A High-Speed Asynchronous Decompression Circuit for Embedded Processors

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

Andrew Wolfe†
Department of EE
Princeton University
Princeton, NJ

Steven M. Nowick‡
Department of CS
Columbia University
New York, NY

Abstract

This paper describes the architecture and implementation of a high-speed decompression engine for embedded processors. The engine is targeted to processors where embedded programs are stored in compressed form, and decompressed at runtime during instruction cache refill.

The decompression engine uses a unique asynchronous variable decompression rate *architecture to process Huffman-encoded instructions. The resulting circuit is significantly smaller than comparable synchronous decoders, yet has a higher throughput rate than almost all existing designs. The 0.8μ layout is all full-custom and contains predominantly dynamic domino logic. The top-level control, as well as several small state machines, are implemented using asynchronous logic. The design operates without a user-supplied clock.*

Simulations using Lsim show average throughput of 32 bits/45 ns on the output side, corresponding to about 480 Mbit/sec on the input side. The chip has been manufactured by MOSIS; tests show that the asynchronous implementation operates correctly, with an average throughput exceeding simulations: 32 bits/39 ns on the output side, corresponding to about 560 Mbit/sec on the input side. This speed is acceptable for our application. The area of the design (excluding the pad-frame overhead) is only 0.75 mm^2. The design is the first fabricated chip for an instruction decompression unit for embedded processors.

1: Introduction

Embedded systems incorporate microprocessors or microcontrollers to implement communication and control functions for consumer electronics products. These systems are sensitive to many design constraints, including limits of size, weight, power consumption and cost. Many interesting design problems for embedded systems involve optimizing these properties for high-volume products, such as consumer electronics. Since the programs in embedded systems are generally stored in ROM, the size of the program has a direct impact on the per-unit cost of the device.

In previous work, a method was proposed whereby embedded programs are stored in compressed form and decompressed at run time during instruction-cache refill [Wolfe92, Kozuch94]. Using a Huffman encoding scheme, a compression ratio of 73% was reported

* While at Department of Electrical Engineering, Princeton University, Princeton, NJ

†This work was funded in part from NSF under award MIP-9408462 and by an AT&T foundation gift.
‡This research was funded in part by an NSF CAREER Award MIP-9501880 and by an Alfred P. Sloan Research Fellowship.

for the MIPS instruction set. In this paper, we describe the architecture and implementation of a *high-speed decompression engine* for such applications. There are two key requirements for such a decompression circuit. First, it must be extremely small to be widely applicable. Reducing the area requirements for program memory would be futile if all of the saved space were consumed by the decompression logic. Second, the circuit must also provide very high performance to avoid slowing down the processor.

Our decompression engine uses a unique *variable decompression rate* architecture, as well as some innovative asynchronous timing mechanisms, to obtain much smaller chip area than previous designs, with higher throughput than nearly all existing designs. In addition to providing an excellent solution to a specific application requirement, this design demonstrates the benefits of asynchronous circuits in a practical system.

Asynchronous design provides numerous opportunities to optimize a design for the most common inputs [Nowick96, Nowick97, Davis95]. This feature is particularly important in the case of an entropy code like a Huffman code, where the distribution of input values is essentially known. An asynchronous methodology also allows the design of small, high-speed circuits without the risk or overhead associated with high speed clocks. Recently, a number of asynchronous chips have been successfully fabricated, both for microprocessors and DSPs [Furber94, Furber97, Kessels97, Martin89, Nielsen96, Yun97].

Our proposed decompression engine achieves its cost/performance advantage over existing designs through a combination of architectural improvements as well as aggressive circuit design. Novel features of this chip include :

- A variable-input-rate/variable-output-rate asynchronous architecture that allows the decoder timing to be optimized for the most common cases while concurrently processing many bits from the input data stream
- A simplified symbol-length detection circuit that minimizes area
- A customized ROM implementation that overlaps output-symbol lookup with symbol-length detection and minimizes area of the decoding logic
- Asynchronous circuit timing using several different synchronization mechanisms
- Use of dynamic domino logic circuits with few stages for high speed and small area

The resulting circuit has been implemented in 0.8μ CMOS using the MOSIS CMOSX design rules. The complete decoder including input and output buffers for 32-bit I/O interfaces occupies a net area of 0.75 mm^2, which corresponds to the area of at most 3 Kbytes of instruction ROM. Simulated performance based on a 150 Kbyte program sample exceeds 480 Mbit/sec in decoding the input stream, while the fabricated chip operates at 560 Mbit/sec.

The design is significantly smaller than comparable synchronous decoders, and yet has a higher throughput rate than almost all existing designs. It is the first fabricated chip for an instruction decompression engine for embedded processors.

The paper is organized as follows. Section 2 discusses related work on the design of synchronous Huffman decoders. Section 3 presents background on the proposed compression scheme, and also motivates our use of an asynchronous approach. A basic overview of our decompression engine architecture is presented in Section 4, followed by detailed descriptions of the components and circuit implementations in Section 5. Simulation and fabrication results are presented in Section 6. Section 7 discusses the tradeoffs between an asynchronous and synchronous implementation, and Section 8 presents conclusions.

2: Related Work

Decompression of Huffman codes is not a new problem: many previous synchronous implementations of Huffman decoders have been described in the literature. Virtually all recent practical implementations of Huffman decoders are 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.

Rudberg [Rudberg96] presents a design for a constant-input-rate synchronous Huffman decoder. His design uses aggressive pipelining to break the critical dependency loop in length detection, thereby allowing the implementation of a constant-input-rate/variable-output-rate decoder without complex state machines. However, while the circuit allows a high clock speed, it can only process one input bit per cycle and thus only supports a 120 Mbit/sec decode rate. The area of this circuit is unknown.

Some basic ideas for partitioning the output-symbol ROM into multiple small ROMs and for extracting symbols from the input stream with a barrel shifter are described in [Hashemian94] and [Choi95]. Similar mechanisms are employed in our design. These concepts have also been included in a decoder for the MPEG-2 DCT coefficient table that has been implemented and described in [Park95]. The circuit 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. The authors claim a peak performance of 680 Mbit/sec, but this is based on the decoding of maximal-length 17-bit codewords at 40 MHz. Since the worst possible 114 symbol Huffman code has an average symbol length of 7 bits and a more typical code would have an average symbol length of no more than 5 bits, sustained performance in the 200–250 Mbit/sec range is more realistic.

Two of the fastest recent decoders were developed in Japan. Both implementations integrate the VLD decoder with IDCT transform to completely decode the video stream. The first design, at Osaka University [Onoye95], uses pipelined design with a separate stage for input shifting, length detection and symbol decoding. The VLD layout using ASIC libraries occupies about 5 mm^2 in a 0.6μ process. The second design, at Toshiba Microelectronics [Matsui94], uses one stage for length and symbol lookup (details were not given). To compensate for slow length detection, it uses a two-stage pipelined shifting scheme that removes the adder delay from the critical path. The 28k-transistor VLD macro occupies about 5 mm^2 in a 0.8μ process (0.5μ nFETS are used). Both of these chips operate at 3.3 V. Since they include both VLD and IDCT blocks, it is unclear whether the speed is limited by the VLD critical path. The Osaka University chip is clocked at 100 MHz and the Toshiba chip at 200 MHz. After scaling to our conditions (5 V and 0.8μ process), the normalized rates are 95 MHz and 150 MHz, respectively. The latter speed is achieved mainly by using aggressive but area-expensive circuit techniques like pass-transistor and differential amplifying logic, which we have not used. In comparison, our asynchronous chip has an output rate of 32 bits/39 ns, which is roughly 103 MHz. At the same time, our decoder area is roughly 5 times smaller than each of these designs.

