown stacks, in addition to the usual enable transistor controlled by the phi0 clock, so Align will not evaluate until *shift_done* is high.

4.6 Code ROM: Generating Output Symbols

Once a Huffman code has been mapped to a particular class, the actual fixed-length output symbol must be generated. A Code ROM is used to perform the translation. It is implemented by 3 stages: Decode, ROM, and Merge. These stages implement a compact lookup table, using a 2-way decoding process.

The basic idea is to use an optimization: decode the 0 to 5 enumerating bits within a class *in parallel* with the class matching process, and then use the class matching bit as an enable signal to the decoder. Note that, even though input symbols range up to 14 bits in length, our approach avoids a full decode into 2^{14} word lines. Our ROM decoder generates only the required 256 word lines, through careful use of match classes and enumerating bits extracted from our variant Huffman encodings.

The decoders use dynamic logic and are activated by transitions on the *b(i)* inputs from the Alignment Network (starting with the least significant output bit, b(0)) and the *m(i)* inputs from the Match Logic. As a further optimization, the decoder logic is shared between similar classes. Figure 11 shows the whole decoding structure of the ROM, along with an example for classes 26, 27, and 28. The shaded boxes represent decoders which produce pair of ROM word lines. The total number of decoders is 145, only 16% more than the minimum of 125 (which occurs with maximum sharing), but 60% fewer than 240 without sharing any decoders.

One decoder output will be enabled and will drive one word line of a 9-bit wide ROM. This ROM contains the 8-bit output value and a completion bit that is slower than any other ROM value. As a performance optimization, there are actually 3 ROM arrays that have their outputs merged by additional logic outside the ROM.

5 Results

The new Huffman decoder has been designed and laid out using the Mentor Graphics design suite and the MOSIS CMOSX 0.8μ process using 3 metal layers. The layout is all custom and contains about 6100 transistors, and it contains no standard cells, except for a few inverters, NAND gates, and registers. The total area is 0.75 mm^2, which corresponds to an area of at most 3 Kbytes of instruction ROM.

5.1 Simulation Results

The performance of the design has been measured through circuit simulation using Lsim (Mentor Graphics). Since we are primarily interested in the throughput of the decoder, we measured the cycle time for decoding each symbol. In practice, the decoder operation also includes other operations, such as loading of new bytes of data into the Input Buffer from instruction memory, as the buffer empties, but we assume that the external program memory is not the bottleneck. We also assume that the Output Buffer is read by the environment before it is required for the next symbol.

Simulations using Lsim, using a CMOSX 0.8μ process, show an average throughput of 32 bits/25 ns on the output side, which is equivalent to 163 MBytes/sec, or 1303 Mbit/sec. This speed corresponds to about 889 Mbit/sec on the input side.

In more detail, simulations were obtained by decoding a real encoded program, using a 150 Kbyte sample. To simulate the performance, the critical path was divided into several smaller critical paths. The delay of each smaller

