H.264 Decoder: A Case Study in Multiple Design Points

Kermin Fleming, Chun-Chieh Lin,
Nirav Dave, Arvind
MIT - CSAIL
Cambridge, MA
{kfleming,ragnarok,ndave,arvind}@csail.mit.edu

Gopal Raghavan,
Jamey Hicks
Nokia Research Center - Cambridge
Cambridge, MA
{gopal.raghavan,jamey.hicks}@nokia.com

Abstract:

H.264, a state-of-the-art video compression standard, is used across a range of products from cell phones to HDTV. These products have vastly different performance, power and cost requirements, necessitating different hardware-software solutions for H.264 decoding. We show that a design methodology and associated tools which support synthesis from high-level descriptions and which allow modular refinement throughout the design cycle, can share the majority of design effort across multiple design points. Using Bluespec SystemVerilog, we have created a variety of designs for the H.264 decoder tuned to support decoding at resolutions ranging from QCIF video (176×144 *@15 frames/second) to 1080p video (*1280×1080*)p @60 frames/second) in a 180nm process. Some of these design points require major transformations of pipelining to increase performance or to reduce area. We also explore several common design issues surrounding memory structures, such as caches and on-chip vs. off-chip memories.*

We believe the design methodology used in this paper is directly applicable to many IP blocks involving algorithmic specifications. The same design capabilities also permit rapid microarchitecture exploration and changes in RTL late in the design process even in non-algorithmic IP blocks.

1 Introduction

The explosion of mobile device market has caused an increase in the need for fast and low-power applications like video encoding, decoding, and image manipulation. The high demand for complex mobile applications offers a number of new opportunities for ASIC development, since hardware solutions may consume one hundredth to one thousandth of the power consumed by a software solution.

These new application areas require detailed domain specific knowledge, making it difficult for hardware designers, who often lack the requisite expertise, to estimate the effect a microarchitectural modification will have on the price, power, or performance of a total design until large portions of the design have been completed. This behavioral uncertainty also applies to the performance requirements of individual blocks – implementation of other components can change the available design budgets in unexpected ways. A given feature, such as video decoding, needs to be implemented for a wide variety of products at different price, performance, and power design points, depending on the target application and market. The question is how best to produce these different designs.

One way to create hardware designs at multiple points on the power-performance scale is to build a very-parallel high-performance design and then apply frequency-voltage scaling to tune the design to the desired behavior. While this strategy undoubtedly produces very low power designs in current technologies, it ignores economic reality: reducing the silicon area of a chip directly impacts the cost of the chip. For parts shipping in very high volumes, cost per unit is a major engineering factor. Since reducing area involves reusing circuits and exploiting less parallelism, exploring the area-power trade-off is of great commercial importance.

It is desirable to adopt a methodology which allows the designer to modify designs late in the design cycle and which facilitates creating derivative designs from common pool of source code. The key capability to support such a methodology is a set of tools that automatically generate high-quality circuits from high-level descriptions and a set of design principles conducive to rapid design refinement.

To our knowledge no current industrial methodology is able to offer sufficient flexibility late in the design process. In Verilog and other RTL based development processes, designers are forced to commit to a number of decisions early in the design process, particularly in relation to inter-module communication and coordination. Any change to the datapath that affects control logic requires a significant effort and exacerbates the already onerous verification process. In most companies RTL changes late in the design cycle are prohibited by the management unless no other workaround is possible.

SystemC and other C-based languages offer flexibility in writing design descriptions, which designers exploit extensively to develop early hardware models upon which software can be developed. However, designers sometimes experience difficulty in getting SystemC or its variants to represent/generate the hardware they desire. Either the final hardware designs cannot be expressed at all or unnatural phrasings are required to express the design in manner that enables the tool chain to infer appropriate hardware. In our experience, the tools for hardware synthesis from SystemC

and other C-based languages are not competitive with traditional synthesis tools for Verilog.

Bluespec, a relatively new language for hardware design, has the properties necessary to support late design changes. Its type system supports polymorphism and higher order functions, both of which are needed for naturally expressing parameterized designs. It offers a model of concurrency based on guarded atomic actions, which ensures that all design behaviors can be understood in terms of a sequence of atomic actions on the state. Thus, if an atomic action takes longer as a consequence of the refinement, the resulting design is still guaranteed to be correct. Bluespec supports modules with guarded interfaces which provide a sound way of connecting modules without having to expose internal properties of the modules. Most importantly, Bluespec compiler offers the ability to synthesize high-quality circuits in a "push-button" manner. Several studies have shown that it is possible to create designs in Bluespec very quickly and once a design is working, to modify its parts to gain much deeper insights into the cost and effectiveness of sub-blocks [2, 4, 14].

In this paper we will explore the effectiveness of our design methodology in the development of an H.264 decoder, a modern video CODEC. We strictly enforce latency-insensitive design principles to give both the compiler and the user more leeway in implementation decisions regarding each module. In some cases, latency insensitive design may introduce area and throughput overhead above a functionally equivalent latency sensitive design. If necessary, once a good architecture is found timing-sensitive optimizations can be applied as a further refinement. In our design, we have found no need for such low-level optimizations. *We discuss the initial implementation of H.264 which took less than a man-year to develop and which showed relatively good performance (25 frames per second for 720p). We will show how modular refinement permitted us to make architectural changes to our initial design allowing us to decode 1080p resolution video streams at greater than 60 fps. We discuss the insights provided by these architectural transformations.*

Organization: We begin with an overview of H.264 decoder in Section 2 and discuss its important blocks. We then describe our initial implementation in Bluespec and show the synthesis results in Section 3. In Section 4, we discuss three significant modifications of our initial design. A new pipeline structure for the deblocking filter is described in Subsection 4.1; the effect of separating chroma and luma calculations on the pipelines is discussed in Subsection 4.2; the effect of introducing caches is discussed in Subsection 4.3; and finally our development of a low area decoder is discussed in Subsection 4.4. For each change we show the synthesis results. In Section 5 we discuss some of the implementations of H.264 that have been reported in literature. Finally, we present our conclusions in Section 6.

