A CONFIGURABLE H.265-COMPATIBLE MOTION ESTIMATION ACCELERATOR ARCHITECTURE SUITABLE FOR REALTIME 4K VIDEO ENCODING

By

MICHAEL BRALY
B.S. (Harvey Mudd College) May, 2009

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair, Dr. Bevan M. Baas

Member, Dr. Rajeevan Amirtharajah

Member, Dr. Soheil Ghiasi

Committee in charge
2015

Abstract

The design for a second generation motion estimation accelerator is presented and demonstrated as suitable for H.265/HEVC (MEACC2). Motion estimation is the most computationally intensive task in video encoding, and its share of the processing load for video coding has continued to increase with the release of new video formats and coding standards, such as Digital 4K and H.265/HEVC. MEACC2 has two 4 KB frame memories necessary to hold the ACT and REF frames, designed using a Standard Cell Memory technique, with line-based pixel write, and block-based pixel accesses. It computes 16 pixel sum absolute differences (SADs) per cycle, in a 4x4 block, pipelined to take advantage of the high throughput block pixel memories. MEACC2 also continues to support configurable search patterns and threshold-based early termination. MEACC2 is independently clocked, can sustain a 812 MHz operating frequency and occupies approximately 1.041 mm^2 post place and route in a 65 nm CMOS technology node. Taken together, MEACC2 can sustain a throughput of 105 MPixels/s while encoding the video stream johnny_60 with a hexagonal 'ABA' pattern with no early termination, as its worst performance, which is sufficient to encode 720p video at 110 frames per second (FPS). Multiple search algorithms are run against a battery of 6 video sequences using MEACC2. These runs demonstrate the adaptability and suitability of MEACC2 for video coding in H.265/HEVC at high throughput, and also demonstrate the efficacy and tradeoff present in a novel search pattern algorithm, 12-pt Circular Search.

Acknowledgments

I would like to thank my adviser, Professor Bevan Baas. In 2009, he was willing to take a chance on me, when it seemed like no one else would. His advice, teaching, and example, have helped me build something I am proud of, and the lessons I have learned at UC Davis have continued to help me in my life in industry. I would also thank my parents, who have supported me always, and have allowed me to forge my own path in life, one that I don't think any of us would have imagined when I was still growing up, out in East Davis. Thank you to Trevin, Aaron, John, Brent, and Eman. You guys are awesome, and were always willing to chat about research, even though I was the only one doing any sort of video processing at all! An additional thank you to Aaron, for taking the time to do the final synthesis and place and route flows, and then going above and beyond to play with the density settings to find the optimal P&R result. Finally, thank you Lizzie. For being so very patient.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The smartphone revolution is in full swing. Apple introduced the iPhone eight years ago, June 29th, 2007. Since then, Google has introduced the Android platform, and in 2013 an estimated 1 billion smartphones had been shipped worldwide. Each of these smartphones offers video capture and playback functionality. This rapidly growing market is driving even greater interest in fast video encode and decode functionality, while placing greater constraints on power budgets as even more functionality and sensors are brought onto the device. Additionally, the video being served onto smart devices is also available on PCs and new smart Televisions. At YouTube, a video streaming website, the number of videos served per day grew by 1 billion videos streamed between 2011 and 2012, to a total of 4 billion videos served per day.

A digital video stream consists of a series of still images, called frames which have a width and height, given in pixels. These frames are played back at a fixed rate, given in terms of frames per second (FPS). As the number of videos being served has grown, so has the size and quality of the video stream expected by customers. Television companies advertise the launch of their 4K products, which display frames as large as 7680 x 4320 pixels and YouTube supports 1080p videos delivered at 60 FPS.

Raw video streams tend to contain a large amount of redundant information, as each frame repeats every single pixel in the field of view even if nothing has changed. These raw video streams also require a tremendous amount of space, as each pixel requires at least several bytes of storage. Digital video compression reduces the size of the stored video file by eliminating redundant information while retaining enough of the original video stream so that it can be recreated on demand. The process is necessarily lossy, and designers trade off reconstructed video quality for storage space.

1.1 Project Goals

This work covers the design of a motion estimation hardware accelerator, named MEACC2, primarily for *inter*-frame motion estimation acceleration, with AsAP as a demonstration platform. As such the final device is expected to integrate cleanly with any compute platform which follows the general interconnect principles defined for AsAP2 and AsAP3. A key features of AsAP which makes it well suited as a test platform for video processing is the presence of fully-programmable independent processors and large on-chip shared memories. At the beginning the initial project requirements were defined as follows:

- Capable of real-time video processing in at least 1080p
- Compliant with the H.265 standard
- Capable of video processing in 4K formats
- Support for both built in and programmable search patterns
- Support for Full Search Pattern
- Explore the memory size vs. performance tradeoff in configurable accelerators
- Explore the use of Standard Cell Memories in AsAP based accelerator design
- Lay the groundwork for the development of an AsAP based H.265 Codec

1.2 Contributions

The time frame of this work extended further than initially expected, and so the main contributions include the following:

- The design and implementation of MEACC2, an H.265 capable hardware accelerator compatible with the 3rd generation AsAP interconnect, a circuit switched 16 bit dual-FIFO inter-block interface.
  - Complete RTL, written in Verilog HDL
  - Synthesized in 65 nm CMOS with a maximum frequency of 812 MHz post place and route
- The creation of matlab functional model of MEACC2, with the capability to generate test-benches for Post-Si validation.
- The introduction of a 12-point block-motion algorithm which fills the gap between high-cost/high-fidelity BMAs and low-cost/low-fidelity BMAs.

1.3 Overview

Chapter 2 introduces the fundamentals of digital video compression, including the motion estimation process. Chapter 3 covers the AsAP platform, features of interest, and how MEACC2 integrates with the whole system. Chapter 4 covers related work on motion estimation generally including other platforms such as FPGAs, ASICs, and CPU instruction set extensions. Chapter 5 introduces the MEACC2 architecture, including its instruction set, memory organization, and expected AsAP to MEACC2 interactions. Chapter 6 presents the MEACC2 datasheet, and post place and route die photo. Chapter 7 introduces the matlab model, its data structures, classes, and overall software architecture. Chapter 8 introduces the tradeoffs and performance estimations enabled by the matlab model. Chapter 9 summarizes this work's contribution, makes a few predictions, and outlines some ideas for future research and follow up.

Chapter 2

Digital Video Compression

The goal of digital video compression is to reduce the size of a video stream, by identifying redundant information, removing it, and replacing it with a scheme to recreate that information in the decompression step. There are two kinds of of redundancies: *inter*-frame redundancy exists between frames in a video stream, *intra*-frame redundancy exists within a single frame of a video stream. Another way to think of these two kinds of redundancy is to think of *inter*-frame redundancy as describing a repetition of data over *time* while *intra*-frame redundancy is describing a repetition of data over *space*. An object which is present throughout an entire scene would be an example of the kind of redundancy that *inter*-frame compression seeks to remove. A large area sky taking up most of the top-half of a scene would be the sort of information redundancy that *intra*-frame compression would remove.

Redundancy is a qualitative description of an effect that humans see. The computer must be able to *quantify* the similarity between two sets of images. This quantification process generates a *figure of merit* which the compute process can use to determine whether or not the two images are redundant enough to remove without significant loss of image quality. Two examples of Figures of merit are mean absolute error (MAE) and sum of absolute difference (SAD) [1]. These Figures of merit are applied to pixel differences between the images. In the video coding standards that this work addresses (H.264 and H.265), the accepted Figure of merit is SAD. The advantages and disadvantages of particular Figures

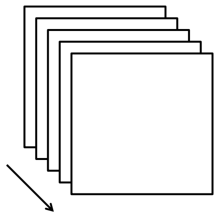


Figure 2.1: Inter-frame redundancies exist between multiple frames of a video stream

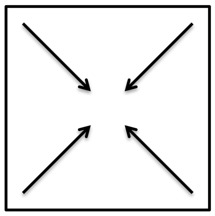


Figure 2.2: Intra-frame redundancies exist within a single frame of a video stream

of merit are beyond the scope of this work. Figure 2.3 gives a worked example of how to compute the SAD of two blocks of pixels.

For any two blocks of pixels in the pixel arrays A and R, of width N and height M the SAD is given:

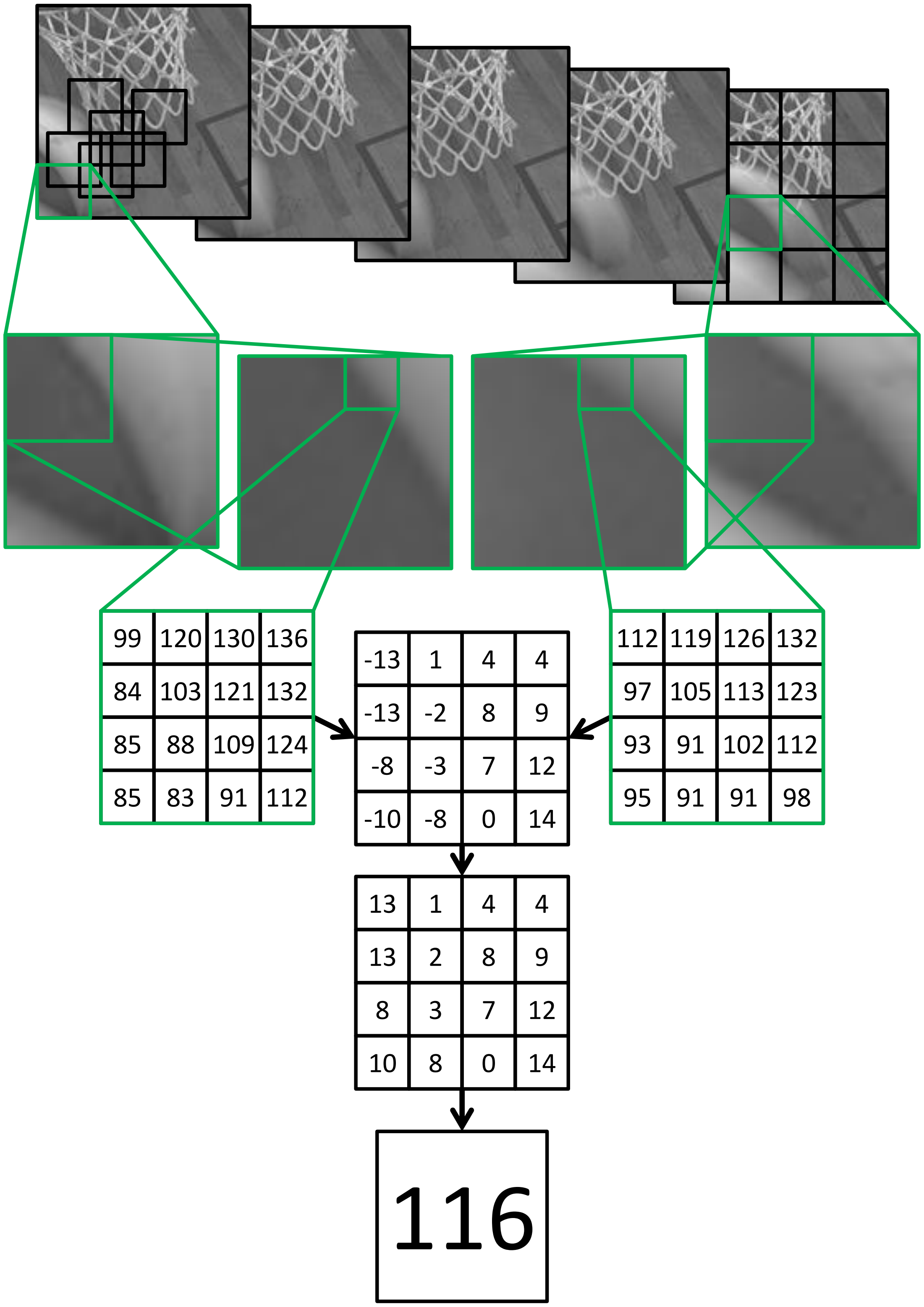
$$SAD(A, R) = \sum_{i=0}^{N} \sum_{j=0}^{M} |A(x + i, y + j) - R(x + i, y + j)|$$

2.1 Video Coding Terms in Historical Context from H.261 to H.265

The standards can be viewed as a progression of terms and techniques. Video coding techniques have been largely accretive over the years, where each new standard adds additional coding tools to the standard and old coding tools continue to remain relevant. This has lead the computational complexity of video coding to scale not only along the axis of total number of pixel samples processed, but also along the axis of which coding features are supported by a particular encoder.

2.1.1 H.261

Introduced the concept of the macroblock. Each macroblock is a 16x16 array of luma samples and two corresponding 8x8 arrays of chroma samples, using 4:2:0 sampling and a YCbCr color space [2]. The coding algorithm uses a hybrid of motion compensated inter-picture prediction and spatial transform coding with scalar quantization, zig-zag scanning



99	120	130	136
84	103	121	132
85	88	109	124
85	83	91	112

-13	1	4	4
-13	-2	8	9
-8	-3	7	12
-10	-8	0	14

112	119	126	132
97	105	113	123
93	91	102	112
95	91	91	98

13	1	4	4
13	2	8	9
8	3	7	12
10	8	0	14

Figure 2.3: Example SAD computation

and entropy encoding. The standard only defined the video decode process, the encoding was left open. This meant that encoders could pre-process data before encoding, and decoders could post-process after decoding - deblocking filters were a form a post-processing to reduce the appearance of block-shaped artifacts. It also only had support for integer-valued motion vectors. Transform coding used an 8x8 Discrete Cosine Transform to reduce the spatial redundancy [2].

2.1.1.1 Color Space

YCbCr describes the color space. *YUV* describes a file that uses YCbCr for color encoding. YCbCr breaks the color space into luma (Y, brightness) and chrominance (UV, color) components. Black and white only images have only luma components. Luminance is denoted Y and luma by Y. *Luminance* is perceptual brightness, what the eye/brain actually sees. *Luma* is electronic brightness eg. a voltage or a digital value.

2.1.1.2 J:a:b Sampling

A quick way of describing the subsampling scheme for a region J pixels wide and 2 pixels high. The number of chrominance samples (Cr, Cb) in the first (even) row is denoted a, while the number of chrominance samples in the second (odd) row is denoted b [2]. Subsampling takes advantage of the fact that human vision cares more about brightness than color, and so coding techniques save bits by sampling the chrominance less carefully than the luminance.

2.1.1.3 Entropy Encoding

Entropy encoding describes a wide range of lossless data compression schemes, which are data independent. Huffman and arithmetic coding are examples of entropy encodings. If the entropy characteristics of a data stream can be approximated beforehand, it can then be devolved into a static code, allowing data storage without any loss of fidelity [2].

2.1.2 H.262

Introduced support for both interlaced and progressive video systems while dividing frames into 3 classes, I-frames (intra-coded), P-frames (predictive-coded), and B-frames (bidirectionally-predictive-coded). Allows for a number of subsampling schemes with 4:2:0 continuing to be the norm.

2.1.2.1 Interlaced and Progressive Video

Interlaced video frames divide the image into two parts, a top-field and a bottom-field consisting of the odd numbered horizontal lines and even numbered horizontal lines respectively. Fields are transmitted and decoded in pairs. Progressive video means that fields and frames are the same, the image is not divided [2].

2.1.2.2 Intra-Coded Frames (I-Frames)

An Intra-coded frame (I-Frame), is a compressed version of a raw frame that uses information from that frame only [2]. An I-frame then, can be decoded independently of its neighboring frames. Typically the I-frame is broken into 8x8 pixel blocks, the DCT is applied, the results quantized (this is where data fidelity is lost) and then compressed using run-length codes and other similar techniques.

2.1.2.3 Predictive-Coded Frames (P-Frames)

P-frames can get a more compact compression than I-frames because they make use of data from previous I and P frames [2]. To generate a P-frame, the previous reference frame (either an I or P frame) is kept and the current frame is broken into 16x16 pixel macroblocks. Then, for each 16x16 macroblock in the current frame, the reference frame is searched for the smallest distortion match. The offset of the smallest distortion match is saved as a motion vector, and a residual between the two blocks computed. If not suitable match is found, the macroblock is treated like an I-frame macroblock.

2.1.2.4 Bidirectionally-Predictive-Coded Frames (B-Frames)

B-frames are never reference frames and use information from both directions (from either I or P frames) [2]. They generally get an even more compact resulting compression than a P frame.

2.1.2.5 Group of Pictures (GoP)

A series of I, B, and P frames. Useful for packing sets of frames together to be sent/handled as a group [2]. In H.262 usually every 15th frame is an I frame, but this is a flexible part of the standard. An example group of pictures might contain the following set of I, P, and B frames: IBBPBBPBBPBB.

2.1.3 H.264

Further extended H.262 with new ways to do transforms, quantizations, and encodings, greater macroblock size coverage, and introduces new loss-resilience features.

2.1.3.1 Variable Block-Size Motion Estimation (VBSME)

Macroblocks can take on a number of different sizes in VBSME schemes, instead of being fixed to 16x16. The valid sizes and shapes are:

- 16x16
- 16x8
- 8x16
- 8x8
- 8x4
- 4x8
- 4x4

These new shapes are used to get finer grain segmentation around moving regions in the video stream [2]. A macroblock can now be made up of multiple blocks (eg, 4 8x8 regions instead of 1 16x16 region) and each of those blocks can have their own motion vector. So each macroblock can have up to 32 motion vectors (a B macroblock with 16 4x4 partitions).

2.1.3.2 Sub-Pixel Precision

Quarter-pixel precision is supported for greater accuracy. Chroma samples support $\frac{1}{8}$ pixel precision since chroma is expected to be sampled at half the rate of luma in 4:2:0 mode.

2.1.3.3 Context-Adaptive Binary Arithmetic Coding and Variable-Length Coding (CABAC and CAVLC)

CAVLC and CABAC are used to code already quantized transform coefficient values [2]. There is a complexity tradeoff between CAVLC and CABAC, where CABAC can compress more efficiently than CAVLC, but is more computationally intensive [3]. CABAC was introduced in 2001 [4] and CAVLC in 2002 [5] and both were integrated into the H.264 standard recommendation [3].

2.1.3.4 Exponential Golomb Coding (Exp-Golomb)

Exponential Golomb coding is another form of coding used for the more general forms of the standard (CABAC and CAVLC target primarily the image data, one would use Exp-Golomb for header tags and other metadata) [2].

2.1.4 H.265

Up to double the compression effectiveness of HEVC (bitrate based) and the target is to allow up to 1000:1 compression for easily compressible video streams. Designed with the assumption of progressive video, so no explicit support for interlaced video [6].

2.1.4.1 Coding Tree Units (CTUs) and Coding Tree Blocks (CTBs)

Coding tree units are analogous to the macroblocks of previous standards [7]. In 4:2:0 the CTU contains 3 CTBs, 1 luma CTB and 2 chroma CTBs. The size of the luma CTB is given $L \times L$ where $L = (16, 32, 64)$. The CTBs can be partitioned into smaller subunits called Coding Blocks (CBs), while the CTU is partitioned into Coding Units (CU) [7]. A CU typically contain the luma CB and the chroma CBs, for a total of 3 CBs. Each CU also has associated prediction units (PUs) and a tree of transform units (TUs). Prediction units have associated prediction blocks (PBs) ranging in size from 64x64 to 4x4. The transform units have associated transform blocks (TBs). There are transform functions defined for square TBs of 4, 8, 16, and 32 pixels.

Fundamentally, motion estimation hardware deals with the lowest level coding block. There are more possible block sizes, many used in asymmetrical motion prediction (AMP) [7].