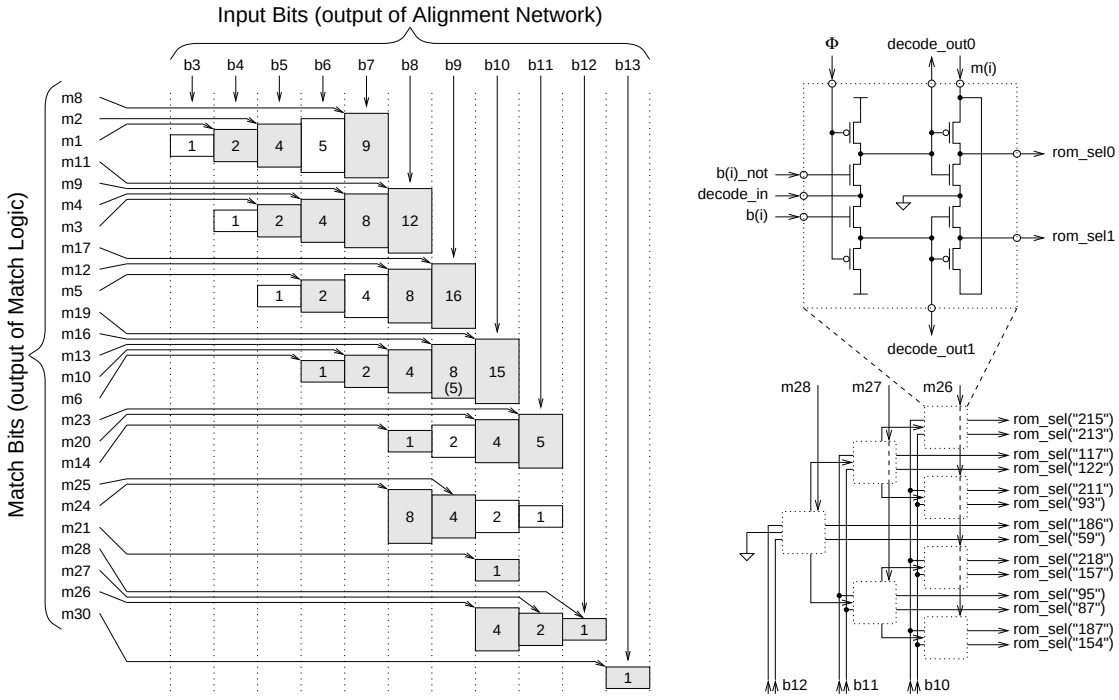

Figure 11. ROM Decoders

critical path was parametrized depending on the a particular input data. Each path was measured; the Adder was not on the critical path. In case of a shift, the critical path goes through the shifting network, all the way to shift-done. This delay can be parametrized depending on the number of shifts. In case of no shift, the delay from the end of Length to shift-done is fixed – due to control overhead. The final delays were then obtained by a software-simulating decoding process. For each cycle, the program computes the circuit conditions, determines the critical path and adds up the paramerized delays. This process gives cycle times accurate to about 0.1 ns.

Results on the sample input stream are shown in Figure 12. Decode cycles for individual input symbols ranged from 4.50 ns to 14.37 ns. There are two large peaks around 4.5 ns and 6.3 ns. The mean cycle time is 6.14 ns is 134% faster than the worst-case cycle time. This cycle time corresponds to a throughput rate of 32 bits/25 ns on the output side, which is equivalent to 163 MBytes/sec, or 1303 Mbit/sec. Since the average input symbol length is 5.46 bits for the test sample, this represents performance in excess of 889 Mbit/sec on the input side.

As indicated in Section 2, this performance is estimated as faster than other comparable synchronous Huffman decoders, after normalizing for technology and voltage. At the same time, the area of the design is 5-10 times smaller than most synchronous designs.

Figure 13 shows the distribution of decoding cycle times for 32-bit instructions. This represents the rate at which 32-bit instructions can be delivered to the instruction cache or directly to the processor.

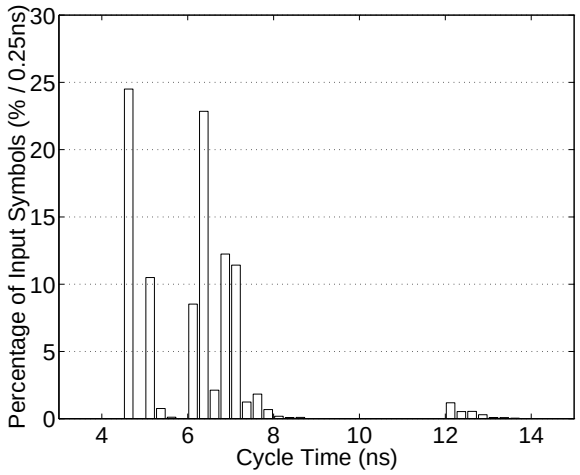

Figure 12. Distribution of cycle times for decoding individual symbols

Finally, Figure 14 shows the distribution of decode times for full 32-byte cache lines.