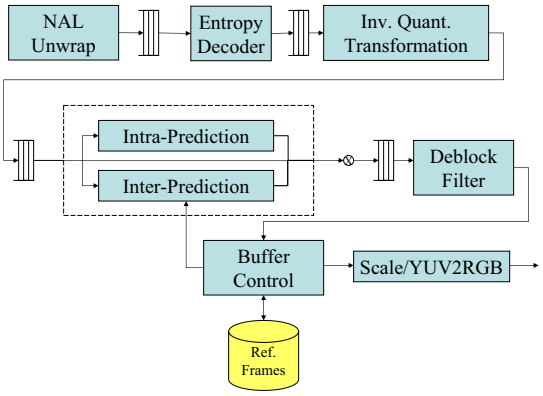


Figure 1. H.264 Decoder Block Diagram

2 The H.264 CODEC

The H.264 Advanced Video CODEC is an ITU standard for encoding and decoding video with a target coding efficiency twice that of H.263 and with comparable quality to H.262 (MPEG2) [6, 15]. H.264 enables PAL (720×576) resolution video to be transmitted at 1Mbit/sec. Like other video coding standards, H.264 specifies how to reconstruct video from a bit stream but does not specify how to encode video. H.264 shares many of the techniques used in other video CODECs and adds new variations of these techniques to improve coding efficiency.

The computational requirements of decoding H.264 video vary depending on video resolution, frame rate, and level of compression used. At the low end, mobile phone applications favor videos encoded in the QCIF format (176×144) at 15 frames per second. At the high end of the spectrum, HD-DVD videos are encoded at 1080p (1920×1080) at 60 frames per second.

H.264 reconstructs video at the granularity of 16×16 pixel macroblocks, which may be further subdivided in some decoding steps. H.264 uses two main techniques to reduce the number of bits necessary to encode video. Intraprediction predicts macroblocks in a frame from other previously-decoded spatially-local macroblocks in the same frame. Interprediction predicts macroblocks from indexed macroblocks in previously decoded frames. Within a coded frame, *slices*, or groups of macroblocks, may be intrapredicted, interpredicted from the previous frame, or interpredicted from multiple reference frames. Figure 1 shows a block diagram of our H.264 decoder.

The H.264 standard defines several *profiles*, which use different combinations of compression features. Our design is targeted for the simplest H.264 profile, the baseline profile. We implement all features of the baseline profile, with the exception of flexible macroblock ordering and arbitrary slice ordering, two seldom used data-resilience features.

NAL Unwrap: The Network Adaptation Layer (NAL) interprets sequences of bits and finds and marks the stream

with the coarse grain packeting information. The NAL also extracts high-level control information and passes it downstream to subsequent blocks.

Entropy Decoder: The H.264 CODEC uses variable-length entropy coding to encode integers. H.264 uses two techniques for this: CAVLC(Context Adaptive Variable Length Coding) and CABAC(Context Adaptive Binary Arithmetic Coding). Both techniques feature context-aware bit-mappings that vary during decoding. CABAC produces better compression but its complicated probability models makes it more computationally intensive.

Inverse Transformation and Quantization: H.264, like many video CODECs, represents data via a fixed prediction, based on previously decoded image data, coupled with a residual error value representing the difference between the fixed prediction and the original image. This greatly enhances compression, since the prediction modes can be concisely expressed. In H.264 error-correction residual can be either 4×4 or 8×8 pixels (previous standards used only 8×8 blocks). Since residual data exhibits high spatial entropy, H.264 employs a lossy, low-pass discrete cosine transform to develop a compact representation of the residual values. H.264 also allows variable quantization of DCT coefficients to enhance coding density.

Intraprediction: Video frames have a high amount of spatial similarity. Intraprediction use previously decoded, spatially-local macroblocks to predict the next macroblock. Intraprediction works well for low-detail images.

Interprediction: In video, frames nearby in time have only small differences. Interprediction attempts to capitalize on this similarity by encoding macroblocks in the current frame using a reference to a macroblock in a previous frame and a vector representing the movement that macroblock took to a $\frac{1}{4}$ pixel granularity. The decode uses an interpolation process known as *motion compensation* to generate the prediction value. Fractional motion vectors are interpolated from multiple previous macroblocks.

Deblocking Filter: Since lossy compression used to encode pixel blocks in H.264, decoding errors appear most visibly at the block boundaries. To remove these visual artifacts, the H.264 CODEC incorporates a smoothing filter into its encoding loop. However, not all inter-block discontinuities are undesirable; edges in the original image may naturally occur on block boundaries. H.264 incorporates fine-grained filter control to preserve these edges.

Buffer Control: H.264 does not require interpredicted images to depend on temporally-local, temporally-ordered images. Rather, frames can be predicted from previously decoded frames corresponding to frames far in the past or future of the video. Buffer control maintains a set of previously decoded frames and is responsible for handling the in-stream requests to access (*e.g.,*delete, prediction logic reads, writes from deblocking) these frames in its store.