2.1.4.2 Allowed Prediction Block Sizes

- 64x64
- 32x64
- 64x32
- 32x32
- 16x32
- 32x16
- 16x16
- 8x16
- 16x8
- 8x8
- 4x8

- 8x4

2.1.4.3 AMP Prediction Block Sizes

Support for asymmetrical motion prediction enables blocks oriented in both $N \times (\frac{N}{4})$ and $N \times (\frac{3N}{4})$ directions [7]. Reported experimental results demonstrate a 1% improvement in bit-rate at the cost of 15% additional encoding time [8]. The standard also establishes 4x8 and 8x4 as the minimum sizes for a prediction block (PB), and so AMP cannot be used for values of N smaller than 16.

- 16x64
- 48x64
- 64x16
- 64x48
- 8x32
- 24x32
- 32x8
- 32x24
- 4x16
- 12x16
- 16x4
- 16x12

2.1.4.4 Motion Vector Signaling

Advanced motion vector prediction (AMVP) is used to pick probably candidates based on data from adjacent prediction blocks and the reference picture. There is also a merge mode that allows MVs to be inherited from temporal or spatially neighboring PBs.

The prediction step helps guide the search, if using a pattern search, or pick a better search area candidate if using a full search [7].

2.1.4.5 Motion Compensation

Quarter-sample precision is used for the MVs and 7 to 8 tap filters are used for interpolation of fractional-sample positions. H.264 used six-tap filtering with half-sample precision and linear interpolation to gain quarter-sample precision [2].

2.1.4.6 Prediction Modes

Intrapicture prediction supports 33 directional modes, plus planar and DC modes (total of 35 modes).

2.1.4.7 Context Adaptive Binary Arithmetic Coding (CABAC)

Similar to CABAC from H.264 but with several throughput-optimizations for parallel processing architectures and compression performance [7].

2.2 H.264 and H.265 in Depth

IEEE promulgates a standard for video coding referred to as H.264 [3], and since 2011 has begun to promulgate a new standard, H.265 [9]. These standards allow the people who design hardware to encode video and the people who design hardware to decode video to be two separate subsets. There are additional standards which are also used for this purpose, Google, for instance, promulgates the V8 and V9 standards, which are roughly equivalent to H.264 and H.265 . The primary goal of the H.265 coding standard was to increase the compression efficiency of video streams by 50% without negatively impacting the overall video quality [10]. Initial analysis of the H.265 standard indicates that the standard meets that goal, with demonstrations on multiple video streams [11]. Each of these standards contain a set of tools to use to compress a video stream. For H.265, the various effects of each of these tools has been broken out into different levels, trying to define a smooth tradeoff curve between computational complexity and final result quality [12].

2.2.1 Macroblocks and Coding Units

Motion estimation which makes use of variable block sizes is referred to *Variable Block Size Motion Estimation* (VBSME) [13]. H.264 made use of groups of pixels, called *macroblocks* to perform the encoding operation. Instead of matching pixels, the standard calls for blocks of pixels to be matched against other blocks of pixels. This technique was carried forward into the H.265 standard, in the form of *coding units* contained within a data-structure called a *coding trees*. For the purposes of this work, the important thing to know about both macroblocks and coding units, is that they can vary in size during operation. Different parts of a video stream can be coded with all the same size of block, or different sizes of blocks. Figure 2.4 gives a graphical representation of pixel block shapes supported in H.265 and H.264 compliant coding. There are a sit of shapes in H.265 referred to as the asymmetrical motion prediction vectors. These include all shapes that are not square or 1:2 ratio rectangular. Further investigation into AMP showed that there was only a 0.8% coding efficiency gain for a 14% increase in coding effort. Therefore, MEACC2 does not make use of AMP shapes. Figure 2.5 shows the AMP shapes which are not supported by MEACC2.

There were investigations into how to make the most effective macroblock divisions for a particular frame [14] and how to make those decisions quickly [15], targeting the H.264 application space. That research has been carried forward into coding trees.

2.2.2 Coding Trees

As part of the shift to H.265 , groups of pixels are grouped at multiple levels of hierarchy in a coding tree. A basic coding tree is very similar to the H.264 understanding of the frame, which contains many macroblocks of various sizes. In a coding tree, each frame has a coding tree, that coding tree has branches of various sizes, those branches have blocks of pixels of a size based on the depth of the branch node. Therefore, quick decisions on how to divide the coding tree result in faster compression speed, though an ideal coding tree would be necessary for maximum compression efficiency. An initial investigation into how to merge coding trees, also demonstrates that coding tree structures were 3% more effective

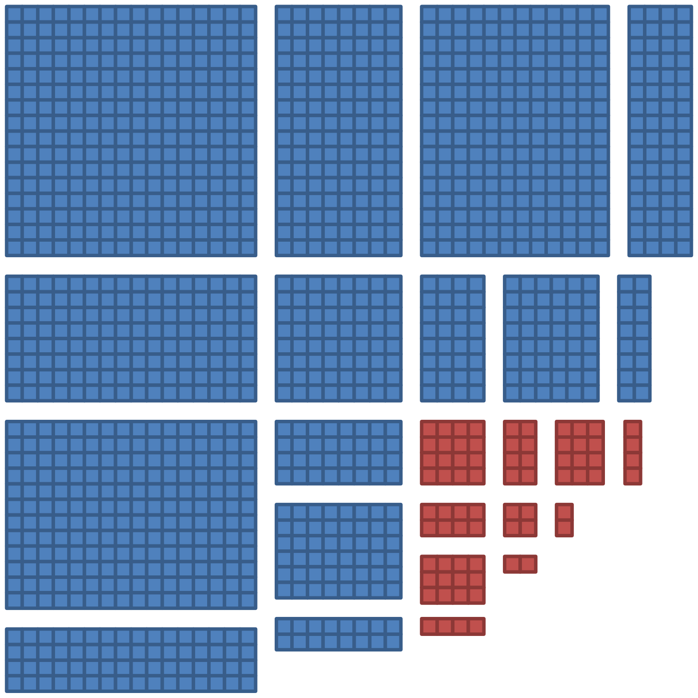


Figure 2.4: Shapes supported in H.265 and H.264. each square represents a 4x4 block of pixels. Blue shapes are only supported in H.265

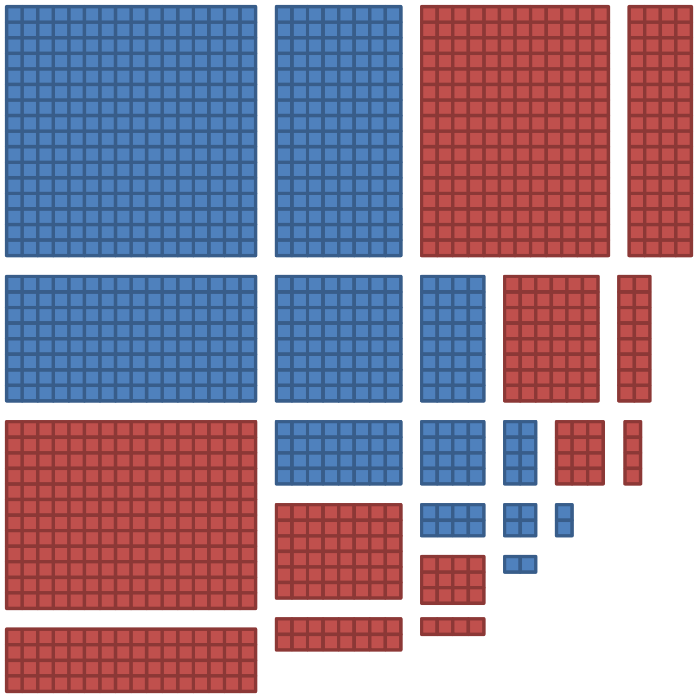


Figure 2.5: Shapes supported in H.265 including AVMP. each square represents a 4x4 block of pixels. Red shapes are AMVP shapes and are not supported by MEACC2

than the equivalent direct mode in H.264 [16]. There has also been work done on how to predict the final shape of the coding tree, and using such prediction techniques combined with other hardware saving techniques have demonstrated a 2x performance increase and a 35% energy cost decrease [17].

2.2.3 Slices and Tiles

Tiles are a technique available in H.265 to leverage parallel hardware [18]. These are similar to the slice technique used in H.264 [7]. Previous work with slices demonstrated that the overall coding process could be split into up to 16 slices with linear efficiency gains per slice added [19]. The expectation is that each tile is processed in parallel, and then information from each of the tile processing jobs can be used to refine the compression in future frames. In the meantime, from a hardware perspective, each tile can be treated as a separate, and independent unit, for much of the initial processing, including motion estimation. Our work then, can target a proof of concept of a single *tile* which can then be extrapolated outwards to video streams of significantly larger size. Tiles are not free, and does come with a cost in final video stream quality. The tile partition information is encoded in the final video stream, decoders then parse the tile information and use it to reassemble the stream at decode time.

2.3 Block Motion Algorithms

Block motion algorithms (BMAs) encompass a class of search algorithms for finding the smallest SAD match for a set block of pixels. They are invariant with regards to the total size of the block of pixels, so the same algorithm can be applied to an 8x8 block of pixels and a 64x64 block of pixels. The design space of BMAs trades the total number of pixel blocks checked, for the *expected* fitness of the final block match.

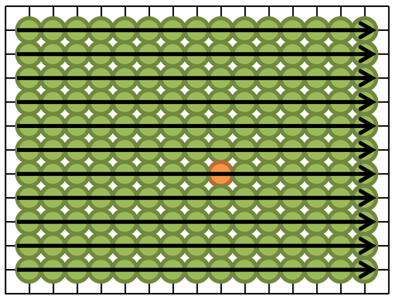
2.3.1 Full Search

Full search is the simplest block motion algorithm, checking all possible blocks in a given search space. It guarantees the smallest distortion match within a search space is found, but it also costs the maximum amount of compute to find that match. It can be further enhanced with early termination logic so that the search is ended early if the smallest distortion match found so far is of a minimum threshold of quality, or with decimation, where the total number of points checked is reduced in an invariant manner (checking every other candidate in a full search would be a decimation by 2). Since it guarantees the highest quality match in a frame, the Full Search is a useful tool for determining the maximum quality of matches in a video stream, in order to quantify the quality degradation of search patterns which use less compute. Three worked examples of a full search implementation are given in Figure 2.6.

2.3.2 Pattern Search

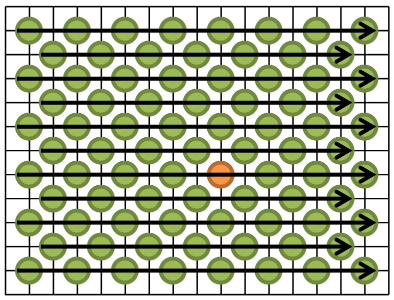
Pattern searches are also block motion algorithms, but they extend the full search by reducing the total number of block candidates checked, while still managing the reduction in match quality to an acceptable level. The acceptable level of degradation is dependent on the application space. These patterns can be thought of as an extension of the decimation technique used with full search algorithms. Instead of systematically checking every single possible candidate in a search range, a pattern search only checks a subset of those possible points. Some algorithm, which varies depending upon the pattern search, is used to determine which points to check, and in what order. Center-biased search patterns take as their starting point the position of the original block being compared. This follows from an observation, that if things in the video stream are static, the objects in that image do not move over time, and *spatially* local blocks would be good probable matches for the search between frames.

Once the initial point is checked, if the threshold value is not met additional points are checked. This is where the various center-biased search patterns begin to distinguish themselves from each other. The center not being a suitable match would imply that there has been some movement within the frame. A place to continue searching then, would be around the initial point. Checking all the points surrounding the center of the search would defeat part of the purpose of a search pattern (dramatically reducing the number of points checked), so the patterns are designed to capture as many possible motion directions,



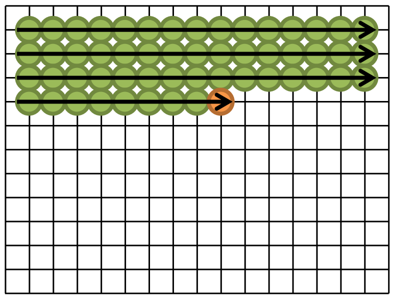
Full Search

Full search patterns check every possible point in the search area in a fixed order. In this example, the all the green points are checked, and the orange point is found to have the best SAD.



Full Search with Decimation

In a decimated Full Search, not every single point is checked, but rather only a regular subset of the points. The search does however, still check every non-decimated point in the search area, so even though the orange point has the best SAD, the search continues.



Full Search with Early Termination

In a Full Search with early termination, the search is ended when the first point which has a better SAD than a given threshold is found. This can be combined with decimation, but in this example it is not.

Figure 2.6: Different kinds of Full Search patterns

while still keeping the total of points checked to a minimum. A cross shaped search pattern would only capture motion in four directions, while a diamond shaped pattern can capture movement in up to eight directions. Each pattern is suitable for different kinds of motion. If a video stream's general motion behavior is known ahead of time, or that the class of video streams dealt with are known, it is possible to craft a more efficient search pattern that is application specific.

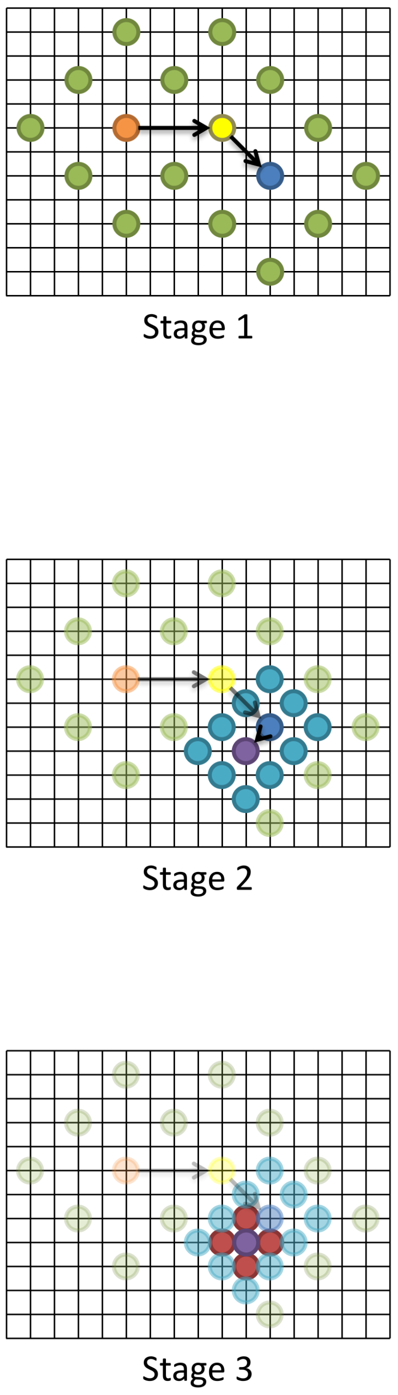
An example of a three stage, center-biased, diamond search pattern is given in Figure 2.7.

2.4 Video Formats

Each iteration of a codec, such as H.264 and H.265 give a series of levels which a video may be encoded in. These levels roughly represent the total bitrate that an encoder or decoder must be able to handle. However, these levels are not how consumers and designers actually interact with video. They interact with video formats, given in resolution and framerate. A number of commonly used video formats are given in Table 2.1, and the levels for H.265 are given, along with example formats and framerates in Table 2.2.

Table 2.1: A selection of video formats

General Use	Name	X	Y	Pixel Count per Frame
Video Conferencing	QCIF	176	144	25344
	CIF	352	288	101376
Digital Monitors / Televisions	480p	640	480	307200
	720p	1280	720	921600
	1080p	1920	1080	2073600
	2160p	3840	2160	8294400
	4320p	7680	4320	33177600
Theater	Digital 4K	4096	2160	8847360
	IMAX	5616	4096	23003136



In stage 1, beginning from the search center (orange), 8 points in each direction are checked in a diamond pattern.
As points with better SADs are found (yellow, then blue), the pattern moves and the diamond search is repeated.
When no better point is found, the search changes to stage 2.

In stage 2, starting from the center found in stage 1, a more compact diamond pattern is used.
Another candidate point with a better SAD is found (purple), and the pattern moves, as in stage 1.
When no better point is found, the search changes to stage 3.

In stage 3, the final pattern used is a compact cross pattern of 4 points.
No better point is found, and as this is the final pattern of the search, the search terminates.