All of these histograms indicate a wide range of data-dependent cycle times, where the asynchronous decoder operates at a mean rate which is significantly better than worst-case.

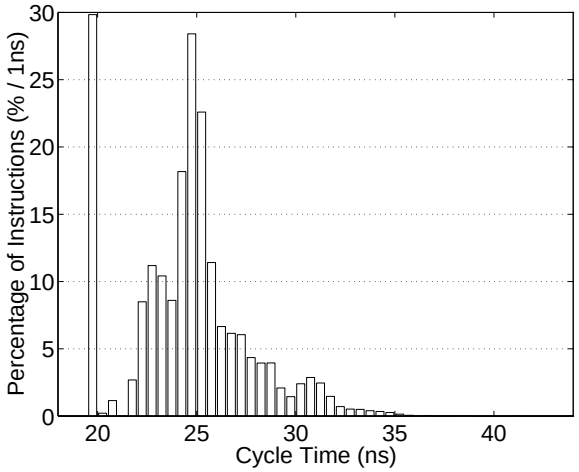

Figure 13. Distribution of cycle times for decoding 32-bit instructions

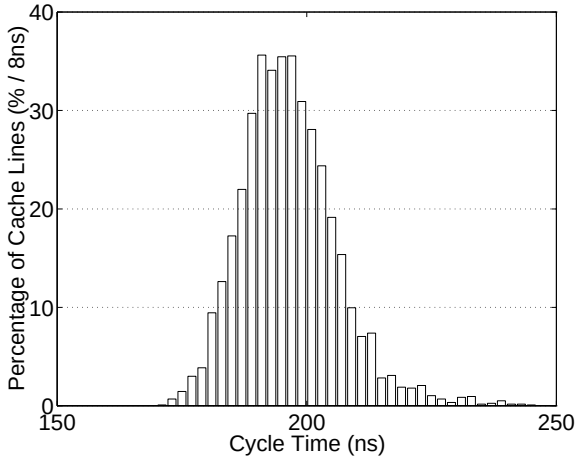

Figure 14. Distribution of cycle times for decoding 32-byte cache lines

5.2 Area Evaluation

As indicated, our design is significantly smaller than most comparable synchronous designs. In particular, our architecture avoids special optimizations for rare large shifts; instead, these are implemented as two separate 1-byte shifts in sequence. As a result, only a small shifter (in Align stage) and small adder are needed. Furthermore, since only 1-byte shifts are performed at any time, no MUXes are required to select one of several byte-shift amounts. Thus, we obtain a considerable savings in hardware by not optimizing for worst-case codes.

Admittedly, some of the area benefits of our design are due to a number of low-level optimizations, which would not appear in some commercial designs which use off-the-shelf parts. In fact, our earlier design [1] can be directly modified to handle a global synchronous clock, instead of an asynchronous clocking signal. That is, our architecture might be usable as a *synchronous* implementation, with comparable area (under 1 mm^2). However, a synchronous version would have significantly worse performance (approximately 14 ns per cycle), since a worst-case clock would be required, so no advantage could be obtained from data-dependent variations (see Figure 12). [5]

In sum, our architecture is an excellent match for asynchronous design, providing low area, while at the same time providing opportunities to obtain a significant performance benefit due to data-dependent variation.

[5] A recent approach has been proposed, called *telescopic units* [2], which allows variable-speed synchronous operation. The idea is to complete operation in 1 cycle for typical data, but in 2 cycles for worst-case data. While this approach might be used here for a synchronous Huffman decoder, we expect that the overhead of set-up, hold, decision logic, managing precharge for both scenarios, etc., at a high rate (6.14 ns typical cycle time), would be quite difficult.