We note in passing that H.264 decoding entails a large amount of computation (as many as 30 8-bit or 16-bit fixed-point multiplies per pixel). Most of these computations take place in four blocks – Inverse Quantization, Inter- and Intra-prediction and the Deblocking filter.

```
module mkH264( IH264 );
  // Instantiate the modules
  NalUnwrap     nalunwrap  <- mkNalUnwrap();
  EntropyDec    entropydec <- mkEntropyDec();
  InverseTrans  invtrans   <- mkInverseTrans();
  Prediction    pred       <- mkPrediction();
  DeblockFilter deblock    <- mkDeblockFilter();
  BufferControl bufctrl    <- mkBufferControl();
  // Internal connections
  mkConnection(nalunwrap.ioout, entropydec.ioin);
  mkConnection(entropydec.ioout_InvTrans,
               invtrans.ioin);
  mkConnection(entropydec.ioout, pred.ioin);
  mkConnection(invtrans.ioout,pred.ioin_InvTrans);
  mkConnection(pred.ioout, deblock.ioin);
  mkConnection(pred.mem_client_buffer,
               bufctrl.inter_server);
  mkConnection(deblock.ioout, bufctrl.ioin);
  // Interface to input generator
  interface ioin = nalunwrap.ioin;
  // Interface for output
  interface ioout = buffercontrol.ioout;
endmodule
```

Figure 2. H.264 Top Level in Bluespec

3 Initial Design

References for advanced video CODECs, such as H.264, are available in two forms: a textual standard definition document that spells out basic algorithms, data formats, rate requirements, etc. and a software reference implementation that captures the functionality of the CODEC [6, 7]. Reference implementations rarely meet performance requirements and tend to be poorly organized due to the large number of contributing developers. In the case of H.264, the reference implementation is an enormous eighty-thousand line C code[7]. Worse than this verbosity, reference codes typically do a poor job of elucidating the dataflow relationships between the various components of the algorithm. Nevertheless, reference implementations and other high-level implementations (such as ffmpeg [5]) play an important role both in the verification of other implementations and in debugging of the specs themselves.

The initial design of H.264 in Bluespec SystemVerilog (BSV) was done by Chun-Chieh Lin [9] in approximately one man-year. This was a remarkable achievement given that Lin started with an English description of the CODEC [6] and 20k lines of C code extracted from FFMPEG [5]. This C code proved to be worthless for hardware development but with some modifications it played a crucial role in verification.

3.1 Coding in Bluespec SystemVerilog

Lin's implementation closely models the block diagram for the CODEC shown in Figure 1. To keep the design as flexible as possible each block was organized to support latency-insensitive communications. Figure 2 shows the Bluespec implementation of the top-level module of the

Frontend Version	Speedup	Area (mm^2)	Post-route Clock Period
Original Frontend	1.00	0.34	6.47 ns
Zero Encoding	2.63	0.33	6.40 ns
2-Stage Exp-Golomb	2.82	0.28	5.96 ns

(a) Frontend Refinement Results (NAL and Entropy decoder)

Version	Speedup	Area (mm^2)	Post-route Clock Period
Original Full	1.00	5.44	11.82 ns
Wider FIFOs	1.69	5.32	12.31 ns
FIFO Sizing	1.88	5.45	11.86 ns

(b) Initial Full Design Refinement Results

Module	Lines of Code	Gates	PAR Area (mm^2)	Normalized Area	Critical Path
mkBufferControl	970	36365	.34	0.05	N/A
mkDeblockFilter	786	291447	2.74	0.40	N/A
mkEntropyDec	1656	34323	.32	0.05	N/A
mkInverseTrans	702	41865	.39	0.06	N/A
mkPrediction	2189	316169	2.97	0.43	N/A
Total	9375	730822	6.88	1.00	13.41 ns

(c) Lin's Final Design Code Size, Gate Count, and Post Place-and-Route Area

Figure 3. Lin's Initial Design[9]

H.264 decoder. Each block corresponds directly to a single Bluespec *module*. These modules have well-defined interfaces consisting of *methods* through which all inter-module communication takes place. For each communication channel in the CODEC design two methods were written: one in the sender which handles the work the sender needed to do to send data, and one which did the corresponding work for the receiver. By connecting these two together using the `mkConnection` statement we can succinctly represent these connections.

Memory Interfaces: Many of the blocks in the design need some amount of local memory not shared by other blocks. It is not clear *a priori* if these memories should be kept local or combined into a single off-chip memory. To support both of these possible organizations, memories were represented as out-of-block modules with delay-insensitive request/response interface. This made it possible to later merge some of the memories with the local module and reorganize the rest as a shared memory whose arbitration was outside of the main pipeline.

Verification Methodology: To verify the design, unit testing of individual modules was performed using the modified C reference codes. To facilitate unit testing, we made use of the special connections (*e.g.,*`mkSniffedConnection`) at different locations of the design which allow packet sniffing. Data collected was compared against similarly instrumented versions of references codes.

Synthesis Methodology: Synthesis results were obtained using Synopsys Design Compiler targeting the Tower 180nm ASIC library. Place-and-route was done using the Cadence Encounter. Designs were verified using post-synthesis simulation. The reported area does not include on-chip or off-chip memory resources. We present post-synthesis gate-level simulation power numbers collected from Sequence PowerTheater.

3.2 Lin's Optimizations:

Once the front end of the design, from NAL unwrap to entropy decoder, was completed, Lin improved some aspects of the design. These results are shown in Figure 3(a) and described in the following paragraphs. *Speedup was calculated by dividing the time needed to decode the test videos in the improved design by the time needed for the first working version of the design.*

Performance Effects of Data Representation: Discrete Cosine Transform tends to produce non-zero coefficients for low frequency components only. As a result, CAVLC decoding typically generates long sequences of consecutive zero sample values when decoding DCT coefficients. Initially each of these zeros was represented as a single "element" in the output stream, meaning every zero had a fixed cycle-cost to handle. By switching to a scheme with simple run-length coding for consecutive zeros between the CAVLC and the inverse transformation block, we were able to dramatically improve the throughput of the earlier part of the decoding pipeline. Not only does this improve the speed, but simplifies the hardware needed to decode the input stream as the stream naturally produces multiple zeros concurrently. In our experiments, the frontend (the NAL unwrapping unit and entropy decoder) showed more than 2.5 times speedup in video decoding times [9].