Figure 2.7: Example 3-stage pattern search

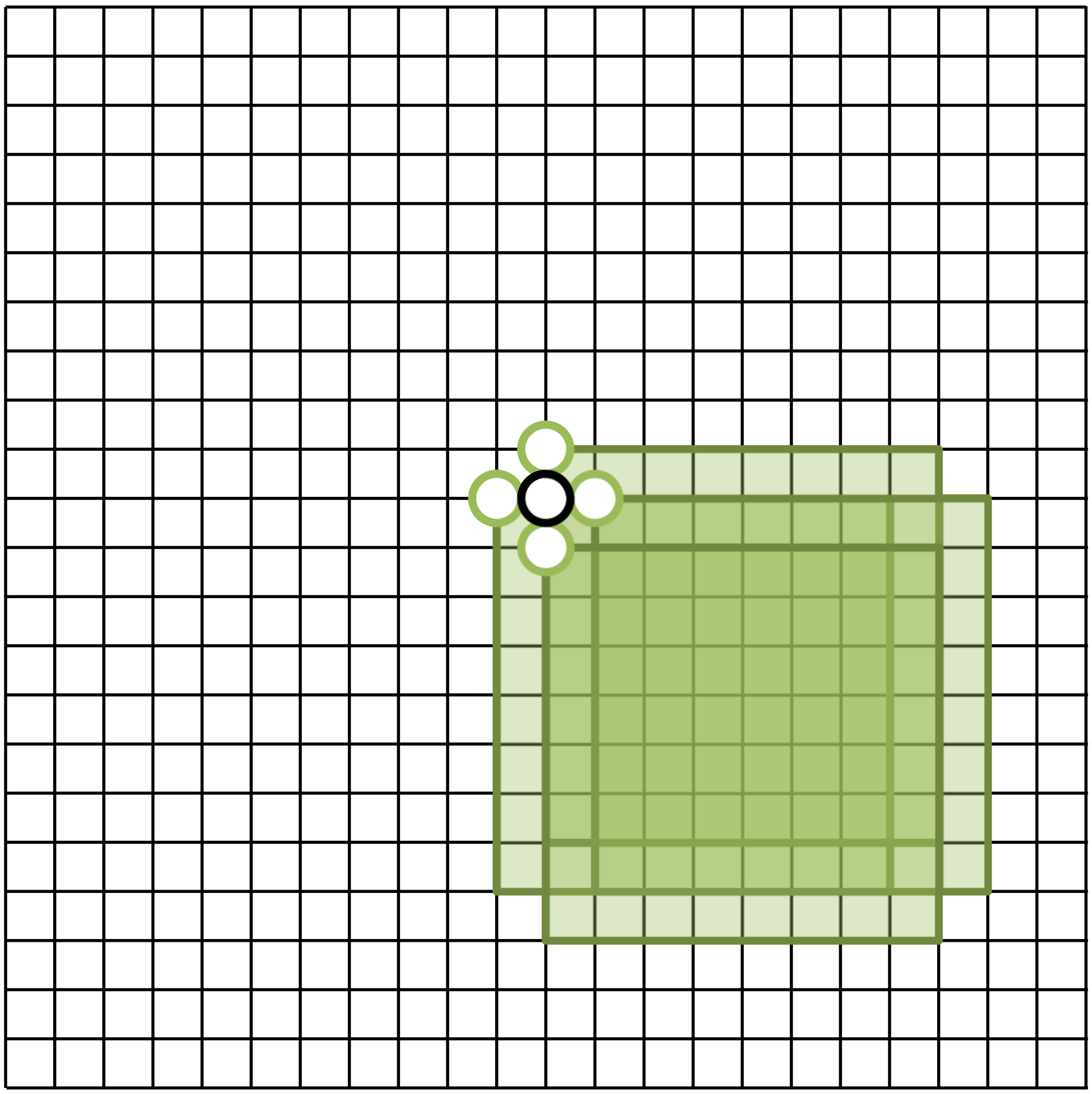


Figure 2.8: Relationship between search pattern points and pixel blocks

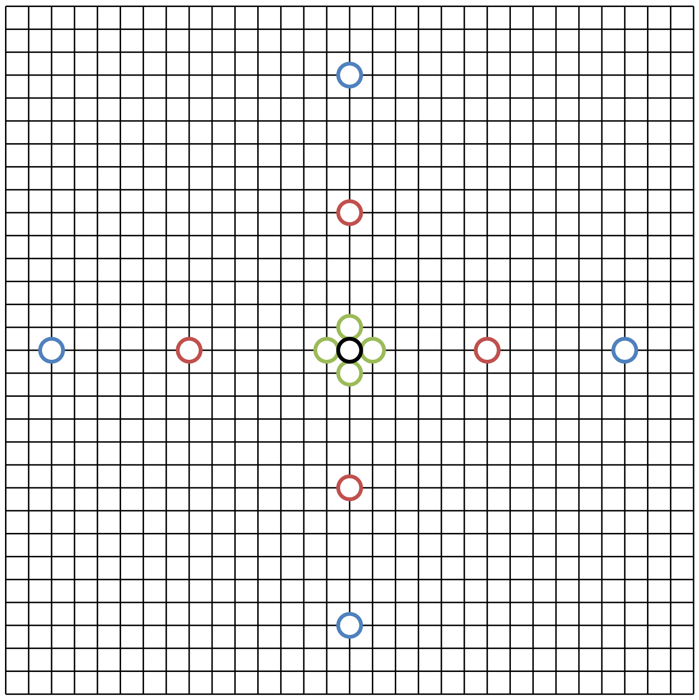


Figure 2.9: Cross patterns of varying width

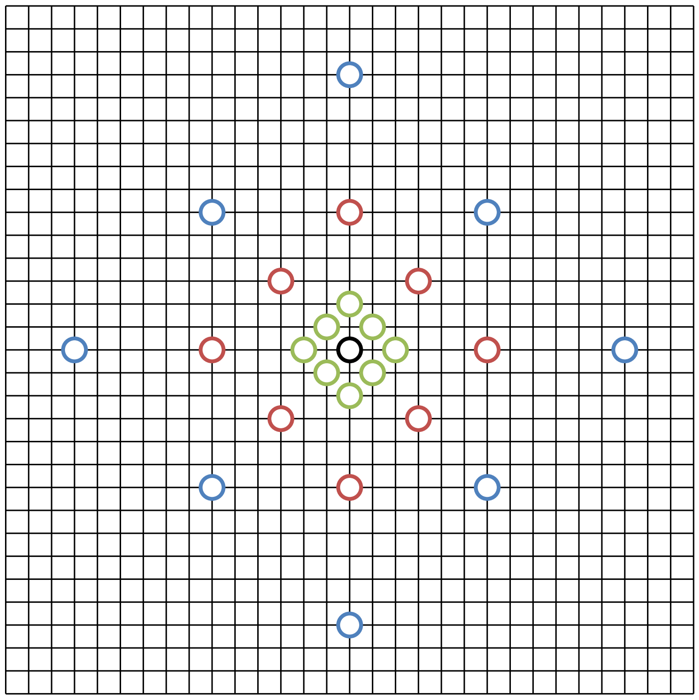


Figure 2.10: Diamond patterns of varying width

Table 2.2: Coding levels in H.265/HEVC

Level	Max Picture Size	Max Sample Rate	MaxSz FPS	Format	FPS
1	36864	552960	15.00	QCIF	15.00
2	122880	3686400	30.00	CIF	30.00
2.1	245760	7372800	30.00	CIF	60.00
3	552960	16588800	30.00	480p	54.00
3.1	983040	33177600	33.75	720p	36.00
4	2228224	66846720	30.00	1080p	32.24
4.1		133693440	60.00	1080p	64.47
5	8912896	267386880	30.00	2160p	32.24
5.1		534773760	60.00	2160p	64.47
5.2		1069547520	120.00	2160p	128.95
6	35651584	1069547520	30.00	4320p	32.24
6.1		2139095040	60.00	4320p	64.47
6.2		4278190080	120.00	4320p	128.95

2.5 Decoders

Initial development on H.265 decoders is underway. Developers are beginning to grasp the overall differences between H.264 and H.265, and the important differences for those working with decoders were laid out as follows [20]:

- Macroblocks are replaced by Coding Units which support a maximum size of 64x64 pixels.
- Prediction Unit shapes may be asymmetrical
- Transform Units may be up to 32x32 pixels
- Up to 33 intra prediction modes
- Advanced skip modes and motion vector prediction
- New Adaptive Loop Filter (ALF)
- A Sample Adaptive Offset (SAO) is present after the deblocking filter
- Tools oriented for parallel processing

Work on high definition video decoders has continued as well, with decoders managing 4096x2160 at 60 FPS in 90 nm CMOS [21]. These decoders demonstrate that even with increasing encoder efficiency, the market and devices that would require that coding efficiency improvement exist and continue to develop.

Chapter 3

The AsAP Platform

MEACC2 was developed to target the AsAP platform as its primary test platform, but AsAP as a platform encourages the development of loosely coupled, and therefore portable accelerator designs. AsAP is a fine-grain many-core architecture originally designed for DSP architectures, with a focus on scalability and power efficiency [22]. AsAP arrays consist of independently clocked processors communicating over dual-clock FIFOs, with each processor having its own instruction and data memories and executing a general instruction set [23], as shown in Figure 3.1. AsAP fabrics can be further enhanced with the addition of large memories or dedicated accelerators. These memory blocks and accelerators are connected to the array though those same dual-clock FIFOs, typically adjacent to two processors, as shown in Figure 3.2. The first generation of the AsAP platform contained 36 processors fabricated in 0.18 μm^2COMS [24], with a maximum operating frequency of over 600 MHz [25], and the second generation of the AsAP platform contained 167 full processor cores in 65 nm [26] with a maximum operating frequency of 1.2 GHz[27], and with enough compute to host a 1080p H.264 baseline residual encoder without any dedicated hardware [28].

3.1 Generalized Interface

The primary form of communication in the array is a 16b wide dual-clock domain FIFO [29]. The FIFOs between each node in the array allow for every processor and

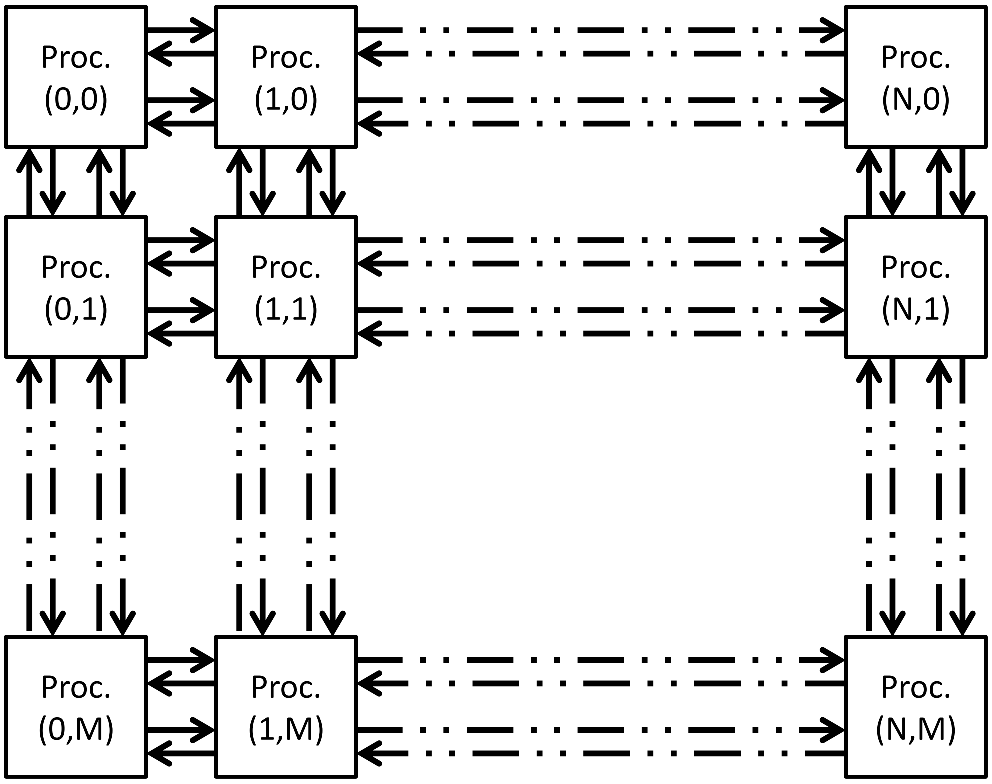


Figure 3.1: An MxN AsAP array

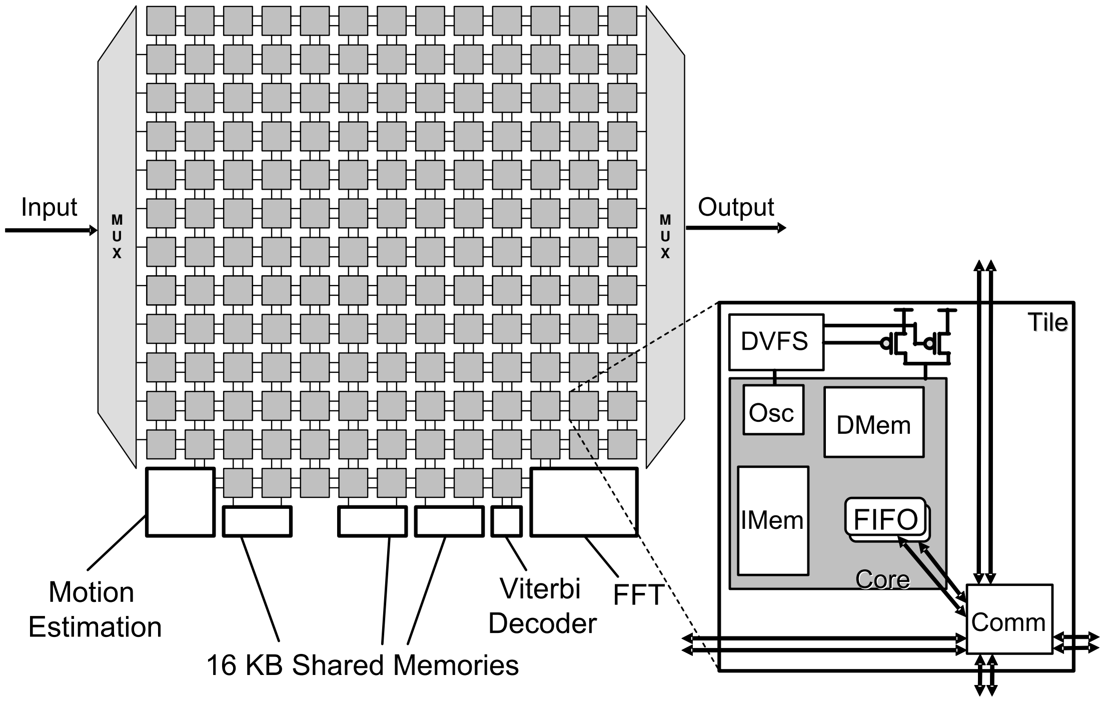


Figure 3.2: A 167 core AsAP Array with big memories and accelerators

accelerator to be independently clocked. This also means that the accelerator design can target high frequency operation without worrying about the design of the rest of the array for high frequency operation as well. Additionally, the general interface of 16b words means that the accelerator can be easily modeled at a high level, as with the matlab model in Chapter 7.

3.2 Scalable Mesh

The scalability of the 2D mesh interconnect of an AsAP array means that as new technology nodes become available, the additional area can be put to productive use. The second generation AsAP array had a total of 167 processors, big memories, and three different kinds of hardware accelerators [30] including an FFT engine [31], and a previous generation motion estimation engine and the associated software encoder to take advantage of that accelerator [32]. With such scalability inherent to the platform, the priority is placed on developing accelerators which can also be scaled, as the latest iterations of the AsAP platform have a current maximum of 1000 processors in 32 nm [33]! Therefore, the MEACC2 was designed to make use of the Tiles paradigm introduced in H.265, which allows for the work of coding a video stream to be partitioned by subdividing the image and processing those sub-images in parallel [6]. Additionally, tools to map applications and the supporting software to take advantage of an accelerator to the device have already been developed and tested in other applications [34].

3.2.1 Circuit Switched Network

The AsAP platform also allows for connections beyond nearest neighbor using a low-cost circuit optimization for stable long-range links [35]. These long-range links, incorporated into a reconfigurable circuit-switched network [36], allow AsAP networks to host applications on fewer cores than an initial design would suggest [37]. Further research into the design of the packet routers used in the circuit switched network resulted in a bufferless router design with 60% greater throughput [38], and an advanced packet router with 7% savings in total energy expended per-packet [39]atran:vcl:phdthesis. These advances allow for AsAP based platforms to make heavy use of inter-processor communication links, suitable for streaming large amounts of data between nodes, such as found in video coding.

3.3 On-Chip External Memory

The large memories, which can be tiled into the AsAP array, ensure that there is sufficient memory to cache an entire frame on-chip. These large memories are accessed just like an accelerator or a processor, across the 16b dual-clock FIFOs [40]. The large memories also make use of a priority service scheme, which could be useful if multiple MEACC2 instances were being serviced by the same memory [41]. Therefore, MEACC2 can focus on solving the smaller problem of which memory to keep local to the computation. The line-based big memory also complements well the block-based memory architectures put forward for accelerator design, and so combines the advantages of both systems, a block-based memory for local pixel data, and a line-based raster-scan compatible large memory for the initial storage of frame memory. Since both the memory and the accelerator can scale alongside the AsAP array, the overall system is scalable to larger video streams.

3.4 Power Scaling

The globally asynchronous, locally synchronous (GALS) architecture allows for voltage and frequency scaling to be used at a fine-grain level to capture power savings not available to monolithic architectures [42], although it introduces some additional, but surmountable challenges in the design of the processor tiles [43]. Designing a stand-alone accelerator using the FIFO based architectures allows the MEACC2 to be part of systems that take advantage of these advances, including recent optimization techniques making use of genetic algorithms for dynamic load distribution [44].

Chapter 4

Related Work

H.264/AVC encoding has been codified since 2003 [3], and so there exist solutions along the entire spectrum of circuit-based research from the last 12 years. These solutions range from general CPU code, dedicated instruction sets, FPGAs, programmable many-core arrays, and application specific ICs.

4.1 Early Termination

Early termination techniques, broadly described, set a threshold value for the final SAD result and then terminate the search once that threshold is met. Compared to a full-search implementation, a similar implementation with early termination reduced total operation count by 93.29%, reduced memory accesses by 69.17%, and increased the total machine cycles by 220%, but did not address the effect on final image quality [45]. Further work on early termination found that a 72% reduction in memory bandwidth could be achieved with a bitrate increase of 1.25% on a 2D systolic array with a search range of ± 16 [46]. An additional investigation into the benefits of early termination found that using such a scheme, on average, reduced total memory bandwidth by 20%, increased bitrate by 0.79% and reduced PSNR by an additional 0.02 dB across a search range of ± 128 [47].

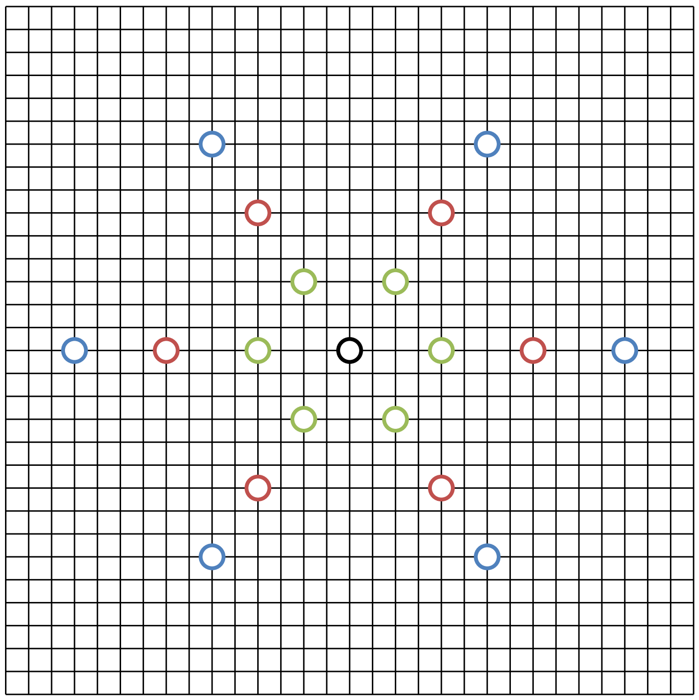


Figure 4.1: HexA

4.2 Search Patterns

Diamond search patterns have been built into dedicated estimators, where repeated repetitions of the diamond pattern can manage 1080p video frames at 55 frames per second [48]. The number of points in a particular search pattern directly effects its computational complexity, but the cross-based patterns miss diagonal movement. Purnachand looked into the hexagonal pattern, recognizing that there are two types, called now HexA and HexB, with examples in Figure 4.1 and Figure 4.2. Further work on search patterns have lead to the novel back and forth hexagonal search patterns of type A and B, such as HexABA and HexBAB, which save 23% number of points checked versus the diamond patterns used in other accelerators [49]. Examples of HexABA and HexBAB are shown in Figure 4.3 and Figure 4.4.