5.3 Sample Simulation

It is also useful to consider the waveforms for an actual simulation, shown in Figure 15. The clocking, or precharge, signal for each stage is denoted *phi-i*, which is also a completion signal of a later stage (see dotted arrows in Figure 4). Signal *phi0* is the Align clock (completion signal from Length ROM, which is used as the completion signal of Match stage; see Section 4.3); *phi1* is the Match clock (delayed completion signal from Length ROM); *phi2* is the Length clock (completion signal from Carry stage); *phi3* is the Carry clock (completion signal from Sum stage); and *phi4* is the Sum clock (completion signal from Align stage). Signal *in_clk* is generated by the shifter, to implement a byte shift, and *shift_done* is the corresponding shift completion signal. Signal *code_done* indicates Code ROM completion (from Merge stage).

Initially, all *phi-i* are 1, indicating precharge released (ready to evaluate), and the Offset is 0. For the simulation, the first Huffman code has a length of 7 bits. Next, *in_clk* is asserted (26 ns), indicating a byte shift (this is a final initialization step to set up the Input Buffer, before steady-state processing). The falling sequence of *phi4*, followed by *phi0* through *phi3* signals, indicates completion of stages Align through Sum. At this point, a new Offset value (7) is produced. There is no shift; therefore, the first cycle is complete.

Note the zero-overhead operation: once Sum is complete (*phi3* is 0) and Shift is done (*shift_done* is 1), the next iteration can begin. At this point, Align has just released precharge (*phi0* high) so there is essentially no overhead. Also note how the Code ROM and Merge completion signal (*code_done* high) enables the shift completion signal (*shift_done* is 1), illustrating another synchronization of threads (see Figure 4).

In the second cycle, the new input symbol has length

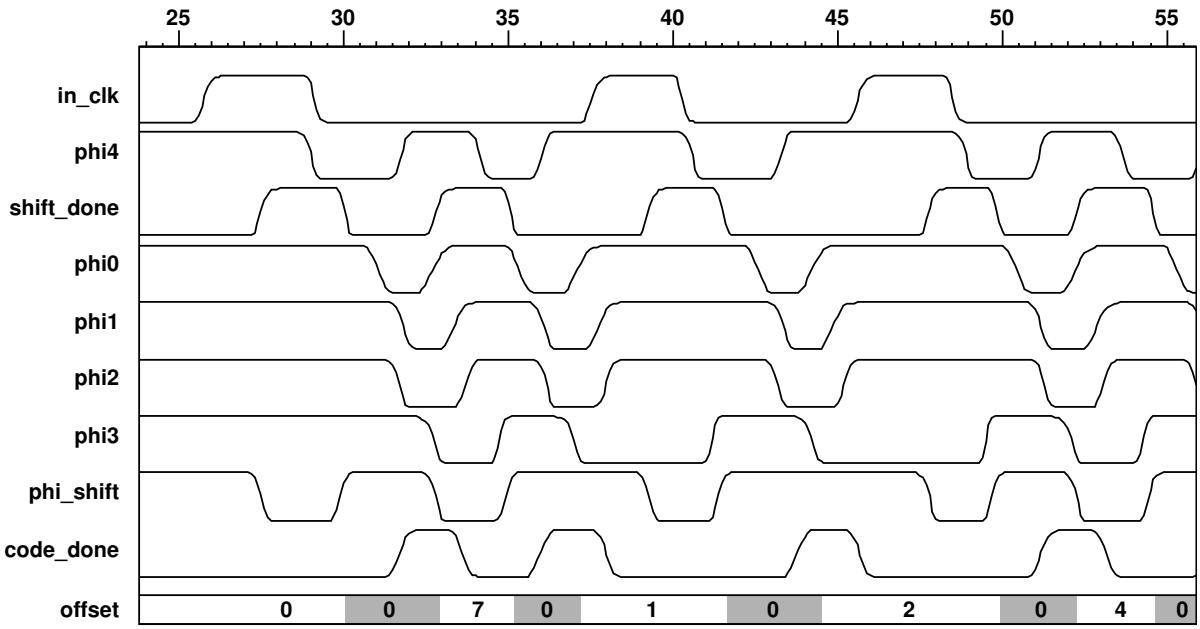

Figure 15. Waveforms from one decoder simulation

2. As a result, there is a 1-byte shift and a 1-bit new Offset ($7 + 2 = 9$). The *in_clk* cycle (starting at 38 ns) indicates the 1-byte shift operation. Note the highly-variable *code_done* waveform, which indicates the data-dependent generation and writing of individual byte output symbols into the Output Buffer.