Finally, Wei and Meng [Wei95] introduce a JPEG Huffman decoder which operates at 40 MHz, but can be pipelined at 80 MHz. This work focuses on programmable codes, rather than the fixed codes which we consider. The resulting chip area is 11 mm^2 in a 1.2μ process, which is over 10 times larger than ours.

3: Background and Motivation

3.1: Compression Scheme

The proposed compression scheme relies on a standard entropy code [Huffman52] that represents fixed-length symbols from a source alphabet as variable-length code symbols. The key challenge in the development of a code compression scheme for existing microprocessor architectures is that the system must run existing programs correctly. This is accomplished by decompressing a program *as it is fetched* into the instruction cache. As a result, the processor core sees only uncompressed program code [Wolfe92, Kozuch94].

Since the instruction cache only holds a small fraction of the program at one time, it is not possible to decompress the entire program at once; therefore, a block-oriented compression scheme is required. The experiments we have performed are based on compressing 32-byte cache lines into smaller byte-aligned blocks. Compression takes place at program development time, and therefore compression time is immaterial. Decompression time, however, *directly* impacts the cache refill time and hence performance. The preferred encoding method uses a fixed Huffman code, based on an aggregate distribution of 8-bit input symbols determined through the analysis of a large set of typical programs for a given processor architecture. This code is not optimal for any given program, but is quite close and it allows the code to be hardwired into the decompression hardware, rather than being stored along with the program.

In principle, the decoding of Huffman codes is a very simple task. Huffman codes are entropy-based, prefix codes that replace fixed-length symbols with variable-length symbols to represent the same information with fewer total bits. The original source data is represented as a sequence of fixed-length symbols. The full set of symbols is called the input alphabet. We experimented with input alphabets ranging from 4 to 16 bits per symbol for this application. Although 16-bit symbols would provide the best compression, 8-bit symbols performed nearly as well and can be decoded with far less hardware.

The frequency of each symbol in the input data is measured, to determine the probability of any given symbol occurring in a randomly selected sample from the input data. This model assumes that the probability of any symbol occurring is independent of other symbols in the data stream. Naturally, if frequently occurring symbols are represented with few bits, and rarely occurring symbols are represented with more bits, the total number of bits required can be minimized. As an example, Figure 1 gives a sample histogram of Huffman code length versus the percentage of input symbols, where the symbols are statically binned into different classes. Figure 2 shows the corresponding histogram based on the actual frequency of occurrence of the input symbols in a given program. Using a Huffman encoding scheme, a compression ratio of 73% was reported for the MIPS instruction set [Wolfe92, Kozuch94].

The number of bits required for each symbol is approximately $\log_2(1/P_{\text{symbol}})$. In order to make it easier to decode this representation, *prefix codes* are used for the output alphabet. Prefix codes have the property that appending additional bits to a valid output symbol never produces another valid output symbol. Therefore, by scanning the coded data one bit at a time until we find a valid symbol, we can immediately decode that symbol and remove it from the data stream. The actual code that was used for this implementation is based on measurements of byte frequencies from programs for the MIPS architecture. The resulting code includes symbols ranging from 2 to 14 bits in length [Wolfe92, Kozuch94].

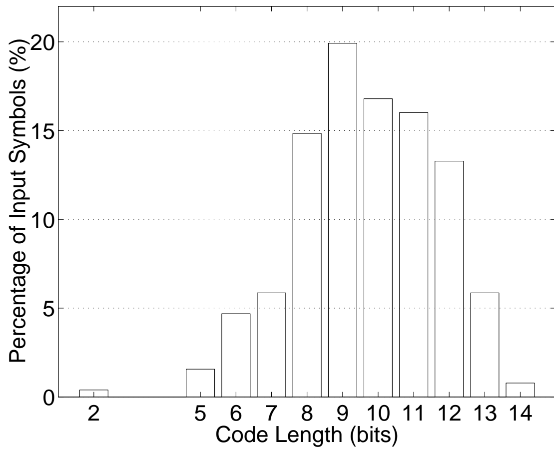

Figure 1. Static code distribution

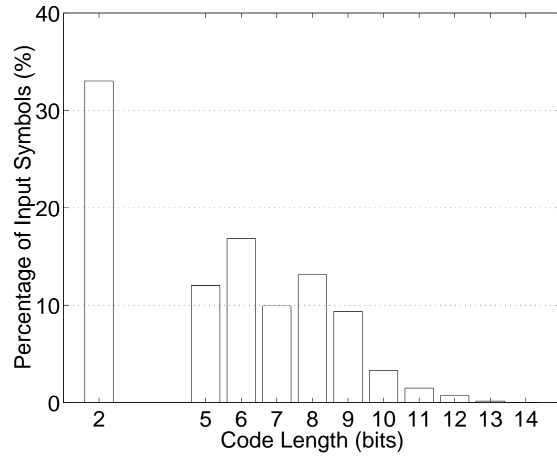

Figure 2. Dynamic code distribution

3.2: Motivation for an Asynchronous Architecture

There are two standard synchronous approaches to decoding Huffman encoded data (see [Rudberg96]).

In a basic *constant-input-rate scheme*, the input data stream is processed at a rate of one bit per cycle by traversing a Huffman code tree through the use of a finite state machine. To achieve 480 Mbit/sec performance using this type of design would require a 480 MHz clock, introducing many very difficult high-speed circuit problems. In fact, it is unlikely that a state machine of adequate complexity can be designed to run at this speed in 0.8μ CMOS. However, as an optimization, the very high-speed clocks required can be eliminated by combining multiple state transitions into a single cycle. This reduces the required clock frequency but increases the complexity and area of the decoder approximately exponentially with respect to the increased performance per clock.

The other commonly-used approach is a *constant-output-rate scheme*. In this design, a portion of the input data stream, at least as large as the longest input symbol, is translated into an output symbol on each cycle. This approach requires a complex shifting network to remove variable length symbols from the input data stream, as well as a more complex symbol detection circuit. The disadvantage of this approach is that the length of the critical paths is dominated by the time to detect and decode the longest input symbol. In the case of our test codes, the 14-bit symbols represent only about .01% of the input data; thus, the vast majority of cycles are limited by a very infrequent worst-case path. Furthermore, an input data buffer and shifting network that can sustain 14 bits of new input data is far more expensive than one that can sustain the average rate of 5–6 bits of input per output symbol.

The compromises required for a high-speed/low area implementation in each of these methods led us to examine an asynchronous approach. Our new architecture has similarities to the constant-output-rate synchronous decoding scheme, but is in fact modified to support both a *variable-input-rate* and a *variable-output-rate*. Our design is optimized so that common symbols can be decoded quickly, while less common symbols require longer combinational logic delays, multiple state transitions and reuse of some hardware. This approach balances good performance with compact area.

4: Overview: Asynchronous Decoder Architecture

This section presents an overview of our asynchronous decoder architecture. Further details on the circuit implementations are presented in Section 5.