Exp-Golomb Decoding Refinement: Exp-Golomb Codes are variable length integer codes used in entropy decoding. In H.264, Exp-Golomb code words can range in size from 1 to 33 bits. To implement this, we initially constructed a large single-cycle function which could handle up to 33 input bits. This implementation had a high area cost. Most Exp-Golomb codewords are much smaller than the worst case size. To reduce the overall cost, we replaced the single-cycle function with two parallel versions: a smaller single-cycle block capable of only handling the smaller code words (up to 16-bits), and a multi-cycle block which took two cycles to find the larger code words. By using these blocks

in tandem, we were able to substantially reduce circuit area and improve the critical path with almost no degradation in cycle-level performance. This tranformation also allow us to reduce the required buffing of the input bitstream from 65 bits down to 33 bits.

Once the full H.264 specification was implemented, Lin explored a number of variants. These results are shown in 3(b) and described in the following paragraphs.

Widening FIFOs: Originally only one pixel-wide sample element could be stored in the stream FIFOs between blocks. However, many processes in H.264 operate on 4-pixel wide elements at a time by definition or have no data dependencies between consecutive pixels and so may operate on larger data aggregations in parallel. Augmenting the pipeline to handle four pixel wide data elements nearly doubled the throughput of the entire system while adding comparatively little area.

FIFO Buffer Sizing for Runtime Performance: Rate matching the throughput of modules in H.264 is quite difficult *a priori*, since each module is doing completely different work with it's own unique input-output timings. Adding more inter-module buffering helps alleviate the jitter across modules by letting the producer build up "work credits" with respect to the consuming module. In such latency insensitive designs, one can increase the size of FIFO buffers connecting successive modules without affecting the functional correctness.

Except for the buffer feeding into the deblocking filter increasing the sizes of buffers had negligible effects on the performance. The initial deblocking filter design exhibited bursty consumption of its input, meaning that for small sizes the queue would possibly empty before new data could be reentered. Increasing the size of the FIFO smoothed out this burstiness. As we shall see later, after pipelining the deblocking filter, this interaction stopped being a bottleneck and a smaller FIFO size could be used.

4 Design Changes

In this section we review a few major architectural changes made after the completion of Lin's final design. These changes were done in only a few man-months of effort by Fleming, with neither guidance from the original designer nor any domain-specific knowledge.

4.1 Reorganizing the Deblocking Filter

As seen in Figure 3(c), the deblocking filter takes 40% of the total area of the original design. Thus, it was the natural starting place for Fleming's refinements.

The H.264 standard is specified by a set of sequential processing steps. While this sequential style makes the specification easier to understand, following the specification directly ignores many opportunities parallelism in a hardware implementation. Some examples that we have already discussed, such as widening the pixel processing data path to exploit data parallelism are well known in the H.264 literature [11]. We now discuss an example of extracting non-trivial pipeline parallelism from the H.264 standard by analyzing the data dependecies implied by the standard's sequential specification.

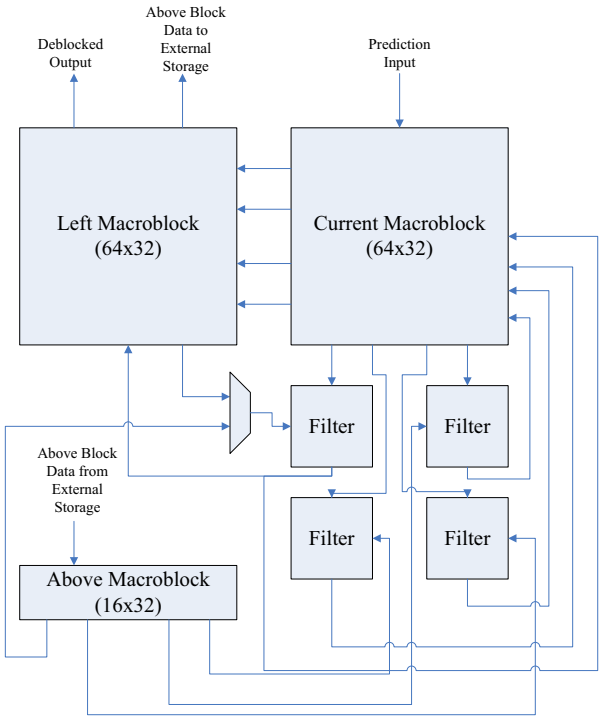
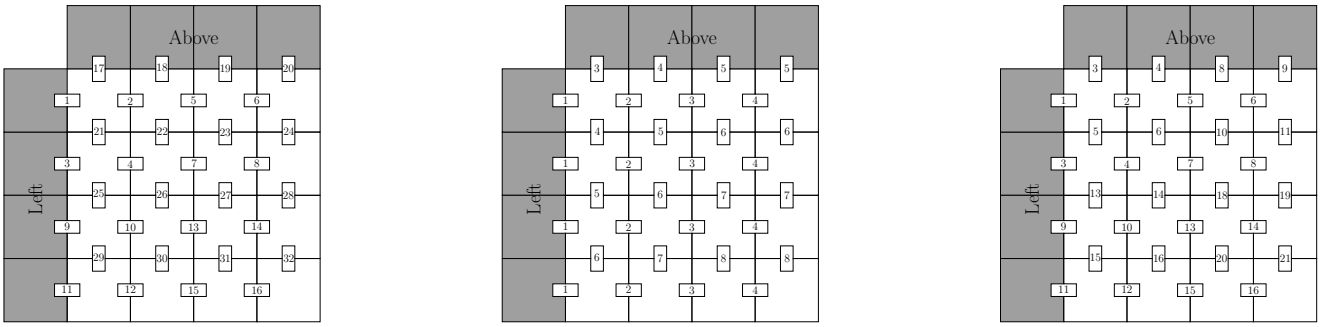


Figure 5. Original Deblocking Filter Design

At a high-level, the deblocking filter applies an 8-pixel filter across each row and column of each 16×16 pixel macroblock, with a four-pixel shift between filter applications. Thus, each pixel is filtered four times, twice horizontally and twice vertically. The H.264 standard specifies the deblocking filter application ordering depicted in Figure 4(a). The original implementation of the deblocking filter, depicted in Figure 5, followed this ordering exactly.