The remaining input symbols in the simulation have bit lengths of 9 and 2, resulting in Offsets of 2 and 4, respectively.

5.4 Bottlenecks in the New Architecture.

While the new architecture achieves a large performance improvement, simulations also indicate that some bottlenecks remain.

First, stages are not entirely balanced, so zero-overhead operation is sometimes violated. As an example, consider the Sum stage in Figure 15. The stage's clock is *phi4*. Signal *phi2* goes low (32 ns), which is the done signal for Carry. Therefore, some 0.5 ns earlier, the Carry outputs were available. However, Sum cannot evaluate until its precharge is released (*phi4* goes high). Therefore, the inputs to Sum must wait for over 0.5 ns, before evaluation begins. Of course, when long shift operations occur, the balance is unavoidably skewed.

Second, the new shift optimizations were not as effective as we expected. As indicated, both Pre-Shift and Shift are enabled early (by Length ROM and Carry, respectively). However, the actual benefit of this optimization is minor. First, Pre-Shift is only enabled one gate-delay before Shift, and this benefit is lost by an extra delay for Pre-Shift in the shift sequencer (see Figure 10). Second, the current overheads of shifting still dominate.

Therefore, while our architecture achieved a significant improvement, there are still bottlenecks that merit further attention.

6 Comparison with a Previous Asynchronous Design

We now compare our new design with an earlier asynchronous decoder design presented in [1]. Both circuits were designed using the same design tools and technology (0.8μ); the earlier design also was fabricated. Both designs have the same area (0.75 mm^2) and the same number of transistors (6100). However, the new design is *83% faster* than the old design.

Architectural Comparison. Several components in the two designs have the same basic implementation: Match; Decode; Code ROM and Merge; and Input and Output Buffers. The Align stage is similar, but is modified in the new design to incorporate multiple synchronization signals.

However, there are a number of significant differences. First, the current organization into a self-timed ring, operating under local control, is entirely new. Similarly, the optimized partitioning of shifting into 3 stages, and the early scheduling of shifts (where Length controls Pre-Shift, and Carry controls Shift), as well as other aspects of the schedule, are new. Also, the design of the shift sequencer circuit is entirely new, as is the design of Offset as a dynamic stage. Finally, our new design uses a 3-stage carry-lookahead adder, where P/K/G Generation is folded directly into the Length ROM bit lines; the goal of splitting the adder into several stages is facilitate low-overhead op-

eration of the self-timed ring. In contrast, the old design used a simple ripple-carry adder.

A block diagram of the old asynchronous architecture is shown in Figure 16. The architecture is organized using a *2-cycle operation* controlled by *global synchronization*. In the first cycle, all dynamic blocks are evaluated: Align, Match, Code ROM, Length ROM and Adder. In the second cycle, all dynamic blocks are precharged, while static shift operations occur, if any. Effectively the asynchronous design generates its own synchronizing global clock, based on the completion of all the events in each cycle.

Performance Analysis. Figure 17 indicates the breakdown of timing of the stages in the old architecture. Many stages exhibited a wide-variation in data-dependent operation. However, the benefits of this variation were lost due to a large *fixed overhead:* the generation of $\Phi+$ and $\Phi-$ clocking signals. The overhead due to the generation and distribution of these clocking signals was 3.85 ns per iteration. In addition, in the second cycle, when no shift occurred, the *precharge time* also contributed 1.67 ns of fixed overhead, where no other useful work was performed. Finally, the ripple-carry adder, under worst-case inputs, provided extra overhead.