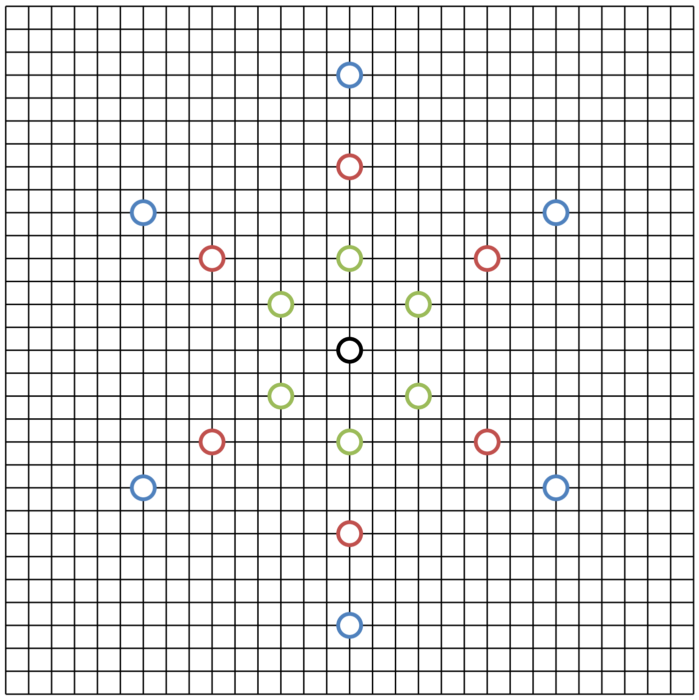


Figure 4.2: HexB

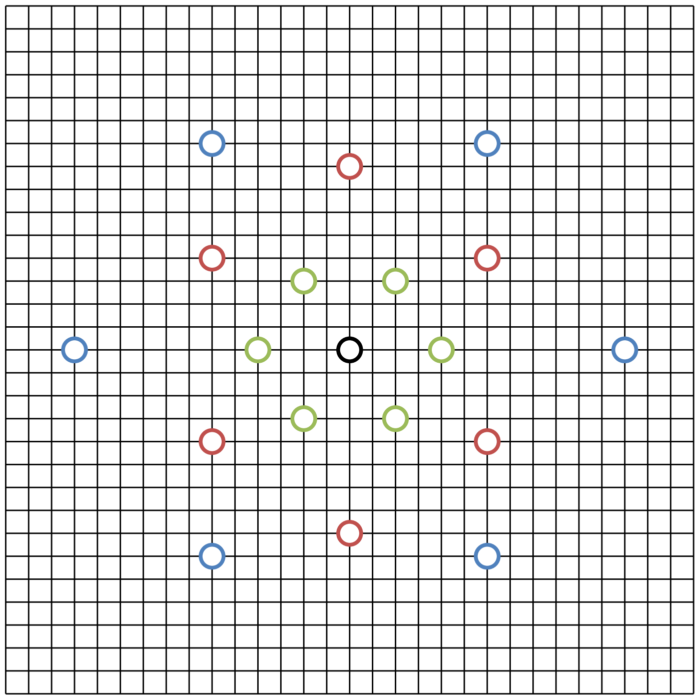


Figure 4.3: HexABA

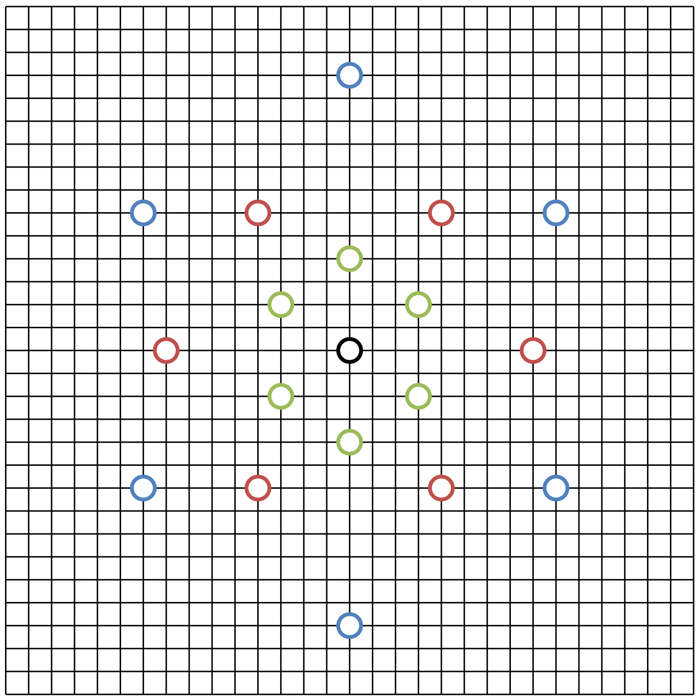


Figure 4.4: HexBAB

4.3 Frame Memory

The question of frame memory, and how much to have present in an accelerator, is a common theme in accelerator design. It is possible to have sufficient memory to contain the entire reference frame, but this doesn't scale well, as each the memory required increases linearly with the total number of pixels, but the total number of pixels increases quadratically with regards to image dimensions. Initial attempts to contain the scaling issue concluded that three levels of memory hierarchy was ideal for the reference frame memory [50]. Others grappled with how much reuse was actually possible, and posited a 2D systolic array which had the ideal memory reuse, but leaves out the total area required by their potential designs [51].

If the memory accesses are not single-access, then how that memory is accessed becomes significant. Block pixel comparisons imply that the memory architecture should support block pixel accesses, moving beyond the line-access patterns inherent to array-based pixel storage. A block-addressed memory space can be constructed on both ASICs and FPGAs with minimal addressing overhead [52]. An FPGA design makes use of modulo math to create pixel-block addressable memories on FPGAs which, in the worst case, have 1.2x memory access time, 1.47x the area, and 1.8x the power as compared to line-access architectures [53]. Further research by the same group found that by permuting the data as it moves into and out of the block-based memory mitigates the downside of the previous design and results in a memory architecture suitable for real-time 1080p video processing [54].

Further work in the FPGA space by Chandrakar resulted in a parameterizeable design for motion estimation which could achieve up to 275 FPS on 1080p video sequences [55]. This design, however, needed to be reimplemented for each video and block size. Therefore, with the relatively long configuration time for FPGAs (order of magnitude seconds to minutes, depending upon the programming interface), his solution is practical for fixed block size execution, but not for variable block size motion estimation. His work might be worth revisiting if programming times for FPGAs drop sufficiently, or if each parametrized design ends up being similar enough to each other to take advantage of new rapid reprogramming features beginning to appear on FPGAs.

Sinangil performed a useful analysis of the amount of memory necessary for an encoder to be fully efficient during motion estimation across various image and block sizes [56]. He also found that previous encoders had dedicated between 50% and 80% of their total area to their motion estimation accelerators, and that 99.9% of all ideal block matches lie within a search area of ± 64 pixels. He also put forward a scheme for managing the prefetch operations of pixels. When Sinangil went to develop a memory aware motion estimation algorithm, based on those results, he found that he could reduce off-chip memory bandwidth by 47x and on-chip memory area by 16% at the cost of 1.6% average bit rate increase [57].

Li and Zhang present domain-specific techniques to reduce DRAM energy consumption for image data access by up to 92%, and should be recalled if a DRAM based memory architecture is constructed to support the on-chip memory already present in a motion estimation accelerator [58].

4.3.1 Standard Cell Memories

Meinerzhagen published an exploration of standard cell memories in 65nm in 2010, demonstrating that these memories could be built with a 49.98% area penalty in trade for a 36.54% power reduction for the overall memory array [59]. Further investigation into how such memories stack up in the subthreshold domain, compared to SRAM macros, found that these SCMs were more reliable than standard SRAM macros, but less than full custom macros designed specifically for subthreshold operation [60]. This research, however, also surfaced the idea that these SCMs could be used in distributed memory blocks closely integrated with logic, and further, that these memories would work consistently with their accompanying logic, a promise that is not a surety with SRAMs. For a design which makes use of voltage dithering or other similar power control techniques, both features integrated into every tile in an AsAP array, these memories would be quite useful. Meinerzhagen then demonstrated a 4K-bit SCM built with an automated compilation flow and demonstrated its reliability at subthreshold voltages [61].

4.3.2 Reference Frame Compression

Another possibility for dealing with the large memory storage requirements is to compress the reference frame and then decompress it before SAD computation. This runs into two primary difficulties. As described by Budagavi, it requires one to pick encoding and decoding techniques that are not too memory or hardware intensive, as that would offset the gains from compressing the reference frame in the first place [62]. Additionally, the compression algorithm chosen, if lossy, results in degradation of the final video coding operation. Gupte attempted to balance the tradeoffs of lossless and lossy compression by making use of lossy compression when performing motion estimation, and lossless compression while executing motion compensation [63]. This combined method resulted in a 39% bandwidth savings, greater than the 25% found by Budagavi, since the bandwidth effect is mostly felt in the motion estimation step. Ma and Segall made use of a similar dual-compression type scheme, where they stored high resolution and low resolution versions of the reference frame, and then also created a residual Table between the high and low resolution images. They incorporated this scheme into the software version of the H.265 encoder and demonstrated an increased bitrate of 1% and a bandwidth savings of 20%. Silvereira then extended the techniques of Huffman encoding to compile a set of of code Tables to store the reference frame. These code Tables gave a bandwidth reduction of 24%

and no bitrate penalty [64]. The limitation of Silvereira's technique is the generation and storage of pre-compiled code Tables, but in situations where the video streams are broadly similar to each other, such as the storage of nightly newscasts, sports matches shot from the same angles, or other similarly static streams, the technique could be applied without facing the code-translation penalty. Wang and Richter looked at the total savings available from purely lossless implementations and showed that smart selection of the lossless encoding could reduce the bitrate by 9.6%, and reduce the necessary size of the memory buffer by up to 80% [65]. Table 4.1 consolidates the results of these works, though it unfortunately must gloss over some of the relative details.

Table 4.1: Bandwidth savings and costs from reference frame compression techniques

Work	BW Savings	PSNR (dB)	Bitrate Increase
[62]	25%	-0.043	1.03%
[63]	17% - 24%	-0.010	0.74%
[66]	20%	-0.006	0.38%
[64]	24%	0	0.00%
[65]	9.6%	0	0.00%

4.4 Accelerating Motion Estimation

Hardware accelerators have been developed for both H.264 and H.265 standards. Some accelerate the whole video coding kernel, and others only address a particular sub-section of the kernel. The motion estimation part of the video coding operation has an interesting design space. These hardware accelerators cover new instruction sets, GPU based designs, ASIC based designs, and ASIP designs. They make use of a number of novel techniques, balancing the tradeoff of final coding quality versus the time and energy required to get there.

4.4.1 Software Baseline Encoder

The standards committee publishes a draft encoder for use on general purpose computing platforms [9]. It is written in C++ and supports all modes of operation present in the full standard. It is not optimized for performance, but rather completeness, and so makes use of both a full-search pattern along with an exhaustive testing of each possible block size for encoding. It should find the most compact encoding possible. Encoding of 4K video streams takes on the order of tens of minutes per frame. It requires no specialized hardware and is portable to any system that can handle its memory requirements.

4.4.2 Dedicated SAD Instructions for CPUs, Embedded Compute Accelerators

Proposed SAD instructions have gone as far as to offer 16x1 and 16x16 block SAD compares, reducing the total cycles count for such operations significantly (32 single-cycle instructions as compared to 1, or 4 cycle instruction) while leaving the high level command and control to the CPU [1]. Other dedicated instructions have focused on the SAD operation at the circuit level, optimizing a function which takes eight pairs of pixels and produces their SAD as efficiently as possible across a wide range of supply corners [67].

4.4.3 GPU-Based Implementations

The expanded availability and programmability of GPGPU compute platforms has lead to the development of H.264 encoders which use the GPU as their primary compute platform. These algorithms makes use of a parallelized full-search ME algorithm constrained by search area and the many compute cores of the GPU to process the whole search space as quickly as possible. As shown by Rodriguez-Sanchez, the motion estimation process can be broken into three main phases: SAD computes, SAD summations, and cost comparison, and such a partitioning in CUDA can give a 70.5x performance increase over pure CPU implementations [48]. In the first phase, the GPU divides the target macroblock into 4x4 subblocks, and then computes the SAD between each of those subblocks and all possible subblocks inside the search area. This is computationally intensive, but makes good use of the many processing elements available inside of the GPU. After all the SADs have been computed, the GPU then recombines those SADs into the various possible block sizes. These block sizes are then ranked, and the smallest SAD candidate chosen. Both step two and three of the process can also take advantage of the GPUs high data parallelism. Zhang, Nezan, and Cousin leveraged OpenCL to more directly compare the differences between pure CPU, heterogeneous, and pure GPU implementations of a motion estimation kernel. Leveraging the use of shared memory, and vector data instructions, they use a technique similar to Rodriguez-Sanchez, they were able to show that an OpenCL kernel could outperform a C implementation in 720p on the same processor by 7.6x, by 38x when using only the GPU, and by 89x when using a combined CPU and GPU processing system [68]. Wang then took a more powerful GPU, a newer version of CUDA, and a more clever work-partitioning strategy for the motion estimation and was able to produce a heterogeneous CPU-GPU combined system which outperformed a pure CPU implementation by 112x [69]. Even though the speedup was impressive, it should be noted that that system was still only able to manage 23.77 FPS on a 2560x1600 video stream, which means that it cannot handle 4K video at full framerate.

These implementations demonstrate that GPU platforms can achieve good performance in terms of framerate, but the power requirements to run a GPU means that their performance suffers when the performance metric incorporates power per operation. Even with that considered, heterogeneous CPU combined with GPU implementations of H.264 encoders produce significantly more throughput than either pure CPU or GPU designs, and for most consumer desktop systems which already contain both CPU and discrete GPU combinations, it would make sense to use these techniques to speed up encoding without additional hardware.

Table 4.2: Motion estimation designs targeting GPU platforms

Work	Language	Platform	Format	Perf.	FPS	Block Sz	Pix/S
[48]	CUDA	GTX480	720p	70.5x	-	16x16	-
[68]	Open CL	I7 2.8 GHz	720p	12.6x	7.6	16x16	7004160
		GT540m	720p	63.3x	38.0	16x16	35020800
		I7 + GT580m	720p	73.3x	44.0	16x16	40550400
[69]	CUDA	Xeon + C2050	1080p	112.0x	77.7	VBSME	161118720

4.4.4 ASIC Designs

There are two general categories of ASIC encoders: configurable and fixed. Fixed encoders have set search patterns and are unable to vary block size. Configurable ASICs have support for varying both of those settings. Enabling configuration complicates the overall hardware, but allows for greater flexibility, future proofing, and implementation of additional features to save power or increase performance by sacrificing differing amount of bit-rate depending upon application.

4.4.4.1 Systolic Arrays

Systolic array implementations are motion estimation engines which make use of many parallel processing elements to generate the SADs for macroblocks as the image frame streams into the device. Lai and Chen introduced a 2D full-search block matching algorithm architecture which achieved 100% hardware utilization in a tile-able architecture [70]. This architecture used a total of 256 PEs to process a 16x16 macroblock within a search area of $[-8, +7]$ in both the X and Y directions, and was scalable to process the same macroblock across a search range of $[-16, +15]$ with 1024 PEs. Elgamel introduced an early termination mechanism in a systolic array which disabled PEs that were not producing a competitive matching candidate, as well as the accumulation adders on the edge of the array, which saved 45% power over a normal array, by reducing the total number of comparisons by 50% [71]. Both of the previous designs could only handle fixed block sizes after implementation. Huang introduced a 2D systolic array implementation that was less efficient, with the PE array being only at 97% utilization, but capable of variable block size computations, chosen at run time, suitable for processing 720x480 video at 30 FPS [72]. This design also made use of a rectangular search range, with a larger search area in the horizontal direction $[-24, +23]$ than the vertical direction $[-16, +15]$. Deng expanded the search area of Huang to $[-32, +31]$ in both directions and scaled it up to handle 720x576 video at 30 FPS, at the cost of roughly double the total number of gates [73].

Chen et al. give a good general analysis of the cost of supporting VBSME in systolic array style implementations, and proposes an architecture suitable for 720p 30

FPS processing [74]. Their design makes heavy use of pixel truncation, rounding to 5 MSB for each pixel. They round that distortion from the loss of 3 LSB was about 0.1 dB, while 4 LSB reduction cost 0.2 dB. Additionally they make use of a prediction unit to choose which area of the search range their implementation checks, massively reducing the total area which must be searched, though rapid changes in direction reduces the quality of their prediction algorithm.

Zhaoqing, Hongshi, Weifeng, and Xubang come to a similar conclusion as Chen et al. that the total computational complexity must be dramatically reduced in order to maintain throughput in larger video stream [75]. They implemented a systolic array that can process 720p video at 60 FPS, but in a very limited search range of $[-8, +7]$, allowing them to shrink the total size of their PE array, and instead add more SRAM to their design, rather than keeping all the pixel in flight inside the PE array. Unfortunately, they did not address the image quality cost of their decision to limit the total search area for each macroblock. Working significantly later than the other systolic array implementations, Byun, Jung, and Kim proposed an encoder suitable for UHDTV (3840x2160 at 30 FPS) using a traveling 64x64 search area and intermediate SAD value storage requiring 20KB of SRAM to store both the reference pixels and the intermediate SADs and supporting the full space of possible block sizes [76]. Table 4.3 summarizes the various systolic array architectures.

Table 4.3: Comparisons between various systolic array (full search) implementations

Work	Srch Area	Block Sz	PEs	Max Res	FPS	MPix/s	MEM	Process
[70]	16H, 16V	16x16	1024	-	-	-	-	-
[71]	15H, 15V	16x16	-	-	-	-	-	-
[72]	24H, 16V	16x16	41	720x480	30	10.36	3.0 KB	0.35 µm
[73]	65H, 65V	4x4-16x16	256	720x576	30	12.44	7.9 KB	0.18 µm
[74]	64H, 32V	4x4-16x16	2048	720p	30	27.64	26.0 KB	0.18 µm
[75]	16H, 16V	4x4-16x16	256	720p	60	55.29	41.6 KB	0.18 µm
[76]	64H, 64V	8x4-64x64	256	2160p	30	248.8	20.0 KB	65 nm

Akin, Ulusel, Ozcan, Sayilar, and Hamzaoglu experimented with predictive SAD calculations applied to systolic arrays [77]. The reasoning, is that since distortion between pixels tends to by spatially correlated, a prediction can be made about the SAD, and whether

$a - b$ or $b - a$ is the positive value. Using a simple one-step predictor, the previous path taken, they achieve 90.1% accuracy on their prediction. Leveraging some mis-prediction mitigation techniques allow them to show a system that loses no PSNR for 2.2% dynamic power savings, or sacrifices up to 0.04% PSNR for a 9.3% savings in dynamic power. In power-tight applications, their techniques could be the difference in meeting an aggressive power budget.

If making use of an FPGA platform to implement a systolic array, Niitsuma and Maruyama have put together an evaluation of different SAD circuits with overlapping search windows in FPGAs from the Vertex family [78]. If making use of an FPGA platform to implement a systolic array, Niitsuma and Maruyama have put together an evaluation of different SAD circuits with overlapping search windows in FPGAs from the Vertex family [78].

4.4.4.2 Non-2D ASICs and ASIPs

There are other motion estimation engines which use different architectures from 2D systolic arrays. These designs make use of search patterns, picking fewer points to sample using a strategy to trade PSNR loss for faster processing and significantly fewer points checked overall. Chun, Kun, Songping, and Zhihua modified a programmable DSP processor architecture to fetch and perform a subtract, absolute, add operation on 8 pixels at a time in the same cycle it fetches the next 8 pixels, resulting in a 20x speedup over a SISD architecture [79]. Since they were extending a programmable processor, their implementation could be extended to cover a wide range of search patterns, though they used it primarily with three step searches (TSS, typically Diamond-Diamond-Cross). Fatemi, Ates, and Salleh experimented with using pixel truncation alongside bit-serial pipeline architecture to improve throughput further, while paying a similar cost to PSNR [80]. Their implementation looks similar to a 2d systolic array implementation, but its use of a bit-serial architecture instead of a bit-parallel one, distinguishes it.

Vanne, Aho, Kuusilinna, and Hamalainen developed their own motion estimation implementation with run time configuration of search patterns, and block access memory architectures [81]. This design can process 1080p video at 30 FPS while consuming 123mW, and they demonstrated its robustness across five different search patterns. They also discussed, in detail, the math necessary to have separable memory addresses such that the pixel memory can be written in lines, but accessed in blocks.

Xiao, Le, and Baas demonstrated a fully-featured H.264 compatible encoder on a 167-core asynchronous array of simple processors (AsAP) platform [82]. The design used a dedicated motion estimation accelerator [83], along with 115 of the simple cores to implement a design suitable for 640x480, 21 FPS video encoding for 931 mW average power consumption. The design could also be scaled to the workload by managing the power supplies, from 95 inter FPS at $0.8V$ to 478 inter FPS at $1.3V$ in QCIF frames. Another way of thinking of this is that the design could operate anywhere from 20% to 100% of its maximum throughput capacity, all by controlling the core voltage levels.