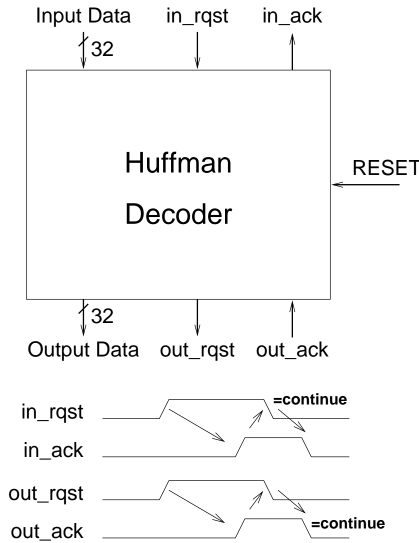

Figure 3. Decoder overview

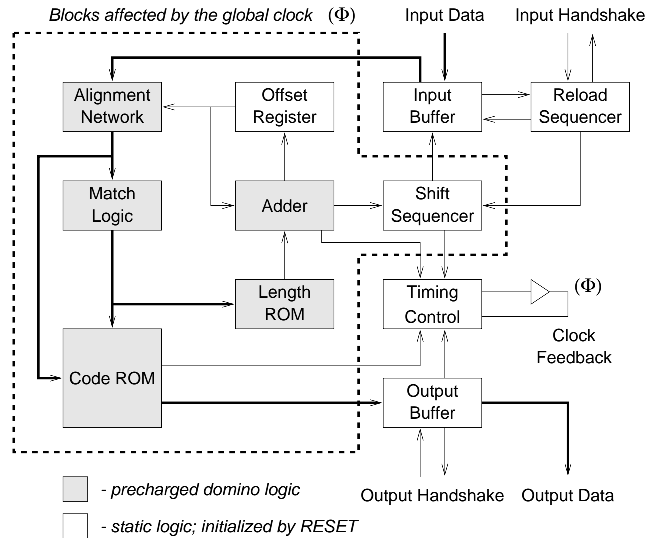

Figure 4. Decoder components and signals

The decoding of a Huffman-encoded program requires a sequence of interfacing and code translation steps. First, data must be fetched from memory into an input buffer as 32-bit words. Next, the input stream must be parsed into variable-length tokens. Each token must be translated into the output byte which it represents. The output bytes must then be buffered into 32-bit output words. Finally, the input symbol bits must be removed from the input stream and the remaining input data must be realigned in the input buffer. The input buffer must then be refilled as space becomes available, to insure that there is adequate data for the next symbol. As many of these steps as possible must be done in parallel to obtain good performance.

Figure 3 gives a top-level view of our decoder module as a black box. Input data is fetched from memory using a simple 4-phase asynchronous handshaking protocol [Davis95]. Similarly, output data 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.

Figure 4 shows the major components of the decoder and their interaction. The *Input Buffer* holds 7 bytes of data. The lower 4 bytes receive new data which are then shifted into the top 3 bytes to be processed. This large buffer allows new data to be fetched while older data is being decoded. The data in the Input Buffer is shifted in 1-byte increments. Therefore, as symbols are removed from the input stream, the remaining input data in the buffer may be misaligned. Realignment is implemented using an *Alignment Network*. The *Offset Register* holds the correct shift offset, thereby extracting an aligned 14 bits (the size of the largest symbol) from the Alignment Network for processing.

The input data is then processed by *Match Logic*, which categorizes the token into one of 31 classes within the Huffman code tree. Each class contains only symbols of a given length, although there may be several classes that contain codes with the same length.

The output *Code ROM* is a compact lookup table, which is used to generate the output symbol. The symbol is generated by a 2-way decoding process. Each match class produces a unique *enable signal* for the corresponding portion of the Code ROM. However, in parallel with the processing of the Match Logic, the aligned input data is forwarded directly to the ROM decoders, which identify one *potential* output symbol for each class. When the Match Logic finally indicates the correct class, it enables the final stage of exactly one of the ROM decoders, which then generates the correct output symbol. The symbol is then stored in the *Output Buffer*.

The class selected by the Match Logic is also used to determine the *length* of the input symbol, by using a small *Length ROM*. The length indicates how many bits of input data to retire from the input data stream. This length is added to the current offset in the Offset Register, to determine the new offset for the next token, which is stored in the Offset Register. If the offset exceeds the range of the Alignment Network, a carry is generated by the Adder, and the Shift Sequencer is instructed to shift data in the Input Buffer by either one or two bytes.

As shown in Figure 4, many of the components are implemented in precharged domino logic (gray modules), while some are implemented in static logic (white modules). The use of precharged domino logic greatly improves speed and reduces area of the design.

A global synchronization or clocking signal Φ, is generated, to synchronize the system. The signal is generated *asynchronously*, based on the actual completion of the various modules. The logic within the dotted region in Figure 4 is controlled by Φ. When Φ is low, the domino logic precharges. When Φ goes high, the logic evaluates its inputs, and finally generates a new output symbol and an offset value. Completion signals from the Code ROM and the Adder are then combined, to indicate to the clock generation circuit that the evaluation phase is complete, and that the next precharge phase may begin. During the precharge phase, static operations may occur; in particular, the data in the Input Buffer is shifted by the Shift Sequencer, if necessary. The precharge phase ends when shifting is complete, the dynamic logic is precharged, and the Output Buffer has room for another symbol. The decompression engine produces one output symbol per Φ cycle, but the length of that cycle varies and may include zero, one, or two Input Buffer shift cycles during the precharge phase.

5: Detailed Implementation

This section provides details on each of the decoder's components.

5.1: Input Buffer and Alignment Network

Extracting new symbols from the input data stream is the critical path in the decoding process. The next input symbol cannot be located until after the length of the current symbol is determined. We use a 2-step extraction process, to reduce the hardware overhead.

The Input Buffer consists of 7 registers, as shown in Figure 5. Four of the registers are 9-bits wide, where each holds 8 bits of data and 1 status bit. A new 32-bit data word is loaded, in parallel and asynchronously using in_load, into these four registers. In addition, a "True" value is loaded into each of the four status bits, to indicate that these registers contain valid data. The signal "Full" is the Boolean AND of these four status bits, and