The key contribution of our new architecture is to eliminate these fixed overheads. By structuring the decoder into an iterative ring, which (i) uses *local synchronization* between stages (rather than global synchronization and clocking distribution), and (ii) *hides precharge time* (using a zero-overhead approach), these fixed delays were largely eliminated. In the old architecture, individual symbol decode cycles ranged from 9.23 ns to 19.66 ns, with a mean cycle time of 11.23 ns. In the new architecture, the range is 4.50 ns to 14.37 ns, with a mean of 6.14 ns. These numbers suggest that we have basically removed a fixed overhead of roughly 4.5 and 5.5 ns per iteration.

7 Conclusions

We have introduced a new architecture and implementation for an asynchronous Huffman decoder, for a compressed-code embedded processors. The decoder achieves a high-speed decode rate with very low area overhead. It is estimated to have higher throughput than any comparable synchronous Huffman decoder (after normalizing for feature size and voltage), yet is 5-10 times smaller than most synchronous designs. The performance is also 83% faster than our previous asynchronous Huffman decoder using the same technology.

The design illustrates the advantages of self-timed rings for variable-speed iterative computations. Interestingly, while many high-performance Huffman decoders are pipelined, pipelining was unnecessary in our case.[6] The ring often operates with a near-zero control overhead.

In the future, it would be interesting to explore if greater benefits could be obtained by pipelining, or if the control overhead of pipelining would outweigh any net throughput improvement.

[6] Note that an iterative ring structure effectively has pipelined precharges, *i.e.,* they are overlapped with evaluations. Here, though, we refer to true pipelining of several simultaneous evaluations.

References

[1] M. Benes, A. Wolfe, and S.M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proceedings of the 17th Conference on Advanced Research in VLSI*. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[2] L. Benini, E. Macii, and M. Poncino. Telescopic units: Increasing the average throughput of pipelined designs by adaptive latency control. In *ACM/IEEE Design Automation Conference*, pages 22–27, June 1997.

[3] G. Birtwistle and A.L. Davis. *Asynchronous Digital Circuit Design*. Springer-Verlag, 1995.

[4] S. Choi and M. Lee. High speed pattern matching for a fast huffman decoder. *IEEE Trans. on Consumer Electronics*, 41(1):97–103, February 1995.

[5] M. Matsui *et al.* 200 mhz video compression macrocells using low-swing differential logic. In *ISSCC*, pages 76–77, 1994.

[6] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, S. Temple, and J.V. Woods. The design and evaluation of an asynchronous microprocessor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 217–220. IEEE Computer Society Press, October 1994.

[7] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver. Amulet2e: An asynchronous embedded controller. In *Async97 Symposium*. ACM, April 1997.

[8] J.D. Garside, S. Temple, and R. Mehra. The amulet2e cache system. In *Async96 Symposium*. ACM, April 1996.

[9] R. Hashemian. Design and implementation of a memory efficient huffman decoding. *IEEE Trans. on Consumer Electronics*, 40(3):345–51, August 1994.

[10] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IEEE*, 40(10):1098–1101, September 1952.

[11] J. Kessels and P. Marston. Design asynchronous standby circuits for a low-power pager. In *Async97 Symposium*. ACM, April 1997.

[12] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *IEEE International Conference on Computer Design*, pages 270–277, October 1994.

[13] S.Y. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In W.J. Dally, J.W. Poulton, and A.T. Ishii, editors, *Proceedings of the 16th Conference on Advanced Research in VLSI*, pages 272–285. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[14] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The design of an asynchronous microprocessor. In *1989 Caltech Conference on Very Large Scale Integration*, 1989.

[15] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: design of a quasi-delay-insensitive microprocessor. *IEEE Design and Test*, 11(2):50–63, Summer 1994.