Kim and Sunwoo introduced a application specific processor that they called MESIP which was capable of 720p, 50 FPS processing for 22.22 mW and a total of 8 KB of SRAM. The MESIP required the development of its own software tools, but can leverage those tools to optimize data-reuse strategies. The execution unit of the MESIP resembles the 2d systolic arrays, but the memory management and search pattern functionality provided by its control unit removes it from the 2d systolic array class.

Table 4.4: ASICs and ASIPs targeting motion estimation

Work	Type	Alg.	Block Size	Format	FPS	Avg. Power	Process
[79]	DSP	TSS	16x16	CIF	24	-	-
[80]	ASIC	FS	4x4-16x16	CIF	41	-	0.18 µm
[81]	ASIC	Prog.	4x4-16x16	1080p	30	-	0.13 µm
[83]	AsAP	FS	4x4-16x16	CIF	210	-	65 nm
[82]	AsAP	Prog.	4x4-16x16	480p	21	931 mW	65 nm
[84]	ASIP	Prog.	4x4-16x16	1080p	30	22.23 mW	90 nm
This	AsAP	Prog.	4x8-64x64	720p	114.4	-	65 nm

For developing ASICs and ASIPs, Yang, Wolf, and Vijaykrishnan have put together a helpful primer on how to predict power and performance for motion estimation engines based on memory transfers, SAD computes, and other motion-estimation specific criteria [85]. The most important understanding to come away with from their analysis is that the total number of points checked is not a sufficient proxy for how much energy the architecture or software expends. This also implies that proposed search patterns cannot justify themselves based simply on the total number of search points examined.

With regards to the datapath of a motion estimation computation, Vanne, Aho, Hamalainen, and Kuusilinna have produced good analysis into the building of fast, efficient SAD compression trees, and best SAD decision trees [86]. These are good areas of enhancement for designs which find themselves compute-bound. Additionally, Kaul, Anders, Mathew, Hsu, Agarwal, Krishnamurthy, and Borkar have done an in-depth design and analysis of SAD compute units and have produced a number of interesting and valuable circuit level enhancements [67].

4.5 Comparative Performance

Table 4.5 gives a breakdown of the relative throughput and pixel efficiency per device. Pixel efficiency is not measured by the total number of pixels processed, but by the total number of pixels handled, so if a methodology can handle whole frame of pixels with only a few search points, it gains the value of the whole pixel frame. Power numbers for the GPU and CPU devices are based on the published manufacturer TDP for that device brand, the heterogeneous CPU/GPU, since the design aims at full utilization of both components, pays the full power price of both devices. Power numbers for this work are projected from the numbers measured by previous AsAP style encoders. Unfortunately the systolic array architectures do not report power numbers, but instead the total number of gates used in their implementations, so its not possible to develop a good estimation of their power compared to the other types of devices. They are included in the Table to give a sense of what sorts of throughput are available with those architectures, and so that if future designs do measure the power number, a full comparison can be properly back-related.

Table 4.5: Throughput and efficiency comparison across the solution space

Work	Type	OpFreq (MHz)	Throughput (MPix/s)	Power (mW)	Efficiency KPix/Joule
[69]	CPU/GPU	3100/575	161.12	318,000	507
[82]	AsAP	400/Var	6.45	931	6,929
This	AsAP	500/Var	22.84	1375	16,613
This	AsAP	1000/Var	45.69	2750	16,613
[81]	ASIC	200	62.21	123	505,756
[84]	ASIP	229	46.08	22	2,072,874
[74]	2D-SA	108	27.65	-	-
[75]	2D-SA	55.6	55.30	-	-
[76]	2D-SA	250	248.83	-	-

Chapter 5

ME2 Architecture

The accelerator can be conceptualized as a specialized micro-controller. It has its own instruction set, communicates with other blocks through input and output FIFOs and has its own clock and sleep signals. This encapsulation makes it easy to integrate as many accelerators as wanted by the designers of any particular AsAP generation.

A top level block diagram of the entire accelerator is sketched out in Figure 5.1. It's assumed that the input and output FIFOs lead to different AsAP tiles, but this is not architecturally necessary, and it is possible for the same block to act as both transmitter and receiver to MEACC2. This is made possible by the transmit and receive commands both being part of the same instruction set, specifically, not having overlapping op-codes.

The pixel datapath components are where the SAD computation occurs and are scalable to differing numbers of pixel computes per cycle. The implemented version of MEACC2 uses a pixel datapath that executes a 4x4 block compare. The datapath is pipelined. Additional details about the pixel datapath are located in Section 5.2.

5.1 Instruction Set

Instructions to MEACC2 are 16b wide and contain a 5b op-code. The op-code space is shared between input and outputs for easier parsing by the AsAP tiles that communicate with the device. Additionally, pixel transfer mode uses all 16b on the instruction to transfer pixels (8b at a time), and so the pixel move operations are blocking and can-

Table 5.1: The 32 instructions of the MEACC2 instruction set

Opcode	**In/Out**	**Instruction Name**
0	In	Write_Burst_ACT
1	In	Set_Burst_REF_X
2	In	Set_Burst_REF_Y
3	In	Write_Burst_REF
4	In	Set_Burst_Width
5	In	Set_Burst_Height
6	In	Set_Write_Pattern_Addr
7	In	Write_Pattern_DX
8	In	Write_Pattern_DY
9	In	Write_Pattern_JMP
10	In	Write_Pattern_VLD_Top
11	In	Write_Pattern_VLD_Bot
12	In	Set_PMV_DX
13	In	Set_PMV_DY
14	In	Set_BLKID
15	In	Set_Thresh_Top
16	In	Set_Thresh_Bot
17	In	Set_ACT_PT_X
18	In	Set_ACT_PT_Y
19	In	Set_REF_PT_X
20	In	Set_REF_PT_Y
21	In	Set_Output_Register
22	In	Start_Search
23	In	Send Pixels to Unit
24	Out	Result_Read
25	Out	Register_Read
26	Out	Pixel_Request
27	Out	Send Pixels to AsAP
28	In	Read REF MEM
29	In	Read ACT MEM
30	In	Read Register
31	In/Out	Issue Ping

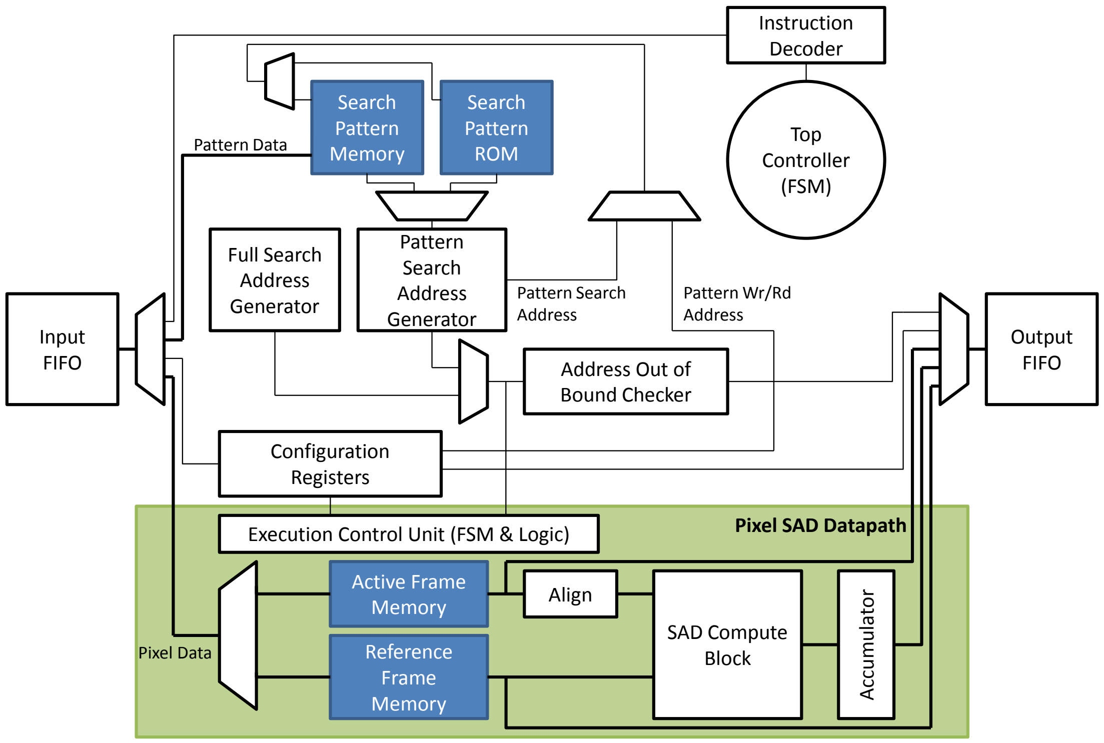


Figure 5.1: Top level block diagram

not be interrupted. There exists a ping instruction which can be used to flush through a pixel mode by repeated use until the responding ping from MEACC2 is transmitted onto the Output FIFO. Further details on this particular debug technique are located with the description of the Ping instruction in Section 5.1.5.1.

5.1.1 Register Input Instructions

These instructions write their operand value to the named register. These operations take a single cycle, and return the MEACC2 state machine to IDLE after resolution. These can be queued one after another in the input FIFO.

5.1.1.1 Set Burst REF X

The Burst REF X register is used only when writing a block of pixels to the reference memory outside of an active search. It denotes the X value of the top left corner of the block of pixels to move into memory. Its structure is given in Table 5.2.

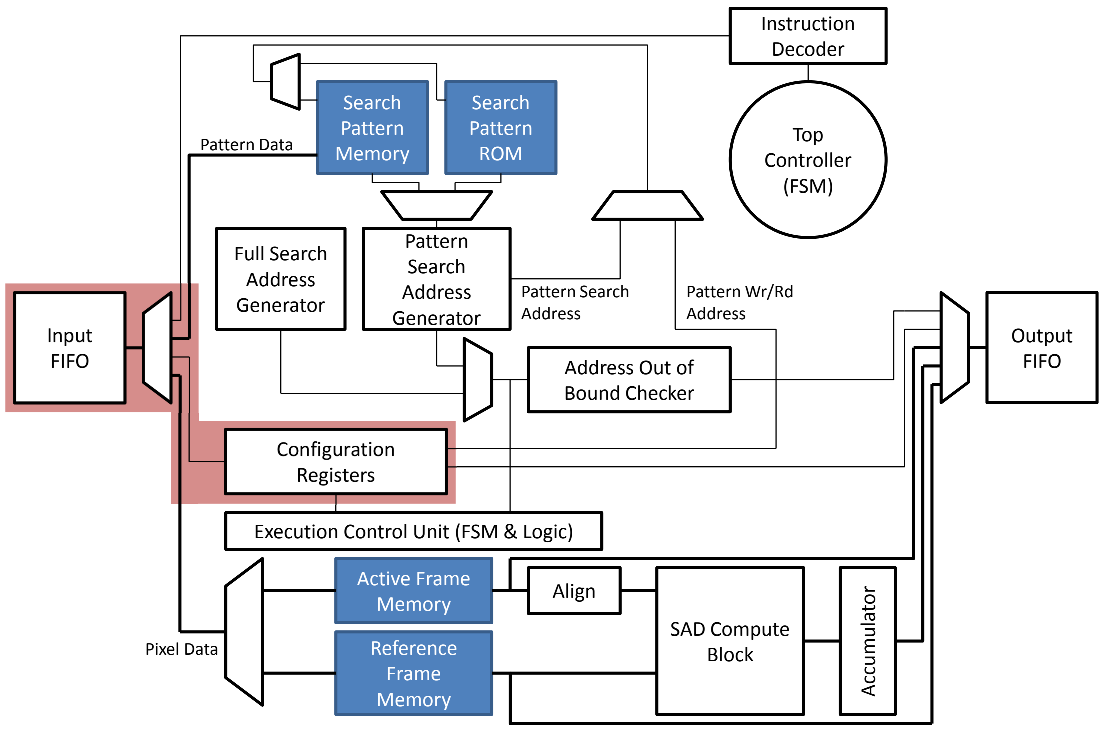


Figure 5.2: Top level register input path

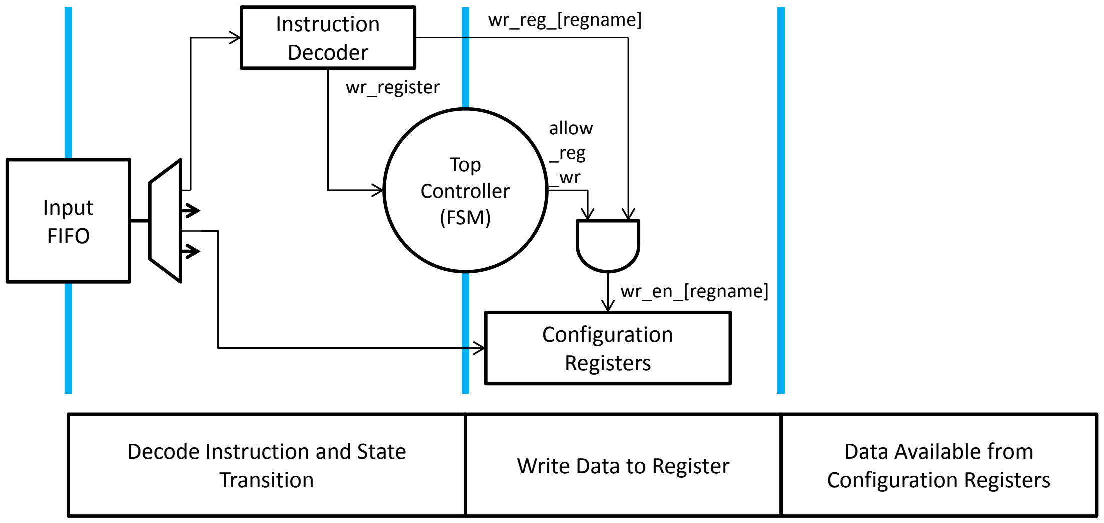


Figure 5.3: Pipeline diagram for register input instructions

Table 5.2: Set burst REF X structure

	Op-code	Unused	Burst REF X
Width	5b	3b	8b
Valid Values	00001	000	[0,255]

5.1.1.2 Set Burst REF Y

The Burst REF Y register is used only when writing a block of pixels to the reference memory outside of an active search. It denotes the Y value of the top left corner of the block of pixels to move into memory. Its structure is given in Table 5.3.

Table 5.3: Set burst REF Y structure

	Op-code	Unused	Burst REF Y
Width	5b	3b	8b
Valid Values	00010	000	[0,255]

5.1.1.3 Set Burst Height

The Burst Height register is used only when writing a block of pixels to the reference memory outside of an active search. It denotes the height (number of horizontal lines) of the block of pixels to move into memory. Its structure is given in Table 5.4. The maximum value is 64, and values above that have an undefined effect (in practice, this probably causes MEACC2to get stuck waiting for pixels. If resetting is not an option, the external controller should push ping instructions into the device until it begins to respond).

Table 5.4: Set burst height structure

	Op-code	Unused	Burst Height
Width	5b	4b	7b
Valid Values	00101	0000	[0,64]

5.1.1.4 Set Burst Width

The Burst Width register is used only when writing a block of pixels to the reference memory outside of an active search. It denotes the width (number of vertical lines) of the block of pixels to move into memory. Its structure is given in Table 5.5. The maximum value is 64, and values above that have an undefined effect (in practice, this probably causes MEACC2to get stuck waiting for pixels. If resetting is not an option, the external controller should push ping instructions into the device until it begins to respond).

Table 5.5: Set burst width structure

	Op-code	Unused	Burst Width
Width	5b	4b	7b
Valid Values	00100	0000	[0,64]

5.1.1.5 Set Write Pattern Address

The Write Pattern Address register is used only when writing to pattern memory. The instructions which actually write to pattern memory are separate from address selection, they are located in Section 5.1.3. This helps us handle the fact that pattern memory is very wide, but broken up into multiple operands. So data is written by operand, into the same address space. The pattern memory is actually split between a ROM and a RAM, and the RAM is located in the bottom of the memory address space, so even though the address space spans [0, 63], this command only take values from [0, 31]. Its structure is given in Table 5.6.

Table 5.6: Set write pattern address structure

	Op-code	Unused	Address
Width	5b	5b	6b
Valid Values	00110	00000	[0,31]

5.1.1.6 Set PMV DX

The PMV DX register is used along with the PMV DY register to fully set a predicted motion vector. This motion vector is used to offset the starting search point from the default center during a pattern search. It has no effect during a full search. It is not changed except by the user or by reset. The structure for this instruction is given in Table 5.7. It is a signed value.

Table 5.7: Set PMV DX structure

	Op-code	Unused	Offset
Width	5b	2b	9b
Valid Values	01100	00	[-255, 255]

5.1.1.7 Set PMV DY

The PMV DY register is used along with the PMV DX register to fully set a predicted motion vector. This motion vector is used to offset the starting search point from the default center during a pattern search. It has no effect during a full search. It is not changed except by the user or by reset. The structure for this instruction is given in Table 5.8. It is a signed value.

Table 5.8: Set PMV DX structure

	Op-code	Unused	Offset
Width	5b	2b	9b
Valid Values	01101	00	[-255, 255]

5.1.1.8 Set BLKID

The Block ID register defines the block size of any search the device executes. This also impacts the memory replacement scheme, but won't trigger memory replacement until a search is started. The block size mappings are given in Table 5.9. The instruction structure is given in Table 5.10. Register values [12,15] are undefined, but in implementation those options are tied to a block size of width and height 4. That sizing is not supported by the H.265 standard, but is the actual size of a single block compute. It hasn't been verified in simulation, so do not use the 4x4 block size without further investigation.

5.1.1.9 Set Thresh Top

The Threshold Top register contains the top 10 bits of the threshold value. The threshold value is a 20 bit value giving the minimum threshold for a successful search. During a search, if this value is non-zero, the search terminates if a SAD value is found less than the threshold value (strictly less, not less than or equal to). If this register and the

Table 5.9: Block ID mappings

Value	X Size (Width)	Y Size (Height)
0	64	64
1	32	64
2	64	32
3	32	32
4	16	32
5	32	16
6	16	16
7	8	16
8	16	8
9	8	8
10	4	8
11	8	4

Table 5.10: Set BLKID structure

	Op-code	Unused	Value
Width	5b	7b	4b
Valid Values	01110	0000000	[0,12]

bottom register are also zero, the search continues until the search pattern terminates as defined by the pattern. The structure for this instruction is given in Table 5.11.

Table 5.11: Set thresh top structure

	Op-code	Unused	Value
Width	5b	1b	10b
Valid Values	01111	0	XXXXXXXXXX

5.1.1.10 Set Thresh Bot

The Threshold Bottom register contains the bottom 10 bits of the threshold value. The threshold value is a 20 bit value giving the minimum threshold for a successful search. During a search, if this value is non-zero, the search terminates if a SAD value is found less than the threshold value (strictly less, not less than or equal to). If this register and the top register are zero, the search continues until the search pattern terminates as defined by the pattern. The structure for this instruction is given in Table 5.12.

Table 5.12: Set thresh bot structure

	Op-code	Unused	Value
Width	5b	1b	10b
Valid Values	10000	0	XXXXXXXXXX

5.1.1.11 Set ACT PT X

The Active Frame Point X register holds the X value of the top left corner of the block in ACT Memory which is used in a search. This is the block of pixels which is compared against every candidate block of pixels. The structure of this instruction is given in Table 5.13.

Table 5.13: Set ACT PT X structure

	Op-code	Unused	Value
Width	5b	3b	8b
Valid Values	10001	000	[0,255]

5.1.1.12 Set ACT PT Y

The Active Frame Point Y register holds the Y value of the top left corner of the block in ACT Memory which is used in a search. This is the block of pixels which is compared against every candidate block of pixels. The structure of this instruction is given in Table 5.14.