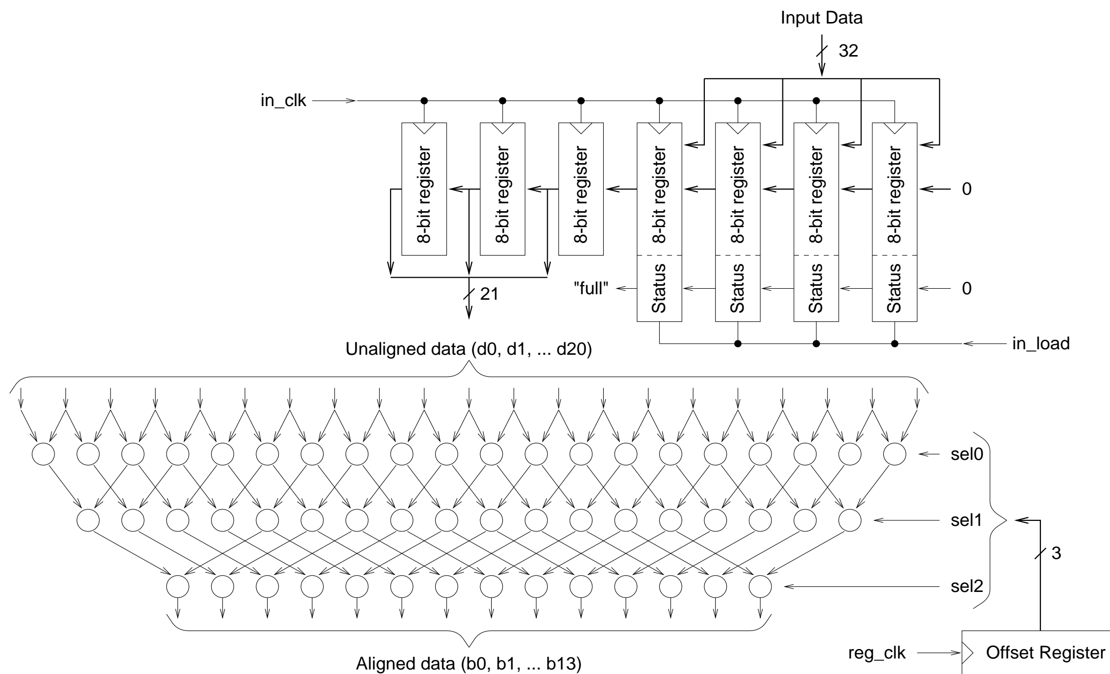

Figure 5. Input Buffer and Alignment Network

indicates that the data has been successfully loaded into these registers. The layout has been designed so that the status bits reliably reflect the worst-case timing of every bit in the register, and thus they can safely be used as completion signals.

Once data and status bits have been loaded into the rightmost registers, they are shifted into the leftmost registers for processing. The *Shift Sequencer* logic controls this shifting, initially shifting the data 3 times to move the data into the four leftmost registers. Each shift moves the data 8 bits forward in the data stream. This may result in a residual misalignment of the beginning of the next input symbol by up to 7 bits. A misalignment is corrected by the barrel shifter structure in the *Alignment Network*, shown on bottom of Figure 5. (Further details on the Shift Sequencer are presented in Subsection 5.4.)

The Alignment Network consists of 3 stages of 2–1 multiplexers implemented in hazard-free, precharged domino logic. The dual-rail data inputs come from the Input Buffer and the dual-rail select inputs from the Offset Register. The outputs are also dual-rail and thus each bit is completed when one of the two wires for one bit of the output falls to zero.

As data is consumed from the Input Buffer and processed, the input data is shifted further. Each shift operation shifts a “False” value into the status bit of the rightmost register indicating that there is no longer valid data in that register. When all four status bits have the value “False”, the signal “Empty” is asserted to the *Reload Sequencer* and more data is loaded from external memory. A completion signal for individual shifts is derived from a delayed sample of the shift clock. The outputs of this buffer, feeding into the Alignment Network, are the 21 leftmost bits of the buffer. These signals, and their complements, are provided to the Alignment Network and must be stable before the precharge phase ends.

5.2: Huffman Decoding: Symbol Assignment and Match Logic

Once the bits that contribute to the next code word are extracted from the data stream, they must be parsed in order to determine the length of the next input symbol and to extract the bits of that symbol. In order to simplify this procedure, we have taken advantage of some of the flexibility of Huffman codes. On the one hand, the length of each Huffman symbol is precisely determined by the frequency distribution of the original input alphabet. On the other hand, the actual *bit encoding* for each input symbol is flexible, as long as the prefix property and the code length requirement are maintained.

Our approach is therefore to select code words of a given length such that those code words contain as many common bits as possible. For example, note that all of the 5 bit long symbols, as shown in Table 1, contain the string 010 as the initial bits while no other symbols contain those values in those positions. This allows us to use that substring in that position to define a *class* of 5-bit symbols, and to use the remaining bits to enumerate the values in that class. In many cases, we can structure the code such that only a small number of bits need to be tested in order to determine the length. This means that we match the class of 5-bit codes by finding bit-strings that contain the string 010 in bits 0–2. Similarly we find a class of 6-bit codes that begin with the string 011 and another that begins with the string 1000.

0	00	2c	10111001	86	110111000	b6	1110111110	fd	11111001010	fa	111111100000
8f	01000	b0	10111010	43	110111001	fe	1110111111	7e	11111001011	f5	111111100001
24	01001	9	10111011	b9	110111010	e2	1111000000	65	11111001100	56	111111100010
1	01010	f8	10111100	8a	110111011	e6	1111000001	67	11111001101	d2	111111100011
10	01011	e7	10111101	6c	110111100	ef	1111000010	f7	11111001110	cd	111111100100
46	011000	a8	10111110	32	110111101	d4	1111000011	71	11111001111	f6	111111100101
25	011001	ae	10111111	a9	110111110	ce	1111000100	3a	11111010000	ed	111111100110
80	011010	88	11000000	b	110111111	7f	1111000101	9e	11111010001	5e	111111100111
8	011011	90	11000001	4c	111000000	4b	1111000110	7b	11111010010	77	111111101000
3	011100	e4	11000010	aa	111000001	4e	1111000111	6b	11111010011	f2	111111101001
21	011101	50	11000011	13	111000010	39	1111001000	6a	11111010100	97	111111101010
c	011110	2a	11000100	64	111000011	2f	1111001001	c3	11111010101	c9	111111101011
4	011111	44	11000101	d	111000100	dc	1111001010	1b	11111010110	7d	111111101100
20	100000	bd	11000110	68	111000101	45	1111001011	66	11111010111	55	111111101101
ff	100001	6	11000111	22	111000110	51	1111001100	35	11111011000	ca	111111101110
2	100010	a5	11001000	2b	111000111	b3	1111001101	4d	11111011001	e9	111111101111
af	100011	bf	11001001	a7	111001000	62	1111001110	79	11111011010	95	111111110000
c0	1001000	1c	11001010	a3	111001001	9c	1111001111	1e	11111011011	9b	111111110001
8c	1001001	8d	11001011	89	111001010	cf	1111010000	3e	11111011100	9f	111111110010
8e	1001010	38	11001100	fc	111001011	4f	1111010001	be	11111011101	fb	111111110011
84	1001011	11	11001101	ad	111001100	f4	1111010010	47	11111011110	69	111111110100
82	1001100	26	11001110	c8	111001101	52	1111010011	e1	11111011111	53	111111110101
e0	1001101	a4	11001111	23	111001110	91	1111010100	1f	11111100000	eb	111111110110
28	1001110	ac	11010000	31	111001111	99	1111010101	b7	11111100001	96	111111110111
c4	1001111	a0	11010001	87	111010000	5c	1111010110	49	11111100010	d7	1111111110000
30	1010000	5	11010010	81	111010001	c5	1111010111	33	11111100011	da	1111111110001
18	1010001	60	11010011	15	111010010	17	1111011000	6f	11111100100	d3	1111111110010
c7	1010010	2e	110101000	58	111010011	c1	1111011001	36	11111100101	bb	1111111110011
14	1010011	ab	110101001	98	111010100	7c	1111011010	e5	11111100110	d5	1111111110100
40	1010100	63	110101010	a	111010101	61	1111011011	93	11111100111	9d	1111111110101
27	1010101	29	110101011	f	111010110	b5	1111011100	f9	11111101000	5d	1111111110110
3c	1010110	92	110101100	83	111010111	b2	1111011101	1a	11111101001	9a	1111111110111
12	10101110	8b	110101101	a2	111011000	e8	1111011110	ee	11111101010	75	1111111111000
48	10101111	d8	110101110	a6	111011001	74	1111011111	76	111111010110	5f	1111111111001
42	10110000	b1	110101111	e	111011010	ec	1111100000	de	111111010111	7a	1111111111010
41	10110001	94	110110000	73	1110110110	37	11111000010	3f	111111011000	57	1111111111011
7	10110010	d0	110110001	6e	1110110111	ea	11111000011	5a	111111011001	ba	1111111111100
85	10110011	c6	110110010	2d	1110111000	d6	11111000100	5b	111111011010	3b	1111111111101
19	10110100	a1	110110011	c2	1110111001	72	11111000101	f1	111111011011	df	1111111111110
78	10110101	16	110110100	cc	1110111010	e3	11111000110	d1	111111011100	dd	11111111111110
34	10110110	b4	110110101	4a	1110111011	1d	11111000111	f3	111111011101	db	11111111111111
b8	10110111	54	110110110	bc	1110111100	d9	11111001000	cb	111111011110		
70	10111000	f0	110110111	59	1110111101	6d	11111001001	3d	111111011111		