Even if a complete edge (four filtration applications) is filtered every cycle, 32 cycles are required to process each macro block. Worse, the filter application ordering exhibits poor locality across the macroblock. The top left 4×4 pixel block is processed first, but 15 filtration steps are applied to other non-local blocks before the top left block is vertically filtered. This poor temporal locality means that an entire macroblock must be stored within the deblocking filter, making macroblock streaming difficult. To improve throughput, we inserted a macroblock pipeline stage in the form of a deep FIFO buffer immediately before our original deblocking filter. The storage requirements of this deblocking filter implementation, which follows the H.264 specification closely, were enormous, as shown in Figure 3(c).

To refine the deblocking filter, we examined the dataflow graph implied by the specification of the deblocking filter, shown in Figure 4(b), and then construct an implementa-



(a) H.264 Deblocking Filter Macroblock Edge Filtration Ordering

(b) Filter Edge Dependencies

(c) Pipelined Deblocking Filter Processing Order

Figure 4. Deblocking Edge Orderings

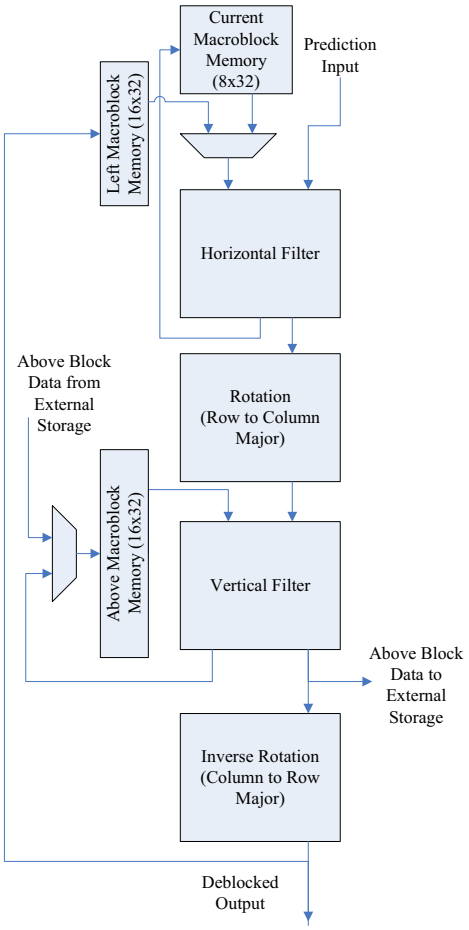


Figure 6. Optimized Deblocking Filter Design

tion that respected these data dependencies. This order also shows that we can achieve high temporal locality if the horizontal and the vertical filtrations of a 4×4 block are done in close succession. To this end, we constructed a streaming pipelined implementation of the deblocking filter, sketched in Figure 6. The new implementation processes horizontal and vertical edges in parallel, using the filter ordering shown in Figure 4(c). Thus the new design will complete the horizontal filtration of 4×4 block, and then immediately commence on the vertical filtration of the block in a streaming fashion. Due to the streaming, the new design stores a little more than half of a macroblock, inclusive of pipeline registers. This design also completely overlaps horizontal and vertical filtration, thereby improving throughput. *As a result of this microarchitectural change, the deblocking filter implementation area decreased dramatically from 2.74 mm^2 to 0.69 mm^2. Using the new deblocking filter, we get a 12% increase in throughput of the entire design, while reducing the design critical path by 35%.* Full synthesis and performance results can be seen in Figure 7.

Methodology Advantages: By treating the overall decoder design as a latency insensitive pipeline, it is easy to completely overhaul the memory access patterns and computation order without affecting correctness. To our knowledge our implementation is the only fully pipelined deblocking filter done at the sub-macroblock granularity. Chen et al., [3] state that the 4×4 block pipelining of the deblocking filter is impractical due to data dependencies within the pipeline and large control overhead. The choice of Bluespec as our language removed the tedium of manually modifying the complex module control logic, and allowed us to focus solely on correcting structural hazards in the pipeline. Because the Bluespec compiler detects and reports structural hazards, we were able to quickly spot and modify problem areas of the pipeline, which greatly reduced debugging time. Our latency-insensitive design meant that timing changes within the deblocking filter did not impact the behavior of other modules in the design. While debugging the new design, we found no integration bugs external to the deblocking filter.

4.2 Separating Luma and Chroma Pipelines

H.264 compresses video streams in the YUV color space, as opposed to the RGB color space commonly used in graphics processing. Three image planes are used to describe a YUV image; Y, the luminance (luma) component represents light intensity, while U and V, the chrominance (chroma) components, represent blue and red shades. Since

the human eye is less sensitive to variations in color than to changes in light intensity, the chroma components may be represented with less resolution without reducing perceived image quality. The H.264 CODEC encodes chroma at one quarter the resolution of luma. It also uses different, but similar, prediction processes for luma and chroma.

In the video stream, luma and chroma data alternate. For the most part, the processing dataflow used to decode the two fields is similar. The original implementation exploited this similarity by using the same pipeline for both luma and chroma. The original implementation Each block switches "modes" at appropriate times to deal with each kind of data. Outside of a few high-level in-stream control values, there are no dependencies between the processing of luma and chroma. By separating the luma and chroma pipelines, we allow the field computations to be overlapped.

In general, block duplication should increase implementation area. However, luma and chroma specific logics are not duplicated and buffering in many modules is reduced due to improved design parallelism. Due to the complexity of these tradeoffs, the affect of duplicating the pipeline on area is difficult to determine without synthesis.

One need not separate luma and chroma processing in every module. For example, the unified inverse transform module has sufficient bandwidth to satistify the requirements for downstream modules. Therefore, we only consider duplicating the interprediction and the deblocking filter modules, since these modules are design bottlenecks.