Table 5.14: Set ACT PT Y structure

	Op-code	Unused	Value
Width	5b	3b	8b
Valid Values	10010	000	[0,255]

5.1.1.13 Set REF PT X

The Reference Frame Point X register holds the X value of the top left corner of the block in REF Memory which is used in the initial compare. This is the first block of pixels which is compared against the block of pixels from ACT Memory. The structure of this instruction is given in Table 5.15.

Table 5.15: Set REF PT X structure

	Op-code	Unused	Value
Width	5b	3b	8b
Valid Values	10011	000	[0,255]

5.1.1.14 Set REF PT Y

The Reference Frame Point Y register holds the Y value of the top left corner of the block in REF Memory which is used in the initial compare. This is the first block of pixels which are compared against the block of pixels from ACT Memory. The structure of this instruction is given in Table 5.16.

Table 5.16: Set REF PT Y structure

	Op-code	Unused	Value
Width	5b	3b	8b
Valid Values	10100	000	[0,255]

5.1.2 Pixel Input Instructions

This is the format for encoding a pair of pixels to be transferred into MEACC2. Pixel transfer operations can be initiated in one of three ways:

- By sending the command Write Burst REF
- By sending the command Write Burst ACT
- MEACC2 can request pixel transfers; these requests appear on the output FIFO as command Pixel Request and the associated memory management components are expected to handle the request.

5.1.2.1 Send Pixels to Unit

Pixels are transferred into MEACC2 in pairs, taking up the entirety of the 16b word available, with a structure as given in Table 5.17. When looking at pixel order, Pixel 0 is the leftmost pixel in the pixel pair being transferred. Pixels are always transferred

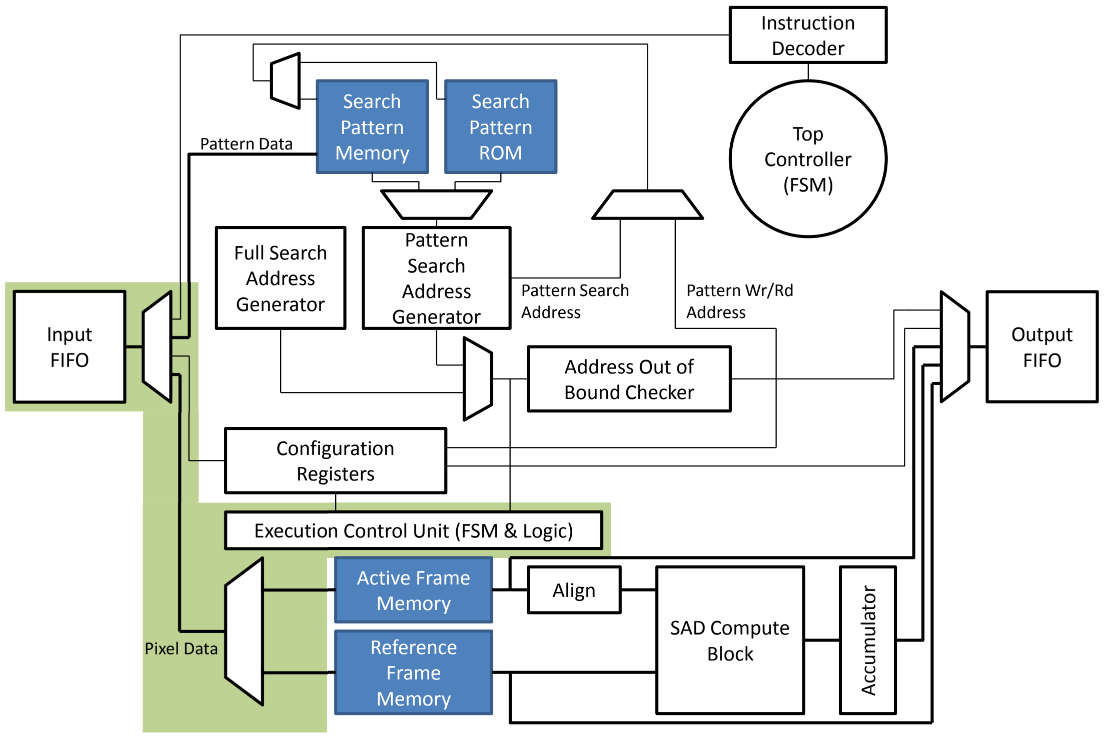


Figure 5.4: Top level pixel input path

in pairs, and pixel pairs always come from the same row of pixels (they have the same Y coordinate).

Table 5.17: Send pixels structure

	Pixel 0	Pixel 1
Width	8b	8b
Valid Values	[0,255]	[0,255]

5.1.3 Pattern Memory Input Instructions

The pattern memory, while used as a monolith by the pattern search execution engine, is capable of being written to by location within a particular memory word. These instructions use their operands to load the Pattern Memory by parts. They make use of the pattern memory address contained in the pattern memory addr register, which can be modified using the Set Pattern Memory Address command given in Section 5.1.1.5.

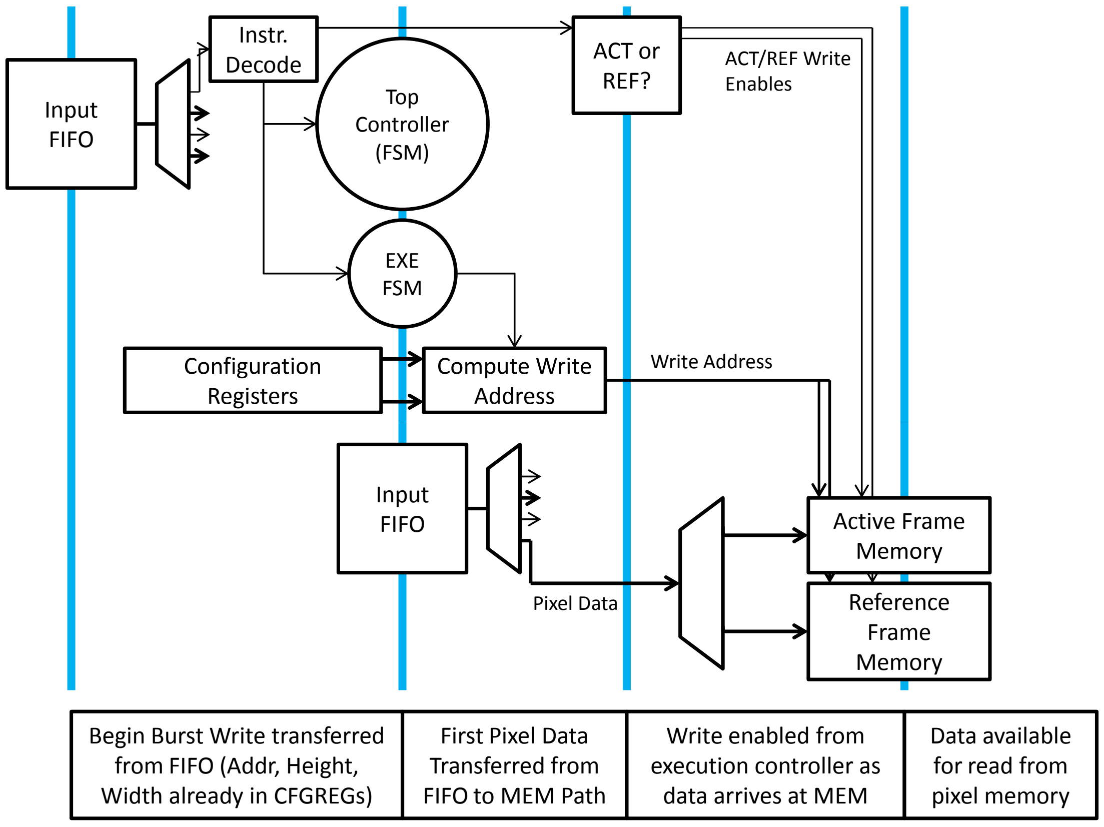

Input FIFO
Instr. Decode
Top Controller (FSM)
ACT or REF?
ACT/REF Write Enables
EXE FSM
Configuration Registers
Compute Write Address
Write Address
Input FIFO
Pixel Data
Active Frame Memory
Reference Frame Memory
Begin Burst Write transferred from FIFO (Addr, Height, Width already in CFGREGs)
First Pixel Data Transferred from FIFO to MEM Path
Write enabled from execution controller as data arrives at MEM
Data available for read from pixel memory

Figure 5.5: Pipeline diagram for pixel input instructions

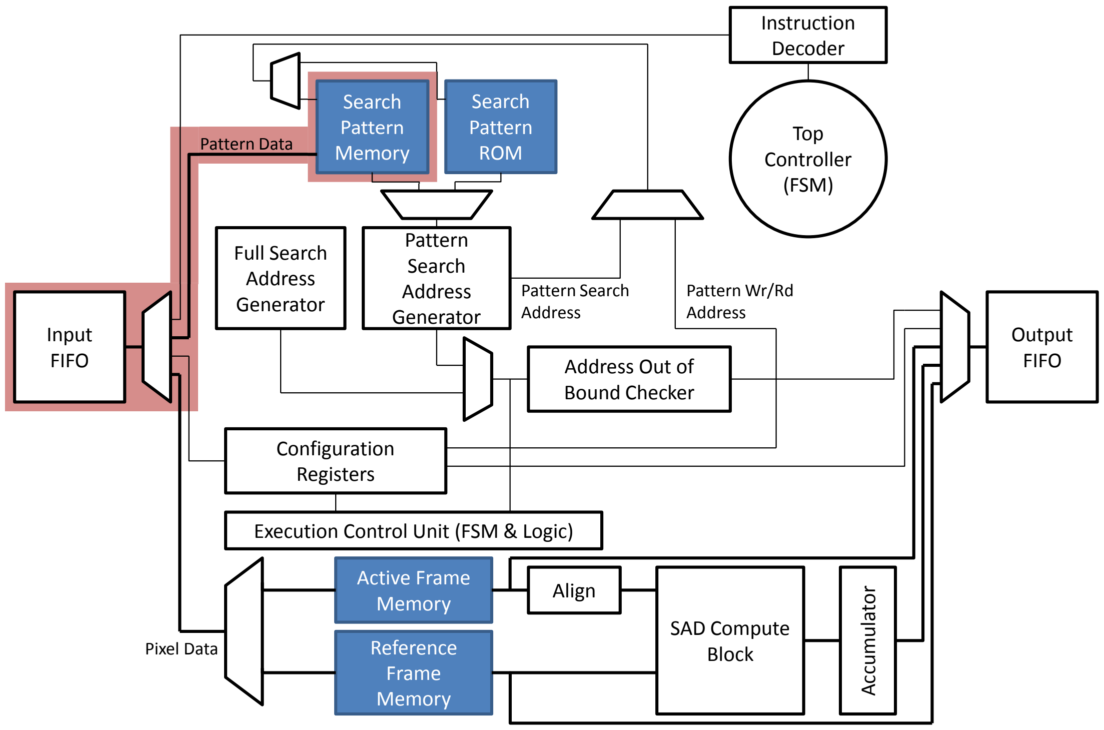


Figure 5.6: Top level pattern memory input path

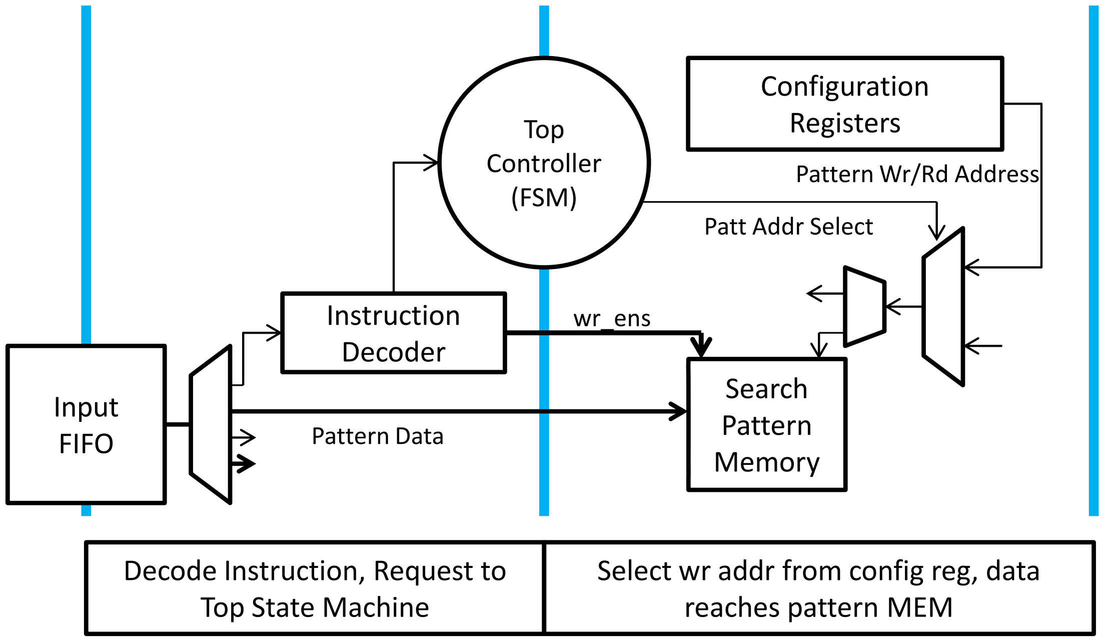


Figure 5.7: Pipeline diagram for pattern memory input instructions

5.1.3.1 Write Pattern DX

This command writes a new value to the part of pattern memory responsible for maintaining the X offset from center for the point addressed by the write pattern memory address register. The structure of this instruction is given in Table 5.18.

Table 5.18: Write pattern DX structure

	Op-code	Unused	Value
Width	5b	2b	9b
Valid Values	00111	000	[-255,255]

5.1.3.2 Write Pattern DY

This command writes a new value to the part of pattern memory responsible for maintaining the Y offset from center for the point addressed by the write pattern memory address register. The structure of this instruction is given in Table 5.19.

Table 5.19: Write pattern DY structure

	Op-code	Unused	Value
Width	5b	2b	9b
Valid Values	01000	000	[-255,255]

5.1.3.3 Write Pattern JMP

This command writes a new value to the part of pattern memory responsible for maintaining the jump address for the point addressed by the write pattern memory address register. The jump address is the point in the pattern memory jumped to when the point is picked as the next best SAD during pattern execution. The structure of this instruction is given in Table 5.20.

Table 5.20: Write pattern JMP structure

	Op-code	Unused	Value
Width	5b	5b	6b
Valid Values	01001	00000	[0,63]

5.1.3.4 Write Pattern VLD Top

This command writes a new value to the part of pattern memory responsible for the top valid bits for the point addressed by the write pattern memory address register. The valid bits are used to prevent repeated visiting of points during a search. The structure of this instruction is given in Table 5.21.

Table 5.21: Write pattern VLD top structure

	Op-code	Unused	Value
Width	5b	3b	8b
Valid Values	01010	00000	XXXXXXXX

5.1.3.5 Write Pattern VLD Bot

This command writes a new value to the part of pattern memory responsible for the bottom valid bits for the point addressed by the write pattern memory address register. The valid bits are used to prevent repeated visiting of points during a search. The structure of this instruction is given in Table 5.22.

Table 5.22: Write pattern VLD bot structure

	Op-code	Unused	Value
Width	5b	3b	8b
Valid Values	01011	00000	XXXXXXXX

5.1.4 Output Instructions

These instructions are used to read out information from registers or memories inside MEACC2. They are intended mostly for debugging purposes.

5.1.4.1 Set Output Register

The output (control) register was initially envisioned as a way to modified what information was read out for a search result. As implemented, it acts as a scratch register which can be written to and read out, but doesn't have any purpose beyond that. The structure of the command to write the output register is given in Table 5.23.

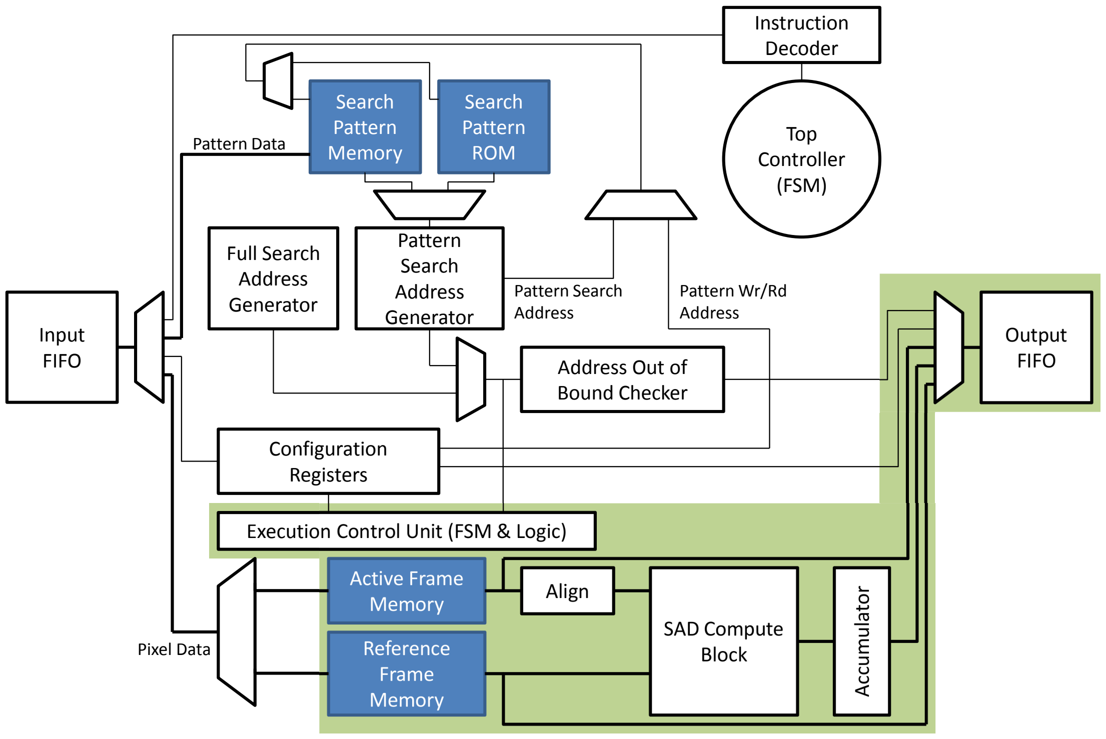


Figure 5.8: Top level output path

Table 5.23: Set output register structure

	Op-code	Unused	Value
Width	5b	5b	6b
Valid Values	10101	00000	XXXXXX

5.1.4.2 Read REF MEM

This instruction causes MEACC2to output the 16 pixels in the 4x4 block of REF memory addressed by this command. The structure of the command is given in Table 5.24. These pixels are not necessarily in raster order, but may be rotated based on the memory address used to get the pixels. This rotation effect is more thoroughly explained in Section 5.3.2. This command cannot access all pixels, as there is insufficient space in a single word to get the full 16 bit address across. Instead, 5 bits each of X and Y address are used, with the bottom 3 bits filled with 0s. Therefore, an address of (4,4) is converted into (32,32) within the device.