Table 1. Rearranged Huffman code for MIPS architecture encoding

Overall, we are able to group the 256 different code words into 31 distinct classes; they are shown in Table 2 along with their qualifying bit patterns ("-" means a don't care). Every member of each class is a code word of equal length and a small number of remaining bits

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

Table 2. Match Classes

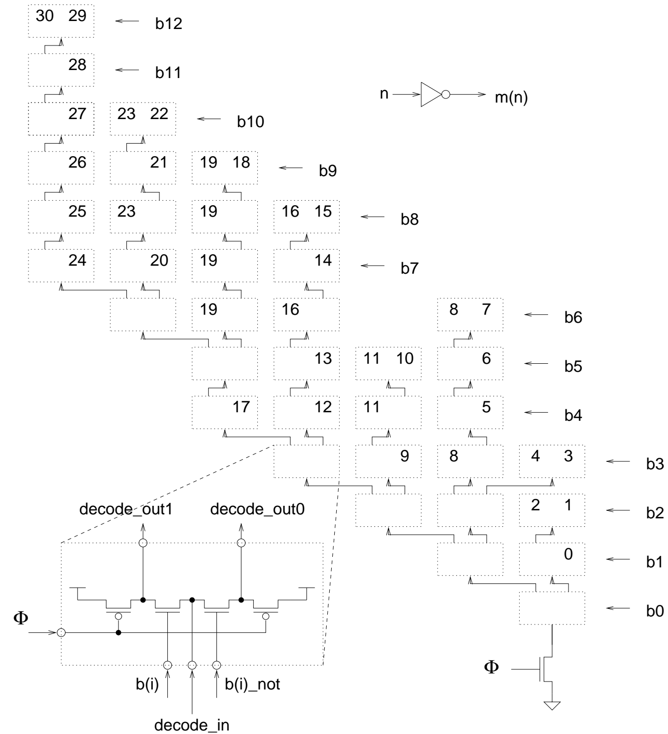

Figure 6. Class Matching logic

(represented with dots) can be used to enumerate the class members. The matching process starts from the top and examines each bit pattern until a match is found.

The *Match Logic* in Figure 6 performs the matching task. Each block represents the 4-transistor circuit shown on the bottom. Both phases of one of the input bits 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 decode_in signal is driven by a sequence of prior stages that represents a value of the preceding bits. The numbers on outputs of the tree represent the length classes. In some cases, several output are connected in a wire-or circuit to indicate a length class. Since only one class is detected, 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 common, codes are matched using the fewest levels of logic; therefore, the average response time is much faster than the worst-case.

5.3: Huffman Decoding: Generating Output Symbols

Once the code has been mapped to a particular class, the actual fixed-length output symbol must be generated. A specialized *Code ROM* construct has been developed to perform this translation. The basic idea is to decode the 1 to 5 enumerating bits[1] within a class *in parallel* with the class matching process, and then to use the class matching bit as

[1] In some cases, there are 0 enumerating bits.

an enable signal to the decoder.

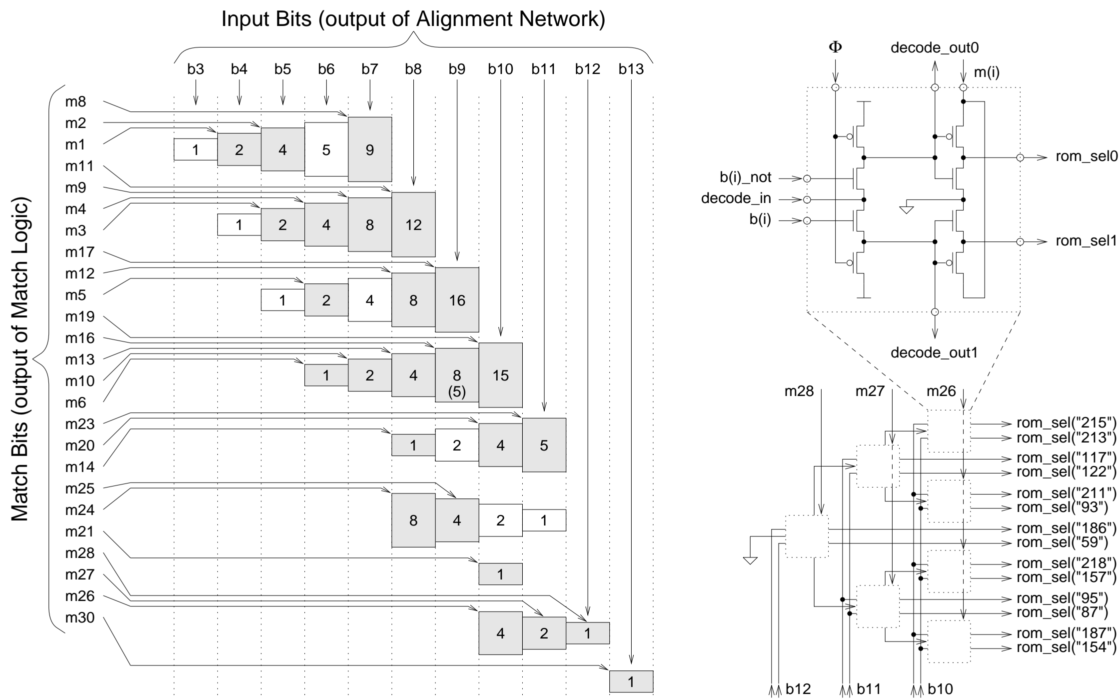

Figure 7. ROM Decoders

The decoders use dynamic logic and are activated by transitions on the b(i) inputs from the Alignment Network and the m(i) inputs from the Match Logic. As a further optimization, the decoder logic is shared between similar classes. Figure 7 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% less 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.

Since the *all-zero output symbol* is so common [Wolfe92, Kozuch94], its input code bypasses the decode ROM and directly drives the merge logic to zero for additional speed. The completion signal indicates that the output code has been determined and also clocks the result into the Output Buffer. Full/Empty bits on the Output Buffer indicate when it is time to output a full 32-bit word to the cache unit.

5.4: Shift Control and Clock Generation

While output symbols are being generated, the code bits that correspond to that symbol must also be removed from the input data stream. Input bits are removed by computing the length of the input symbol and by adjusting the input data by the appropriate number of bits. The length of each input symbol is computed using a small *Length ROM*, which

simply generates a 4-bit length value for each of the 31 match classes. It is implemented using dual-rail hazard-free dynamic logic.