Module specialization turned out to be quite straightforward. Since Bluespec represents behaviors permissively, by removing all rules dealing with luma (or chroma) samples, we immediately obtain the desired behavior. Unused state and combinational logic associated with the excised rules is removed automatically by backend tools.

To replace a module with a split version, we need to split the input stream, pass these streams into the appropriate specialized module and then merge the output stream together. In this form deblocking filter now looks like:

```
module mkDeblock_split(DeblockIFC);
  split     <- mkLumaChromaSplitter;
  deb_luma   <- mkDeblockLuma;
  deb_chroma <- mkDeblockChroma;
  merge      <- mkLumaChromaMerger;
  //Chroma Path
  mkConnection(split.chromaout,deb_chroma.in);
  mkConnection(deb_chroma.out, merge.chromain);
  //Luma Path
  mkConnection(split.lumaout, deb_luma.in);
  mkConnection(deb_luma.out, merge.lumain);
  //Interface
  input  = split.in;
  output = merge.out;
endmodule
```

While this split-join utterance allows us to keep the module interfaces the same, when splitting multiple consecutive blocks it is inefficient. We merge and immediately split two unrelated streams. To keep the streams separate we must modify the module interfaces. This modification lead to a few percent performance gain. *In total, splitting luma and chroma computations achieved a* 20% *performance improvement with only a* 10% *increase in area.*

4.3 Shared Memory and Caching

Interprediction, intraprediction, and deblocking all access previously decoded data. Since several previous frames must be stored at the same time, the memory required to store all of the necessary data can be quite large, particular in larger frame sizes (*e.g.,*1080p). A number of cost/performance/power tradeoffs are possible in the memory subsystem. The highest level of performance can be obtained by giving each module that requires memory its own private, fast, pipelined SRAM. This configuration is area-intensive due to the low bit-density of SRAMs. On the other end of the cost/performance spectrum, a single, low-bandwidth DRAM could be shared between all the modules which access memory. DRAM is unattractive for low-power applications due to its high power consumption per access. Regardless of the backing memory configuration, a cache may improve several figures of merit. Small caches consume far less power than either a large SRAM or a DRAM, perhaps as much as two orders of magnitude less energy per access. Additionally, a small cache close to the block will have lower latency than a larger backing memory. Since our design is latency insensitive, it is particularly suited for memory subsystem design explorations.

Intraprediction and Deblocking have well-defined, highly sequential access patterns. In particular, both use previously decoded pixels directly above and to the left of the current block. Each of the previous pixel data is read exactly once, used for computation, and then never used again. Since the data to the left of the current macroblock has high temporal locality and is relatively small, it can be cached in registers in both deblocking and intraprediction. The previously decoded data above the current macroblock, on the other hand, has no temporal locality. To effectively cache this data, the entire previously decoded row must be stored in a fast memory. Depending on the image size that the CODEC is expected to handle, the cache must be scaled up or down accordingly. Since the access pattern to these memories is predictable and sequential, prefetching techniques could also be applied to improve memory throughput. Along this line, our modules optimistically issue memory accesses early and buffer memory replies to help overlap the memory latency with useful computation.

In the interprediction module, pixels may be interpolated by applying a filter to a set of previously decoded pixels from an earlier frame. In this process a single pixel may be reused several times to predict a macroblock in the case of fractional motion vector compensation. Additionally, if a pixel is used, there is high probability that nearby pixels will also be used. The level of temporal These are exactly the access patterns where caches are targeted.

Since luma and chroma interpolation have different prediction computations and memory access patterns, the

Version	Gates	Area	Clock Period	Max. Power
Original	730822	6.88 mm^2	13.41ns	338 mW
Pipelined Deblocking	417256	3.88 mm^2	9.91ns	228 mW
Design targeting QCIF@15 fps	234530	2.22 mm^2	31.22ns	148 mW
Design targeting 720p@30 fps	262169	2.44 mm^2	20.00ns	180 mW
Split Luma/Chroma Deblocking	432971	4.07 mm^2	10.31ns	226 mW
Split Luma/Chroma Interpolation	408865	3.85 mm^2	10.34ns	279 mW
Full Luma/Chroma Split	447721	4.21 mm^2	9.99ns	305 mW

(a) Results of Architectural Exploration

Module	Lines of Code	Gates	Area mm^2 Synth	Area mm^2 PAR	% of Area	Critical Path
mkBufferControl	970	32313	.024	.30	0.08	
mkDeblockFilter	1110	73725	.054	.69	0.18	
mkEntropyDec	1657	27649	.019	.27	0.07	
mkInverseTrans	702	34640	.024	.33	0.08	
mkPrediction	2189	237901	.169	2.22	0.57	
Total	8137	417256	.291	3.88	1.00	9.91 ns

(b) Final Design Code Size, Gate Count, and Post Place-and-Route Area

Clip Resolution	Cycles/Frame			
	Original	Pipelined Deblocking	Low Area(720p)	Full Luma/Chroma Split
1080p	1971000	1900000	N/A	1487000
720p	1070000	940000	1499000	833000
VGA	278000	250000	471000	240000
QCIF	23700	22000	40000	19000

(c) Simulation Throughput

Figure 7. Design Exploration Results

caches should be specialized separately. Figure 9 presents the results of basic cache experimentation. A direct-mapped, blocking cache was inserted between the frame buffer controller and the frame buffer store. Luma and chroma requests were handled by separate caches. The memory access patterns tested were taken from a low-bitrate QCIF video stream and a high-definition 720p clip. *It is evident from Figure 9 that even a small cache is highly effective in capturing interprediction memory locality*. With a four-byte line size and two one-kilobyte cache, 46% of luma memory and 30% of chroma memory requests hit in the cache. Using larger cacheline sizes more than double the hit rates. The largest gain in caching performance occurs when making the cache large enough to contain the image data of three adjacent macroblocks at the same time, that is, large enough to capture the data of the macroblocks to the immediate left and right of the current macroblock, since these data values are likely to have been recently used (left) or to be used again soon (right). Further hit-rate improvements require caching the rows of macroblocks above and below the current macroblock; some of this benifit is seen by the larger caches in the QCIF experiment.