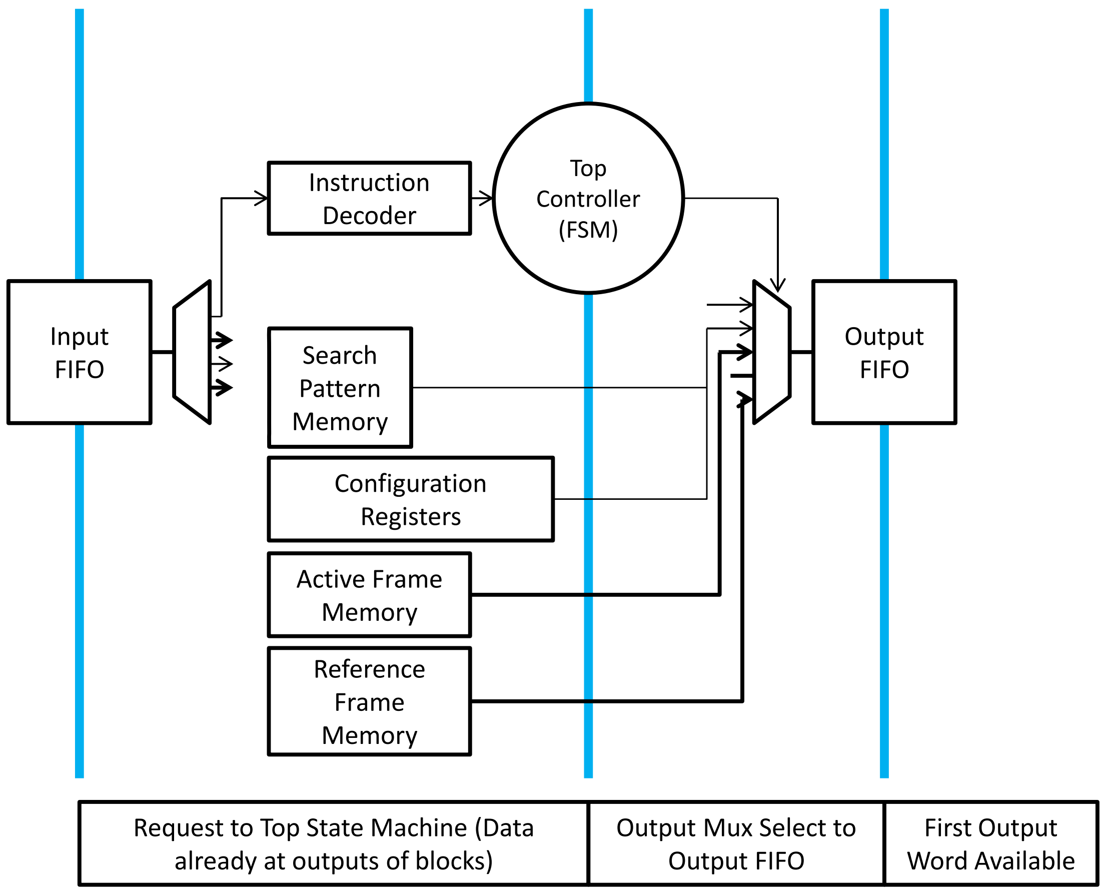


Figure 5.9: Pipeline diagram for output instructions

Table 5.24: Read REF MEM structure

	Op-code	Unused	X Addr	Y Addr
Width	5b	1b	5b	5b
Valid Values	11100	0	[0,31]	[0,31]

5.1.4.3 Read ACT MEM

This instruction causes MEACC2to output the 16 pixels in the 4x4 block of act memory addressed by this command. The structure of the command is given in Table 5.25. These pixels are not necessarily in raster order, but may be rotated based on the memory address used to get the pixels. This rotation effect is more thoroughly explained in Section 5.3.2. This command cannot access all pixels, as there is insufficient space in a single word to get the full 16 bit address across. Instead, 5 bits each of X and Y address are used, with the bottom 3 bits filled with 0s. Therefore, an address of (4,4) is converted into (32,32) within the device.

Table 5.25: Read ACT MEM structure

	Op-code	Unused	X Addr	Y Addr
Width	5b	1b	5b	5b
Valid Values	11101	0	[0,31]	[0,31]

5.1.4.4 Read Register

This command causes the chosen register's value to be read out onto the output FIFO. The registers that are readable with the read register command are given in Table 5.26, and the structure of the command is given in Table 5.27. The output word produced by this instruction takes the form specified in Section 5.1.4.5.

5.1.4.5 Register Read

Register Read is an instruction that only ever appears on the output FIFO. It contains the value from the register requested by the Read Register command placed on the input FIFO. The structure is given in Table 5.28.

Table 5.26: Read register operand lookup table

Register ID	Register
0	Burst REF X
1	Burst REF Y
2	Burst Height
3	Burst Width
4	Pattern Write Address
5	PMV DX
6	PMV DY
7	Block ID
8	Threshold Top Bits
9	Threshold Bottom Bits
10	ACT PT X
11	ACT PT Y
12	REF PT X
13	REF PT Y
14	OUT Register
15	Image SZ X
16	Image SZ Y
17	Pattern Data X Offset (bits [39:31])
18	Pattern Data Y Offset (bits [30:22])
19	Pattern Data Jump Address (bits [21:16])
20	Pattern Data Top Valid (bits [15:8])
21	Pattern Data Bottom Valid (bits [8:0])
[22:31]	Undefined

Table 5.27: Read register structure

	Op-code	Unused	Value
Width	5b	1b	8b
Valid Values	11110	0	[0,255]

Table 5.28: Register read structure

	Op-code	Unused	Value
Width	5b	1b - 5b	10b - 6b
Valid Values	10100	0	XXXXXXXXXX
Valid Values	10111	00000	XXXXXX

5.1.4.6 Result Read

Result Read is only ever present on the output FIFO. It indicates that the information being conveyed is the result of a search. The final result comes out over the course of 4 words, with each word containing the instruction op-code and part of the answer. The structure and order of the command is given in Table 5.29.

Table 5.29: Result read structure

Word 0	Names	Op-code	Unused	SAD Top
	Width	5b	1b	10b
	Valid Values	11000	0	XXXXXXXXXX
Word 1	Names	Op-code	Unused	SAD Bottom
	Width	5b	1b	10b
	Valid Values	11000	0	XXXXXXXXXX
Word 2	Names	Op-code	Unused	X
	Width	5b	3b	8b
	Valid Values	11000	000	[0,255]
Word 3	Names	Op-code	Unused	Y
	Width	5b	3b	8b
	Valid Values	11000	000	[0,255]

5.1.4.7 Pixel Request

Pixel request is only ever present on the output FIFO. It indicates that the MEACC2 requires additional pixels to be put on its input FIFO to complete its current search operation. The request takes a total of 4 words, with each word containing the instruction op-code and part of the request. The structure and order of the command is given in Table 5.30. In the case where pixel requests require both a horizontal and vertical shift, two requests are issued, for a total of eight words.

5.1.5 Operation Instructions

These instructions are sent to begin operations, using data from registers that have already been configured. Some of the instructions have operands as well.

5.1.5.1 Issue Ping

The Issue Ping command causes the device to echo the ping on its Output FIFO. Consequently it can be used to clear inappropriate requests for pixels, or used to ensure the

Table 5.30: Pixel request structure

Word 0	Names	Op-code	Unused	X
	Width	5b	3b	8b
	Valid Values	11010	000	[0,255]
Word 1	Names	Op-code	Unused	Y
	Width	5b	3b	8b
	Valid Values	11010	000	[0,255]
Word 2	Names	Op-code	Unused	W
	Width	5b	4b	7b
	Valid Values	11010	000	[1,64]
Word 3	Names	Op-code	Unused	H
	Width	5b	4b	7b
	Valid Values	11010	000	[1,64]

silicon is alive before attempting to use it for more complex things. The structure is given in Table 5.31.

Table 5.31: Issue ping structure

	Op-code	Value
Width	5b	11b
Valid Values	11111	00000000001

5.1.5.2 Write Burst ACT

The Write Burst ACT command puts the device into a mode, ready to accept a number of pixel pairs sufficient to fill the whole ACT memory (64x64 pixels of 1 byte, 2048 pixel pairs) in raster-scan order starting from the top left pixel pair [(1,0),(0,0)]. This mode is the only way to move memory into ACT memory. It is okay to use such a constrained method because ACT frame pixels change infrequently, only once per search, and if the block size is not the largest supported (64x64) less than once per search overall. The structure is given in Table 5.32.

5.1.5.3 Write Burst REF

The Write Burst REF command puts the device into a mode, ready to accept a number of pixel pairs sufficient to fill the REF Memory in raster-scan order starting from

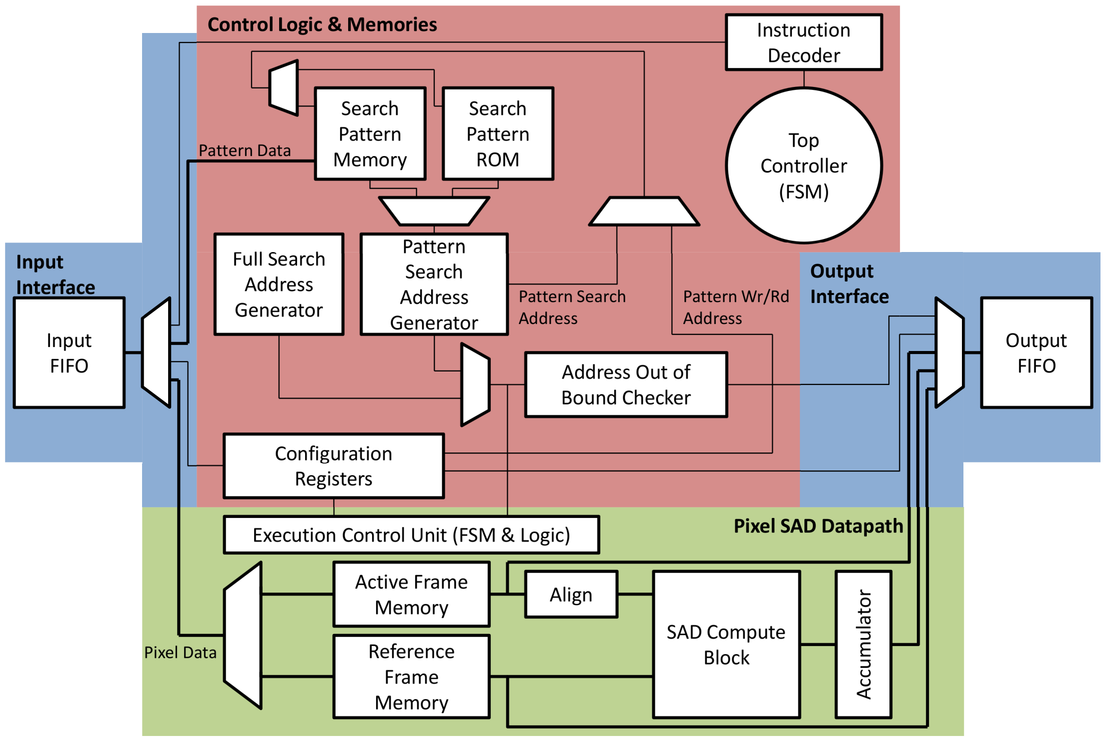


Figure 5.10: Top level block diagram annotated by function

Table 5.32: Write burst ACT structure

	Op-code	Value
Width	5b	11b
Valid Values	00000	00000000000

the top left pixel defined in registers Burst REF X and Burst REF Y, and for the width and height given by the Burst Width and Burst Height registers. This is the only user-initiated way to move pixels into REF Memory. Pixels are also moved into REF Memory when the unit requests pixels. The structure is given in Table 5.33. The unit also maintains an internal set of registers which track where the top left hand point of the REF memory is located in the overall image. These registers are visible to the user through the Read Register command.

Table 5.33: Write burst REF structure

	Op-code	Value
Width	5b	11b
Valid Values	00011	00000000000

5.1.5.4 Start Search

The start search command executes either a full search or a pattern search based on its form. Full Searches also take their decimation arguments here, while pattern searches take their starting pattern address. A full search can be decimated up to 32x in either (or both) dimensions. The search progresses, generating pixel requests if necessary, until it terminates. Once the search terminates it pushes a read result command onto its output FIFO. All searches terminate, eventually. There should only be one set of four words of read result for every one start search command placed on the input FIFO. The structures of the two types of search commands are given in Table 5.34.

Table 5.34: Start search structure

Full Search	Names	Op-code	X Decimation	Y Decimation	PS
	Width	5b	5b	5b	1b
	Valid Values	10110	[0,31]	[0,31]	0
Pattern Search	Names	Op-code	Unused	Pattern Address	PS
	Width	5b	4b	6b	1b
	Valid Values	10110	0000	XXXXXX	1

5.1.6 Limitations

The primary design constraint which the MEACC2 must compensate is the relatively limited bandwidth of the 16b AsAP word. The given width means that 2 pixels can be moved into the unit per machine cycle. Further research has been done into pixel truncation, as an attempt to save memory bandwidth and storage area but the quality of the final search results suffers too much beyond one of two bits of truncation [87]. Future designs could accept the reduction in quality of a 5b pixel to fit 3 pixels/word, or 4b pixels to fit 4 pixels/word.

5.1.7 Example Programs and Latency

A basic use of MEACC2 execution contains 3 major phases: Load, Configuration, and Execution. The execution loop requests additional pixels (if necessary) from the adjacent AsAP tile until the search is resolved. Load is done before register configuration because after the initial pixel load, a user can execute multiple searches by changing configuration registers and re-executing. The unit handles the requesting and reloading the pixels. This is the recommended flow, since MEACC2 can sense whether or not to bring in more pixels to complete the search. To use the MEACC2 without generating any pixel requests, the search range can be constrained so that the pixels which are out of bounds are not considered valid locations for the search. This effectively reduces the search range, which trades final motion estimation quality for faster or more consistent operation time.

1. Load Initial Memory
2. Configure Registers
3. Execute Search
   (a) Request Additional Pixels
   (b) Load Additional Pixels
   (c) Repeat Requests/Loads until Search Terminates
   (d) Return Search Results

An example execution of a pattern search, including search pattern configuration, is given in Table 5.35.

Table 5.35: An example instruction stream

Group Goal	**GRP RPT**	**Commands**	**Operand Values**	**RPT**	**CNT**
Load ACT Data	1	Write_Burst_ACT	-	1	2049
		Pixel Pair	2x ACT Pixels	2048	
Load REF Data	1	Set_Burst_REF_X	0	1	2053
		Set_Burst_REF_Y	0	1	
		Set_Burst_Width	64	1	
		Set_Burst_Height	64	1	
		Write_Burst_REF	-	1	
		Pixel Pair	2x REF Pixels	2048	
Configure Pattern Memory	23	Set_Write_Pattern_Addr	0	1	138
		Write_Pattern_DX	0	1	
		Write_Pattern_DY	0	1	
		Write_Pattern_JMP	9	1	
		Write_Pattern_VLD_Top	1111_1111	1	
		Write_Pattern_VLD_Bot	0000_0000	1	
Configure PMV (Predicted Motion Vector)	1	Set_PMV_DX	0	1	2
		Set_PMV_DY	0	1	
Choose Block	1	Set_BLKID	2	1	1
Set Threshold Value	1	Set_Thresh_Top	0000_0000_00	1	2
		Set_Thresh_Bot	00_0000_0000	1	
Set Active Block	1	Set_ACT_PT_X	0	1	2
		Set_ACT_PT_Y	0	1	
Set Search Center	1	Set_REF_PT_X	0	1	2
		Set_REF_PT_Y	0	1	
Begin Search	1	Start_Search	0	1	1

5.2 Compute Datapath

The execution unit consists of the pixel datapath and an execution controller, shown in Figure 5.11. The functions supported are burst write and read into the pixel memories and block pixel compares of the supported block shapes. These three functions end up being the base on which all the search operations work. This also means that the execution unit could be instantiated on a stand alone basis, or as part of a MEACC2 which could be tiled out for more throughput.

The datapath has an initial latency of 6 cycles. There are improvements that can be made to increase latency in order to improve throughput, but these enhancements are not in line with the serial nature of the configurable search patterns, since the largest patterns still only chain together 12 checkpoints before having to stop for an evaluation step. If block sizes were larger, than the throughput advantages of a deeper pipeline might be justified. The pipeline diagram for the pixel datapath is show in Figure 5.11

The upper level controller is in charge of executing the search and handling the I/O interactions with the FIFOs, so the execution controller only indicates when it is ready for the next data word. At synthesis, the critical path was the read path of the REF memory. This is due to the large (x256) output read multiplexor required by the REF memory's SCM. If the design was re-architected for 8x8 access, the path could be reduced, but there would be wasted accesses when dealing with the smaller block sizes (4x8 and 8x4). Further analysis on the ideal, or most common, block sizes present in average video sequences could reveal whether or not the tradeoff is justified.

5.2.1 Adder Architecture

The compression tree for the SADs is a classic CSA. It could be pipelined, at the cost of registers, but since it wasn't on the critical path, it was not split. If the reference frame memory is reworked to have a shorter critical path, then it may be necessary to revisit the compression tree and split it across pipeline cycles. The required bit widths of the entire SAD operation are given in Figure 5.12, these calculations are based on 8 bit wide pixels.

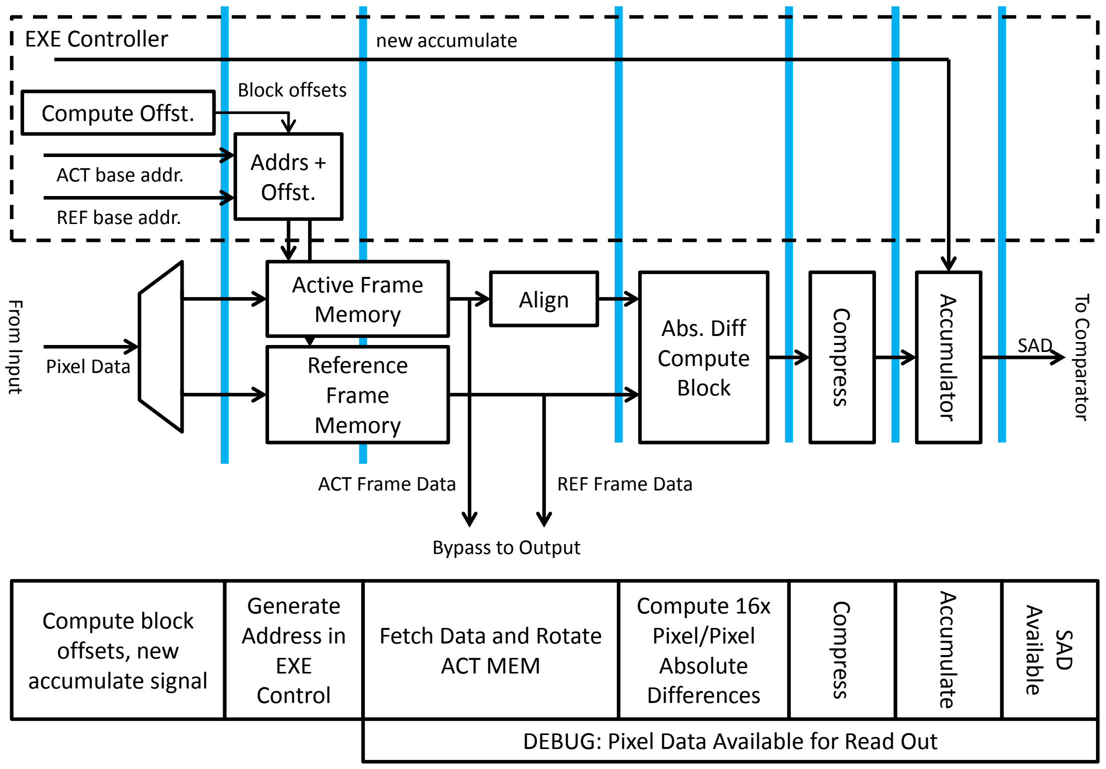


Figure 5.11: Pipeline diagram of the pixel datapath

5.3 Pixel Memory

Pixels are stored in two locations in MEACC2, active frame memory, and reference frame memory. These two memories act as 1st level caches of the image data while the search pattern is being executed and the SAD is being computed. They are both dual-pixel write and 4x4 block access. Pixels are written in pairs in raster-scan order, but are read in 16 pixel, 4x4 blocks from a single address. The read-address is the address of the top left corner pixel. Write addresses use the address of the leftmost pixel.