As an optimization, rather than physically shifting the data by the computed number of bits, a 2-step *shifting* and *alignment* process is used. First, the current offset of the Offset Register is added to the length of the current code. The carry-out of this addition is used to indicate that the data must be physically shifted in the Input Buffer by either 0, 8, or 16 bits. The remainder is then used as the alignment offset for the next cycle.

The completion of this addition, along with completion signal from the Code ROM, together indicate the completion of the entire evaluation phase for the dynamic logic. The logic can then be precharged for the next cycle, which also clocks the new offset into the Offset Register.

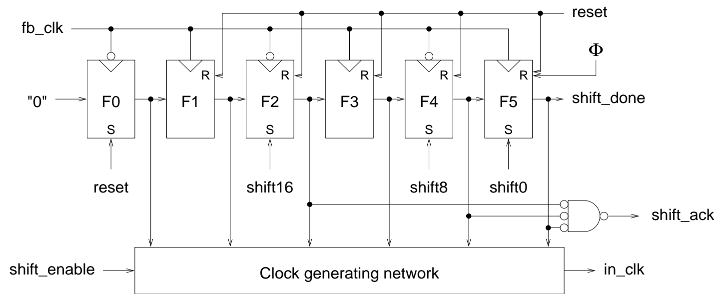

Figure 8. Shift Sequencer

If the Input Buffer needs to be shifted, the shift is also performed during the precharge phase. The *Shift Sequencer* is shown in Figure 8. The 3 inputs, shift0, shift8, and shift16, are 1-hot outputs of the Adder, indicating the desired byte-shift amount. These inputs are used to asynchronously *set* one of the flipflops in the Shift Sequencer, and thus initializes the shift operation. The timing of each shift operation is monitored by a feedback path from the Input Buffer shift clock (fb_clk).

Clocking signals are generated by a *Timing Control unit* shown in Figure 9. The unit is essentially a C-element combined with a clock distribution tree. Feedback from the slowest clocking path, the Code ROM, is used to determine the minimum precharge time.

6: Results

6.1: 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. This cycle time is defined as the time from the end of one precharge phase to the end of the following precharge phase. In practice, this is the time from the beginning of decoding a symbol from the Input Buffer, through the alignment, match, length detection and symbol lookup stages. It also includes computing the shift amount for the next symbol and any shifting of the Input Buffer that is required. It may also include the loading of an additional 4 bytes of data from the input, but we have assumed that the external program memory is not the bottleneck. We have also assumed that the Output Buffer is read before it is required for the next symbol.

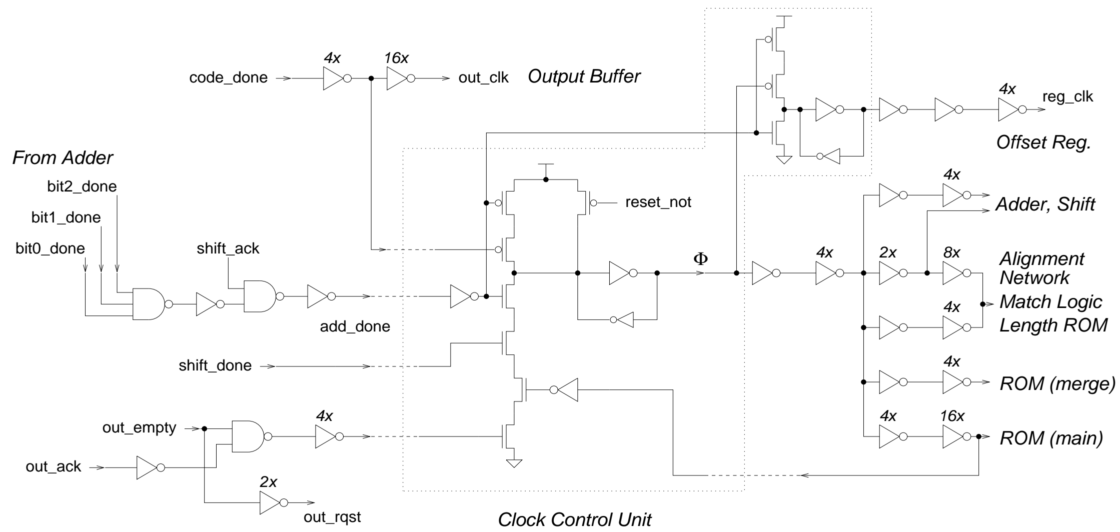

Figure 9. Timing control schematics

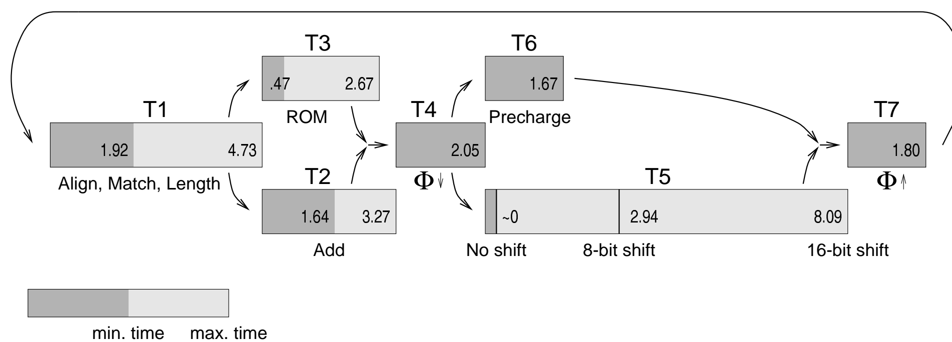

Figure 10. Breakdown of the cycle

Figure 10 shows the breakdown of the cycle time. Delay of each major logic block is only dependent on what kind of data it is processing and this delay might vary. The figure shows minimum and maximum delays for each component, indicating significant data-dependent variation.

We could characterize the behavior of the chip by simulating the cross product of each possible input symbol and each possible starting alignment of the Input Buffer, but this would give little insight into actual circuit performance for real symbol sequences. Instead, we have simulated the time required for each cycle when decoding a real encoded program.