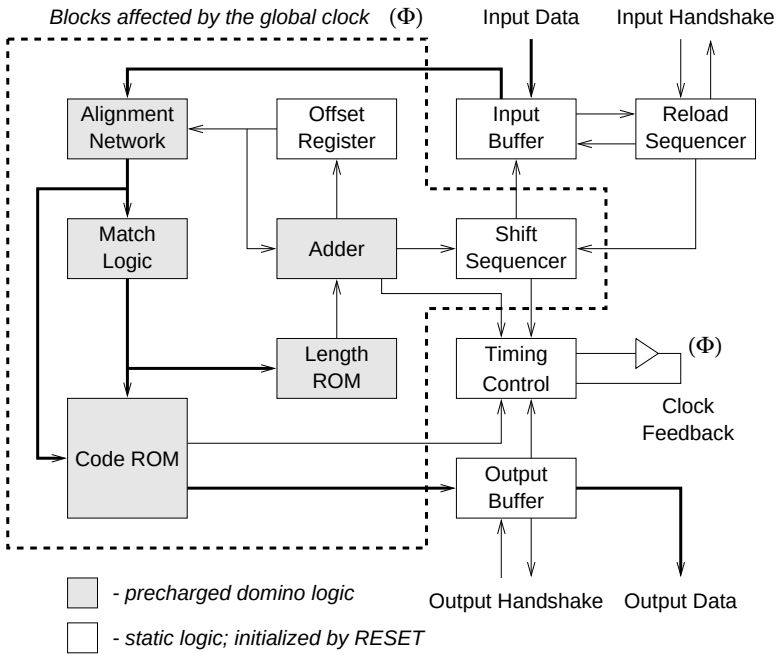

Figure 16. Previous Asynchronous Decoder: architecture and signals

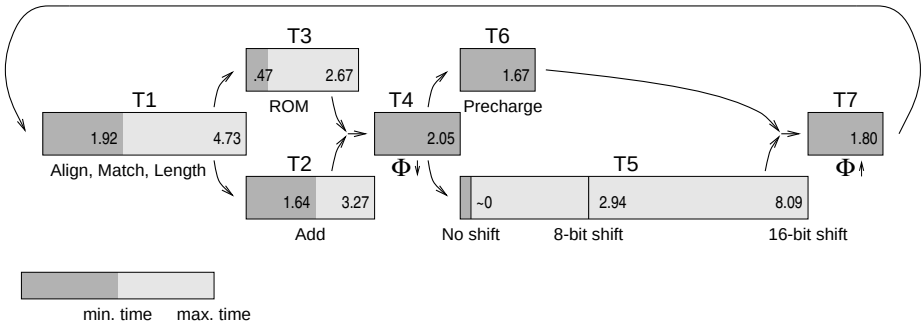

Figure 17. Previous Asynchronous Decoder: breakdown of the cycle

[16] L.S. Nielsen and J. Sparso. A low-power asynchronous data path for a fir filter bank. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, pages 197–207. IEEE Computer Society Press, November 1996.

[17] H. Park, J. Son, and S. Cho. Area efficient fast huffman encoder for multimedia applications. In *ICASSP*, pages 3279–3281, 1995.

[18] M.K. Rudberg and L. Wanhammar. New approaches to high speed huffman decoding. In *ISCAS*, pages 149–152, 1996.

[19] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, thumb and the ARM7TDMI. *IEEE Micro*, 15(5):22–30, October 1995.

[20] J. Sparso and J. Staunstrup. Design and performance analysis of delay insensitive multi-ring structures. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 349–358. IEEE Computer Society Press, January 1993.

[21] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design and Test of Computers*, 11(2):22–32, Summer 1994.

[22] T.E. Williams. Self-timed rings and their application to division. Technical Report CSL-TR-91-482, Computer Systems Laboratory, Stanford University, 1991. Ph.D. Thesis.

[23] T.E. Williams and M.A. Horowitz. A zero-overhead self-timed 54b 160ns CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.

[24] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC processor. In *25th Annual International Symposium on Microarchitecture*, pages 81–91, December 1992.

[25] Y. Yoshida, B.-Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa. An object code compression approach to

embedded processors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 265–268. ACM, August 1997.

[26] K.Y. Yun, P.A. Beerel, A.E. Dooply, J. Arceo, and V. Vakilotojar. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. In *Async97 Symposium*. ACM, April 1997.