4.4 Low Area Design

Up to this point, we have discussed design explorations which focus on mainly improving throughput, typically by expanding computational circuitry. These improvements come at the cost of increasing the chip area. *We now discuss the relative ease with which we were able to decrease the implementation area of our decoder by nearly* 50%. The low area implementations specifically target two performance points, QCIF at 15 fps and 720p at 30 fps. These points were chosen because of their use in common price-sensitive commercial applications (*e.g.,*cellular phones and HD television sets).

The main area savings were obtained by folding computation logic. In the high-performance version, we improved the throughput by applying several data-independent filters in parallel. Each filter requires multiple additions and multiplications which require significant implementation area. In the high-performance implementation, we allow the system to instantiate as many filters as necessary to exploit the parallelism in our design. To reduce area, we constrain the compiler to implement exactly one filter of each kind. The compiler automatically generates the necessary multiplexing logic. Such a transformation in Bluespec requires only a few lines of code; in lower level RTL such a change might require significant rewriting and debugging.

In our original design exploration, many FIFO buffers were sized above their minimum (*i.e.,*one element) to help smooth dynamic rate mismatches between modules. We can easily descrease the buffer sizes back to the minimum. Since our design is latency insensitive, this alteration has

Cache Size (8-bit Pixels)	Chroma Hit Rate Line Size (Pixels)				Luma Hit Rate Line Size (Pixels)			
	4	8	16	32	4	8	16	32
128	.177	.346	.431	.363	.032	.394	.511	.573
256	.336	.530	.636	.453	.033	.398	.516	.576
512	.369	.554	.657	.466	.242	.609	.640	.629
1024	.463	.729	.814	.823	.305	.650	.719	.720
2048	.464	.729	.862	.925	.306	.650	.720	.721
4196	.564	.780	.888	.944	.558	.778	.888	.944

Figure 8. H.264 Interprediction Cache Parameter Exploration (QCIF)

Cache Size (8-bit Pixels)	Chroma Hit Rate Line Size (Pixels)				Luma Hit Rate Line Size (Pixels)			
	4	8	16	32	4	8	16	32
128	.082	.373	.509	.578	.013	.247	.330	.371
256	.138	.429	.565	.635	.067	.275	.348	.384
512	.376	.666	.810	.883	.217	.432	.510	.552
1024	.393	.684	.829	.903	.468	.719	.837	.898
2048	.428	.712	.854	.926	.516	.756	.876	.937
4196	.428	.712	.854	.926	.516	.756	.876	.937

Figure 9. H.264 Interprediction Cache Parameter Exploration (720p)

no impact on correctness of the decoder, though it markedly decreases throughput.

In addition, some minor area savings were achieved in the following ways. We reduced the state elements used for frame related bookkeeping. The widths of these circuits are directly related to the largest frame that can be processed by the decoder. Since the decoder is parameterized by the maximum global frame size, adjusting this variable reduces all relevant circuits in the decoder. We also gained some improvement by synthesizing our decoder with a less aggressive clock frequency.

5 Related Work

The literature contains many examples of H.264 hardware designs, most of which target low-power operation. As such, many of them apply aggressive voltage scaling to obtain extremely low-power consumption. We expect that if we applied the power scaling techniques suggested in these papers we would obtain power improvements.

While a software SystemC reference of H.264 has been written [1], only small portions of the decoder have been synthesized into hardware [17].

Chen et al., [3] conclude that only dedicated hardware accelerators can provide the real-time H.264 decoding at 30 fps, which requires about 3600 GIPS computation and 5570 GBPS memory bandwidth. To meet these throughput requirements, they propose a new hardware architecture with four stage macro block pipelining, hybrid task pipelining scheme and low bandwidth motion-compensation scheme. They implement an H.264 baseline profile decoder that consumes about 186mW for 1080p at 30fps.

Lin et al., [10] describe a low power implementation of H.264. By using a memory-efficient decoder ordering and hierarchical syntax parsing they claim to save 28% and 17% of memory access in the interprediction and intraprediction units respectively. They are able to save 86% of power by using gated clocking techniques. According to their simulation results, 720p at 30fps consumes 45 mW.

Liu et al., [11] implement a H.264 decoder for mobile applications. To conserve power, they employ a 4×4 sub-block level pipelining scheme, clock-gating, voltage scaling, and a three level memory hierarchy. They propose content-switched, hybrid-scheduled, and code-word partitioning methods to achieve high throughput. They report 865 uW for decoding QCIF at 30fps. Our design may be implemented with the maximally parallel 4×4 scheme they suggest for inter and intra prediction, but our design may also be parameterized for less parallelism.

Liu et al., [12] improve their pipelining structure by using domain-pipelined scalability techniques. Their line-pixel look-ahead scheme provides 51% memory power reduction. Operating frequency is reduced by using low-power motion compensation and deblocking filter. They fabricated a test chip that is 15.21 mm2 and consumes only 125 μW decoding QCIF at 15 fps.

Kang et al., [8] implement a decoder in three stages. Initially, they developed a decoder in C and verified it in a virtual environment. After verification, they ran a video display test on an FPGA. They also synthesized all the blocks in a 130 nm ASIC process. Their design, when processing 1080p, consumes about 554 mW at 130 MHz. Some of the less parallel blocks (*e.g.,*parsing), are implemented in software on an ARM processor.

Shih et al., [16] propose a pipelined implementation of the deblocking filter. Within a small margin, their proposed design achieves one filtration step per cycle using a five-stage processing pipeline. Our implementation can be parameterized to obtain a similar pipeline, but our design may be parameterized for greater parallelism.