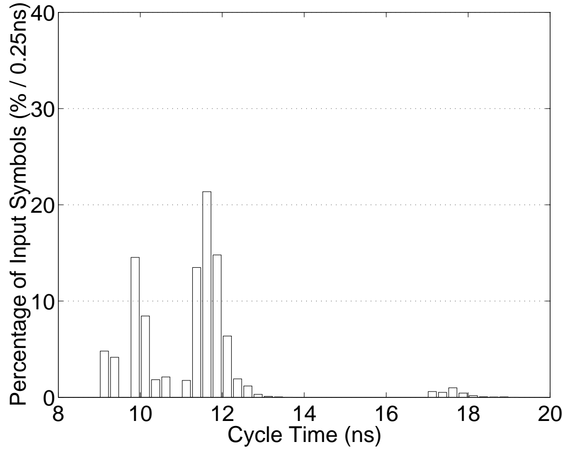

Figure 11. Distribution of cycle times for decoding individual symbols

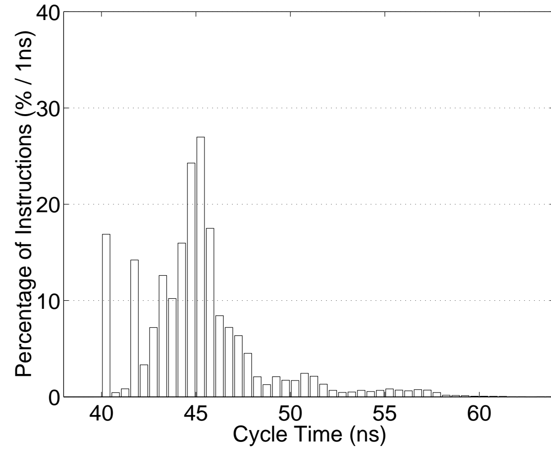

Figure 12. Distribution of cycle times for decoding 32-bit instructions

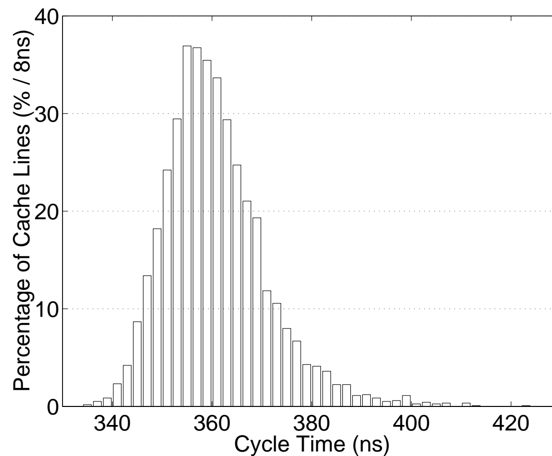

Figure 13. Distribution of cycle times for decoding 32-byte cache lines

The results from one particular 150 Kbyte sample are shown on Figure 11. Individual symbol decode cycles ranged from 9.23 ns to 19.66 ns. The distribution of decode times contains two large peaks around 10.0 ns and 11.5 ns. These represent the cycles that

required zero and one shift of the Input Buffer, respectively. The outlying cycles around 17.5 ns represent two-shift cycles. The mean cycle time of 11.23 ns is about 75% faster than the worst-case cycle time. Since the average input symbol length is 5.46 bits for the test sample, this represents performance in excess of 480 Mbit/sec.

Figure 12 shows the distribution of decoding cycle times for 32-bit instructions. Since the decoder core buffers 4 bytes of output, this represents the rate at which 32-bit instructions can be delivered to the instruction cache or directly to the processor. The CPU can asynchronously read data from the Output Buffer at this rate or it can use a synchronous clock and incorporate wait states whenever the data is not ready. Note that the variation between instruction decode times varies less than for individual symbols as would be expected for uncorrelated input data.

Figure 13 shows the distribution of decode times for full 32-byte cache lines.

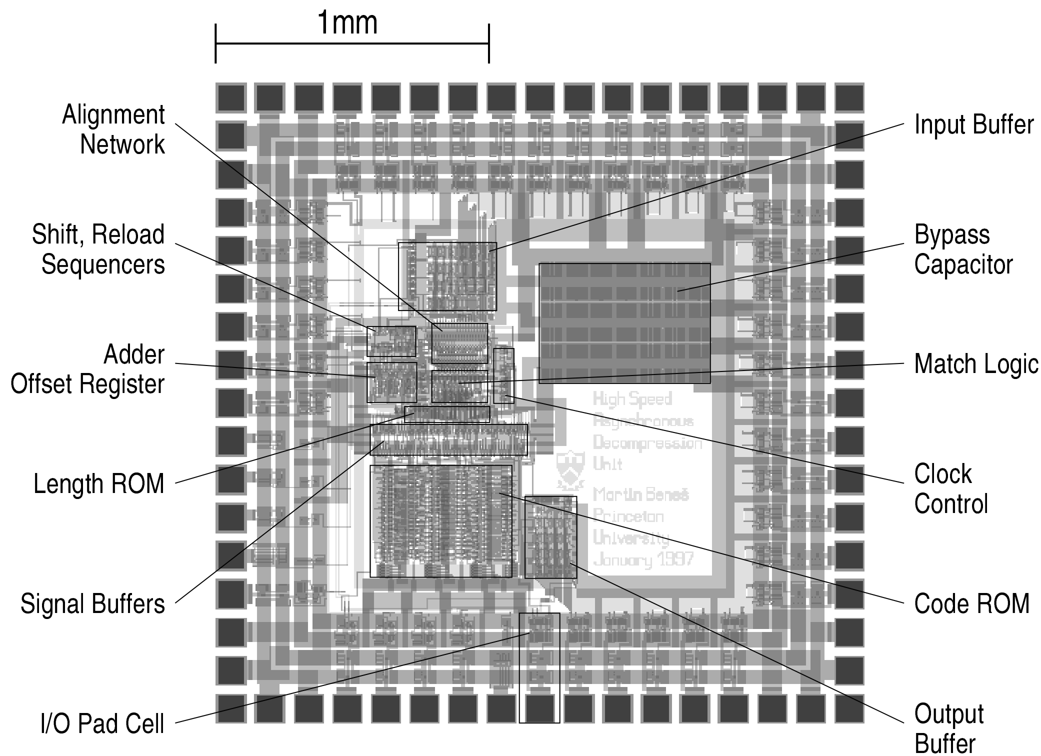

Figure 14. Floorplan and layout of the decoder test chip

6.2: Fabrication Results

The circuits described in this paper have 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 few inverters, NAND gates, and registers. The complete circuit including the input and output registers fits within a 1240 μ x 800 μ region for a total area of under 1 mm^2. Since the layout is somewhat irregular, approximately 0.25 mm^2 of this space could be used for other circuits resulting in a net area of 0.75 mm^2, which corresponds to an area of at most 3 Kbytes of ROM. The test chip uses shared I/O pins to reduce

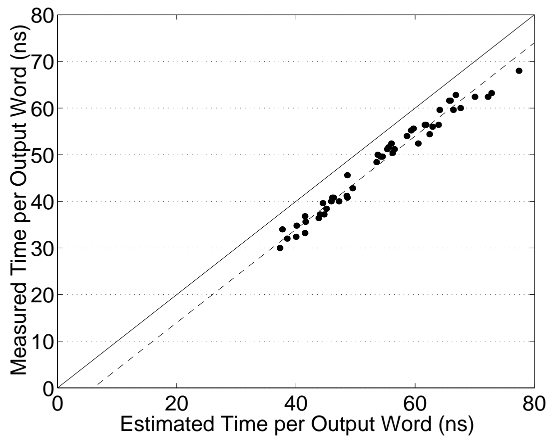

Figure 15. Timing measurements

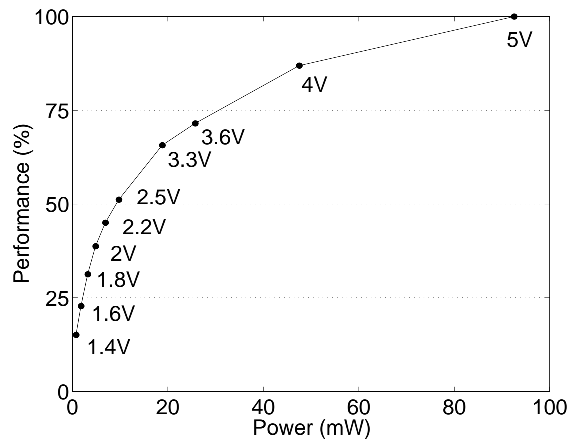

Figure 16. Power measurements

packaging costs and thus only has 60 pads. Figure 14 shows the floorplan and layout of the test chip.

Figure 15 shows the comparison of measured and estimated decode times for a sample of 50 individual instructions. The times roughly correspond to those of Figure 12, but they take into account pin delays associated with the input handshake, which can cost up to 20 ns. The distribution does not correspond to real data, rather it was chosen to cover the range of possible times. The correlation between the measured and estimated times is quite strong, which gives credibility to our simulations. On average, the fabricated chip decodes a 4-byte instruction 6 ns faster than as predicted by simulation (see Figure 15). This would put the average cycle time at 9.7 ns, *i.e.,* the time to generate one decoded output byte. The resulting average input processing rate is 560 Mbit/second.

Figure 16 shows the chip power consumption while running at maximum speed. Measurements were taken at room temperature for different supply voltages. The power is virtually independent of the processed data, since increased complexity of the computation results in slower internal cycles. The chip consumes up to 100 mW at 5 V. At 2.5 V the chip runs about half the speed, but power is cut to 10 mW. Note that, since the chip is asynchronous, power is consumed essentially only during the actual cache refill; at all other times, the system is on standby, and only leakage current occurs.

7: Advantage of the Asynchronous Solution

We now compare our asynchronous design with potential synchronous implementations. All timing data is based on simulation results.

The cycle breakdown of Figure 10 shows that the times T4 and partially T7 represent pure asynchronous overhead. This overhead thus amounts to about 3 ns each cycle, which is on average about 25% of the cycle time. Therefore it may seem that synchronous solution would eliminate this overhead and actually be faster.

However, a synchronous solution faces problems that result in worse performance. It is possible to build a synchronous decoder based on our datapath — including *Alignment Network, Match Logic, Length ROM, Code ROM, and the Adder.* Assuming each cycle produces one output byte, the evaluate phase of the system clock needs to be at least as

long as the longest possible datapath delay: it must take at least 8 ns. Since the Input Buffer needs to be shifted on the falling edge of the system clock, the Input Buffer needs to incorporate multiplexers to select the correct data to be loaded into each 8-bit register. Not only would this increase the hardware complexity, but the propagation delay through these muxes would add time to the evaluate phase of the system clock, making it at least 10 ns. Now, assuming at most 60–70% duty cycle of the clock, the cycle time would end up being 15 ns, at minimum.

Another more realistic solution would provide two overlapping clocks, one used to control the dynamic logic, and to latch the results of the Adder. Another clock would be used to update the Input Buffer during the precharge phase of the main clock. A clock period of 14 ns for this solution seems realistic.

In any case, the asynchronous design would be 20–30% faster. Moreover, we are now working on the next version of the asynchronous solution, which would include a better clocking scheme with overlapping phases. We anticipate that we may be able to eliminate some of the asynchronous overhead and shorten the average cycle time to about 8 ns. In this case, the asynchronous design would be at least 75% faster than the synchronous one.

8: Conclusions

The design presented in this paper illustrates a practical Huffman decoder based on asynchronous circuits. A variable-input-rate, variable-output-rate architecture allows the overall circuit to be optimized for the most common inputs. These circuit optimizations allow the fastest inputs to be processed more than twice as fast as the slowest inputs. Furthermore, due to the skewed distribution of symbols in Huffman-encoded data, the typical performance is 75% faster than the worst-case performance. By combining these techniques with new circuits for length detection and symbol lookup, as well as by using aggressive dynamic logic circuits, we have achieved a performance of 560 Mbit/sec. Our design is significantly smaller than comparable synchronous decoders, yet has a higher throughput rate than almost all existing designs.

References

[Choi95] S. Choi and M. Lee, "High Speed Pattern Matching for a Fast Huffman Decoder", IEEE Trans. On Consumer Electronics, v. 41, no.1, pp. 97–103, Feb. 1995.

[Davis95] A. Davis and S. M. Nowick, "Asynchronous Circuit Design: Motivation, Background and Methods", Chapter in Asynchronous Digital Circuit Design, pp. 1–49, Springer-Verlag (Workshops in Computing Series, 1995).

[Furber94] S.B. Furber, et al., "The Design and Evaluation of an Asynchronous Microprocessor", ICCD '94, pp. 217–220, Oct. 1994.

[Furber97] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day and N.C. Paver, "AMULET2e: An Asynchronous Embedded Controller", Async97, pp. 290–299, Apr. 1997.

[Hashemian94] R. Hashemian, "Design and Implementation of a Memory Efficient Huffman Decoding", IEEE Trans. On Consumer Electronics, v. 40, no. 3, pp. 345–351, Aug. 1994.

[Huffman52] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", Proc. IEEE, v. 40, no. 10, pp. 1098–1101, Sept. 1952.

[Kessels97] J. Kessels and P. Marston, "Designing Asynchronous Standby Circuits for a Low-Power Pager", Async97, pp. 268–278, Apr. 1997.

[Kozuch94] M. Kozuch, and A. Wolfe, "Compression of Embedded System Programs", ICCD '94, pp. 270–277, Oct. 1994.

[Martin89] A.J. Martin et al., "The Design of an Asynchronous Microprocessor", Caltech Conference on Very Large Scale Integration, 1989.

[Matsui94] M. Matsui, et al., "200 MHz Video Compression Macrocells Using Low-Swing Differential Logic", ISSCC 94, pp. 76–77, 1994.

[Nielsen96] L.S. Nielsen and J. Sparso, "A Low-Power Asynchronous Data Path for a FIR Filter Bank", Async96, pp. 197–207, Nov. 1996.

[Nowick96] S.M. Nowick, "Design of a Low-Latency Asynchronous Adder Using Speculative Completion", IEE Proceedings - Computers and Digital Techniques (UK), v. 143, no. 5, pp. 301–307, Sept. 1996.

[Nowick97] S.M. Nowick, K.Y. Yun, P.A. Beerel and A.E. Dooply, "Speculative Completion for the Design of High-Performance Asynchronous Dynamic Adders", Async97, pp. 210–223, Apr. 1997.

[Onoye95] T. Onoye, et al., "HDTV Level MPEG2 Video Decoder VLSI", International Conference on Microelectronics and VLSI, TENCON '95, pp. 468–471, 1995.

[Park95] H. Park, J. Son and S. Cho, "Area Efficient Fast Huffman Encoder for Multimedia Applications", 1995 ICASSP, pp. 3279–3281.

[Rudberg96] M. K. Rudberg and L. Wanhammar, "New Approaches to High Speed Huffman Decoding", 1996 ISCAS, pp. 149–152.

[Wolfe92] A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture", Micro-25, the 25th Annual International Symposium on Microarchitecture, pp. 81–91, Dec. 1992.

[Wei95] B.W.Y. Wei and T.H.Meng, "A Parallel Decoder of Programmable Huffman Codes", IEEE Trans. on Circuits and Systems for Video Technology, v. 5, no. 2, pp. 175–178, Apr. 1995.

[Yun97] K.Y. Yun, P.A. Beerel, V. Vakilotojar, A.E. Dooply and J. Arceo, "The Design and Verification of a High-Performance Low-Control-Overhead Asynchronous Differential Equation Solver", Async97, pp. 140–153, Apr. 1997.