6 Conclusion

Whether one wants to explore several design alternatives to understand area-power tradeoffs or to incorporate a complex IP block in several products with different performance, power or price requirements, one needs an RTL implementation representing each design point. There are a mature set of commercial tools to synthesize hardware from structural RTL, but producing high-quality RTL in the first place remains an expensive proposition. Even retargeting working RTL to a different design point remains a daunting challenge. This paper shows that it is possible to generate multiple RTL designs for a complex IP block (H.264), if one starts with a high-level parameterized description amenable to modular refinement. In particular, *we have demonstrated an inexpensive, low-performance decoder targeting mobile platforms which can be implemented in 2.22 mm^2 and a number of high-performance decoders capable of decoding 1080p video at more than 60 fps that takes only twice that area.* The source code for our decoders is available freely under open source licensing [13].

It has been shown before that some algorithmic applications like OFDM-based wireless protocols are amenable to parametric design exploration [14]. For example, the area in the design of an 802.11a transmitter is dominated by the inverse Fourier transform circuit, which can be parametrically folded. H.264, however, offers a greater challenge because the control flow and processing are much more data-dependent than OFDM protocols, and require complex architectural refinements. We have discussed four major local refinements in this paper, which change parallelism to achieve higher performance, lower power, or reduced area. Though experienced designers may anticipate some of the issues in our initial designs, we seriously doubt if one could have reached our final designs without some experimentation.

The initial implementation of H.264 was done in Bluespec SystemVerilog in less than one man-year, while the refinements were done by another designer within 2 to 3 man-months. Writing RTL for these different variants would have been a tall order, and unlikely to be carried out in industry unless the first design did not meet the requirements. Designers would have used all kinds of ad hoc physical design tricks, voltage scaling and multi-clock domains before changing the RTL because of the fear of exacerbating the verification problem. Also this exploration would have been practically impossible in C-based languages because of the lack of control to express these refinements and the lack of tools that provide high-quality synthesis.

The ability to do modular refinement is essential to avoid a verification nightmare, and the ability to synthesize designs to gate level is essential to study area and power numbers. Thus, the future design methodologies and associated tools must provide both modular refinement and high-level synthesis to support the creation of reusable IP blocks.

Acknowledgments: This work was supported by Nokia sponsored research at CSAIL, MIT. Arvind was partially supported by NSF grant CCF-0541164. We have benefitted from many discussions with Daniel Finchelstein, Vivienne Sze, and Professor Anantha Chandrasakan regarding transformations for low power, particularly field parallelization. We are also thank Jae Lee for his assitance in extracting an H.264 CODEC from ffmpeg.

References

[1] I. Amer, M. Sayed, W. Badawy, and G. Jullien. On the way to an h.264 hw/sw reference model: a systemc modeling strategy to integrate selected ip-blocks with the h.264 software reference model. *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on*, pages 178–181, 2-4 Nov. 2005.

[2] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD'04*, San Jose, CA, 2004.

[3] Tung-Chien Chen, Chung-Jr Lian, and Liang-Gee Chen. Hardware Architecture Design of an H.264/AVC Video Codec. In *Proceedings of ACM Asia South pacific Design Automation Conference*, 2006.

[4] Nirav Dave, Michael Pellauer, Steve Gerding, and Arvind. 802.11a Transmitter: A Case Study in Microarchitectural Exploration. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Napa, CA, 2006.

[5] FFMPEG Multimedia System. `http://www.ffmpeg.org`.

[6] ITU-T Video Coding Experts Group. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, May, 2003.

[7] H.264/AVC Reference Software. `http://iphome.hhi.de/suehring/tml/`.

[8] Hae-Yong Kang, Kyung-Ah Jeong, Jung-Yang Bae, Yong-Su Leee, and Seung-Ho Lee. MPEG AVC/H.264 Decoder With Scalable Bus Architecture and Dual Memory Controller. In *ISCAS 2004: Proceedings of IEEE International Symposium on Circuits and Systems*, 2004.

[9] Chun-Chieh Lin. Implementation of H.264 Decoder in Bluespec System Verilog. Master's thesis, MIT, Cambridge, MA, Feb 2007.

[10] Ting-An Lin, Tsu-Ming Liu, and Chen-Yi Lee. A Low-Power H.264/AVC Decoder. In *Proceedings of IEEE International Symposium on VLSI Design Automation and Test*, pages 283–286, 2005.

[11] Tsu-Ming Liu, Ting-An Lin, Sheng-Zen Wang, Wen-Ping Lee, Jiun-Yan Yang, Kang-Cheng Hou, and Chen-Yi Lee. An 865 muW H.264/AVC Video Decoder for Mobile Applications. In *Proceedings of IEEE Asian Solid-State Circuits Conference*, 2005.

[12] Tsu-Ming Liu, Ting-An Lin, Sheng-Zen Wang, Wen-Ping Lee, Jiun-Yan Yang, Kang-Cheng Hou, and Chen-Yi Lee. A 125 μW Fully Scalable MPEG-2 and H.264/AVC Video Decoder for Mobile Applications. *Solid-State Circuits. IEEE*, 42(1):161–169, 2007.

[13] MIT Open Source Hardware Designs. `http://csg.csail.mit.edu/oshd/index.html`.

[14] Man Cheuk Ng, Muralidaran Vijayaraghavan, Gopal Raghavan, Nirav Dave, Jamey Hicks, and Arvind. From WiFI to WiMAX: Techniques for IP Reuse Across Different OFDM Protocols. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[15] Iain E.G. Richardson. In *H.264 and MPEG-4 Video Compression*. John Willey & Sons, 2003.

[16] Shen-Yu Shih, Cheng-Ru Chang, and Youn-Long Lin. A Near Optimal Deblocking Filter for H.264 Advanced Video Coding. In *The 11th Asia and South Pacific Design Automation Conference*, 2006.

[17] Lochi Yu, Samar Abdi, and Daniel D. Gajski. H.264 tlm in systemc for shared bus platform, January 2007.