5.3.1 Line Access and Block Access Memory Architectures

Line access architectures are common for memories. The classic 1R/1W SRAM is a line-access architecture. Thea primary advantage of the line access architectures is they are simple to address, and are intuitive to use as they have a data-structure parallel in the 2-dimensional array. However, SAD operations are done primarily on MxN blocks of

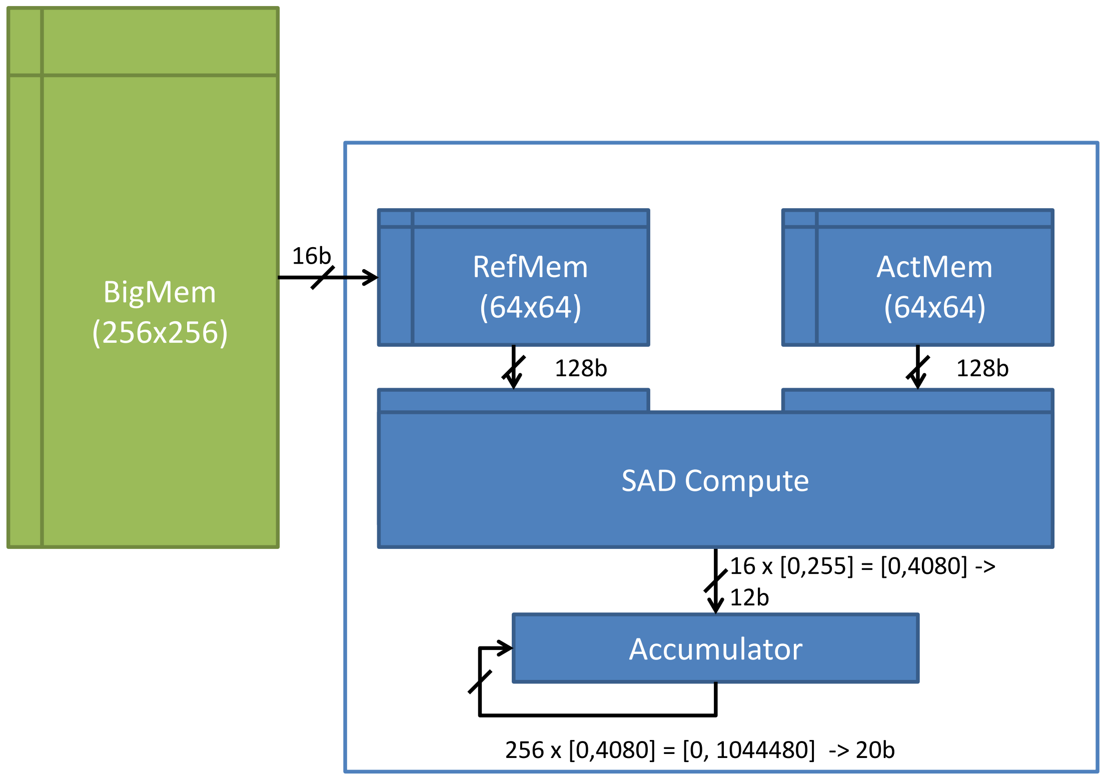


Figure 5.12: Required bit widths for full precision throughout the SAD compute process

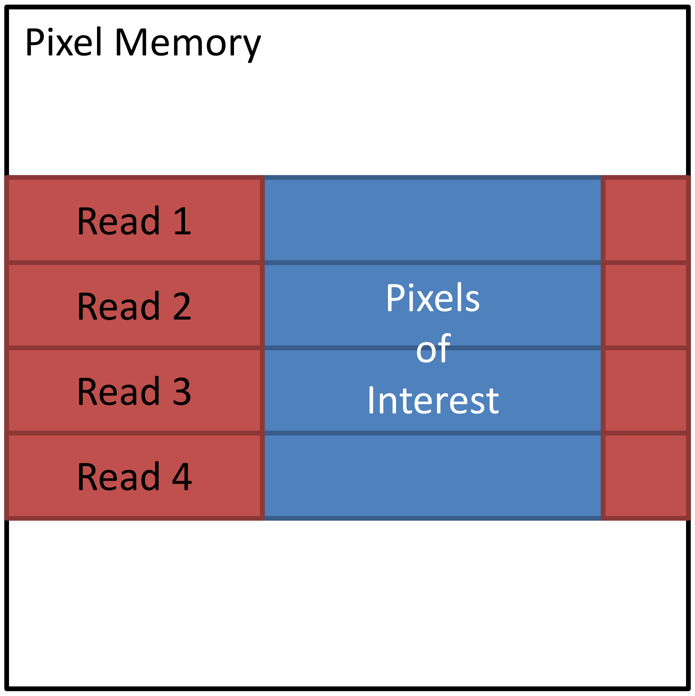


Figure 5.13: Line based memory access

pixels, and for these operations line-based architectures are inefficient, fetching pixels that are not used, or can only be consumed by additional hardware tiled out for the purpose, for example, systolic arrays. At the same time, the additional complexity of block-access memory architectures begins to be worth investigating further, especially for mostly-linear search patterns without wide block fanouts. An example of line-access memory pattern is shown in Figure 5.13, including the wastage from activating unnecessary pixels. An example of block-access memory pattern is shown in Figure 5.14.

5.3.2 SCMs and Block Access Memory Architectures

The core component of a standard cell memory in MEACC2 is a clock gated register. Based on Meinerzhagen's work, both latch and register based memories can be built from a standard flow. The latch based memories are more area efficient, and have a more robust operating profile at sub-threshold voltages, but the design target is high throughput through high frequency operation. For that goal, the register memory is superior [59].

The clock gate is made with the expected latch and logic gate, to prevent spurious

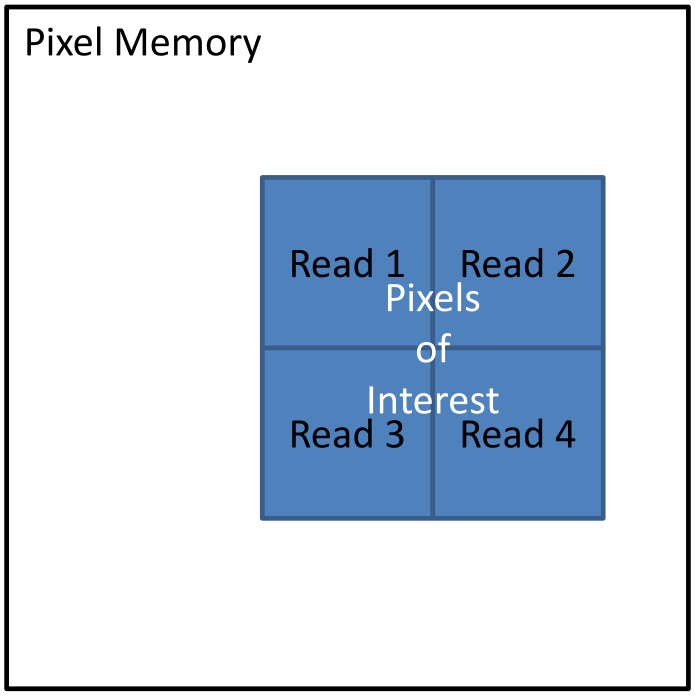


Figure 5.14: Block based memory access

writes due to glitches in the write enable signal. This primary block can be either a single bit, or a row of bits, which all share a clock gate. An example of an SCM style word row is shown in Figure 5.15. Each row of bits can then be combined into the words of a memory array with the enable signals of the clock gates being fed by the write enable signal from a write address decoder. This decoder produces a one-hot encoded signal for the SCM writes. To prevent read before write errors, the read has a forced latency of 1 cycle from the write. This means that if there is a simultaneous write and read from the same memory location (an illegal operation) the read takes the new value. Meinerzhagen also pointed out that a one-hot encoding for the read signal allows for a multiplexor that takes advantage of that encoding to prevent read glitches [60]. If it becomes necessary to squeeze out the most performance possible from our pipeline, that optimization can be sacrificed, or tile out more registers and bring the read-encoding across the pipeline stage. An example of a small SCM is shown in Figure 5.16.

The active (ACT) and reference (REF) frame memories are SCM arrays. The complexity of the arrays and their address decoders are a function of the kinds of access

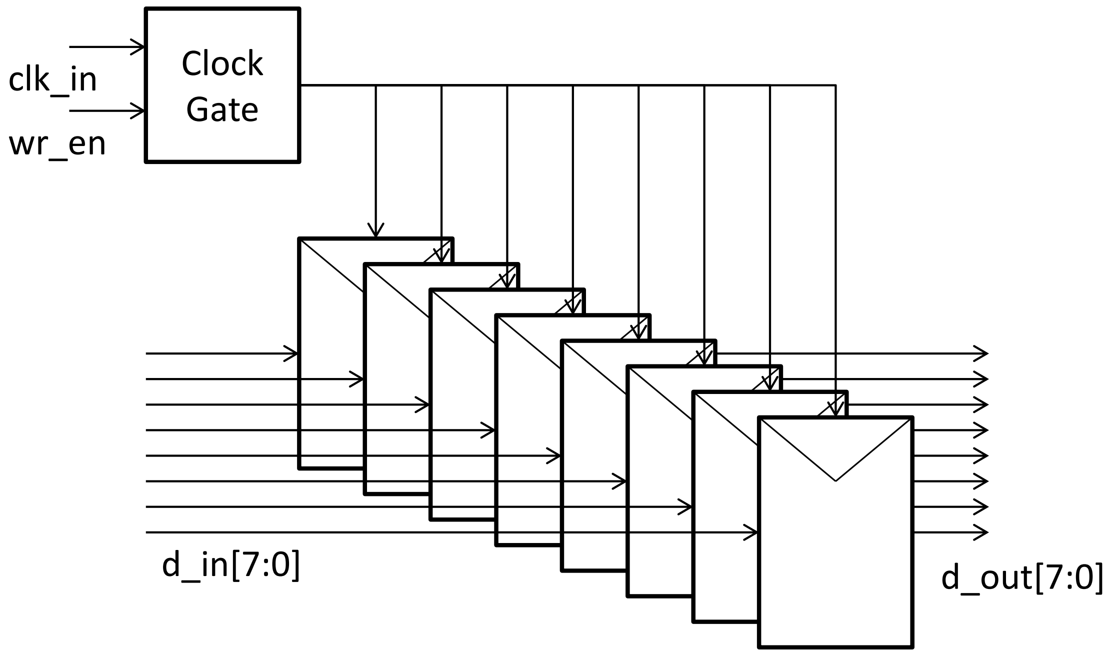


Figure 5.15: A word of standard sell memory

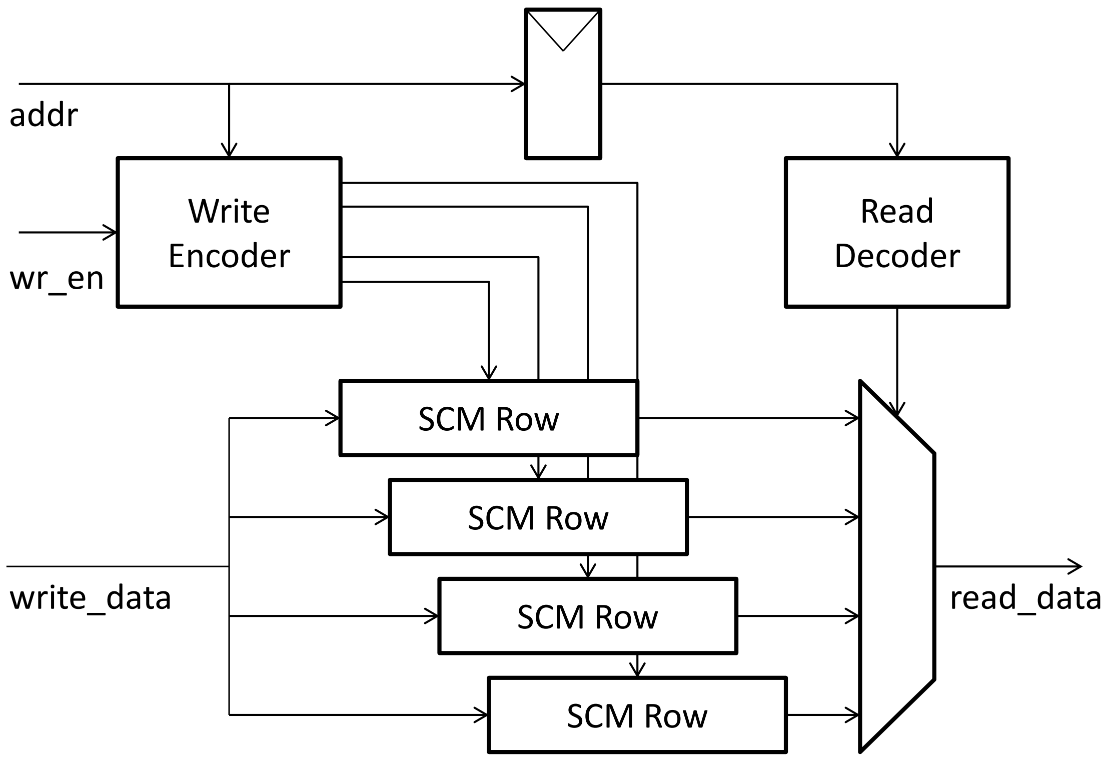
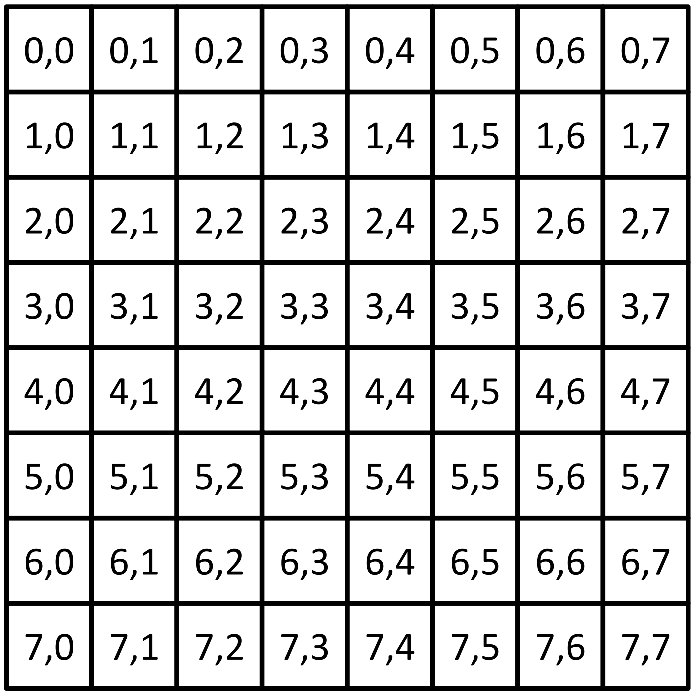


Figure 5.16: A multi-word standard cell memory



0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7

Figure 5.17: ACT memory access pattern

patterns necessary for each type of memory. ACT frame memory accesses are aligned along the possible CTU borders, as show in Figure 5.17. This means that to have a block-access architecture, there are a total of 4 memory banks, one for each line of the 4x4 block that is our atomic access component, because all possible CTU boundaries are multiples of 4. If the CTU boundaries were not multiples of four, the ACT memory would be more similar to the REF memory. The address decoder is a straightforward mod4 check against the y component of the address in order to see where in each of the memory banks the necessary pixels are. A diagram of the components of the ACT memory is given in Figure 5.18.

REF frame memory accesses have no easy pattern of alignment along multiples of 4. Therefore, any one pixel might be accessed by any of the 4x4 squares that encompass it, as shown in Figure 5.19. This means the REF memory must be made up of 16 memory banks. The address is separated into X and Y components, and goes through a two stage process to determine which memory bank a pixel is present in, and where in that memory bank it is stored. A diagram of the the components of the REF memory is given in Figure 5.20.

Both the REF and ACT memories produce the correct group of pixels, but those

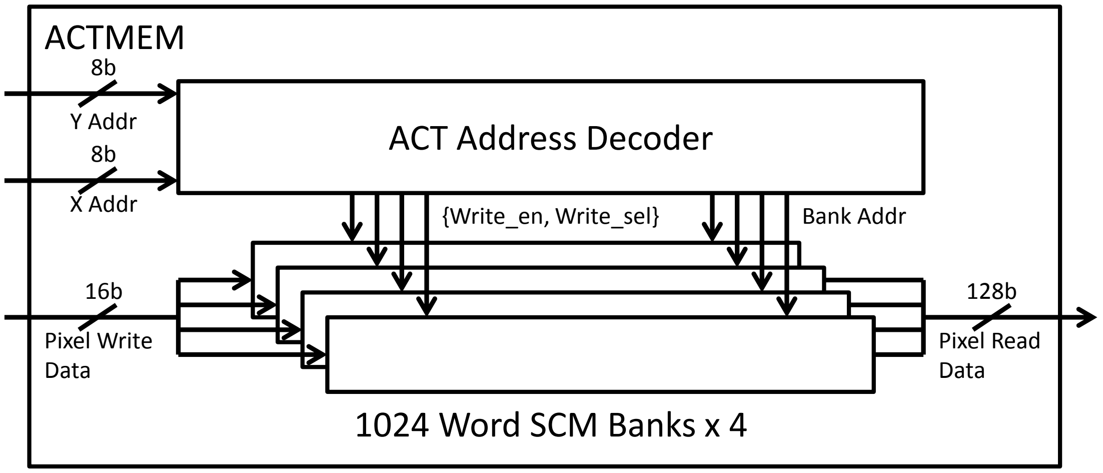


Figure 5.18: Component blocks of the ACT frame memory

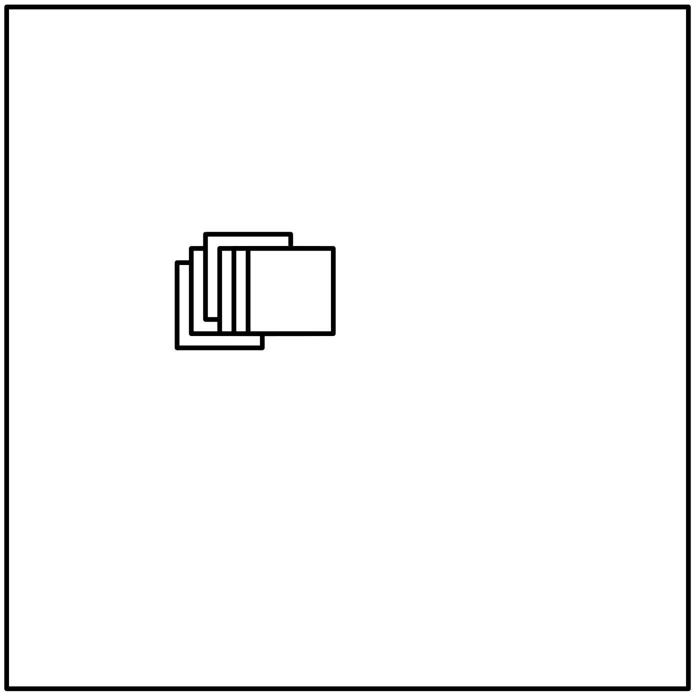


Figure 5.19: REF memory access pattern

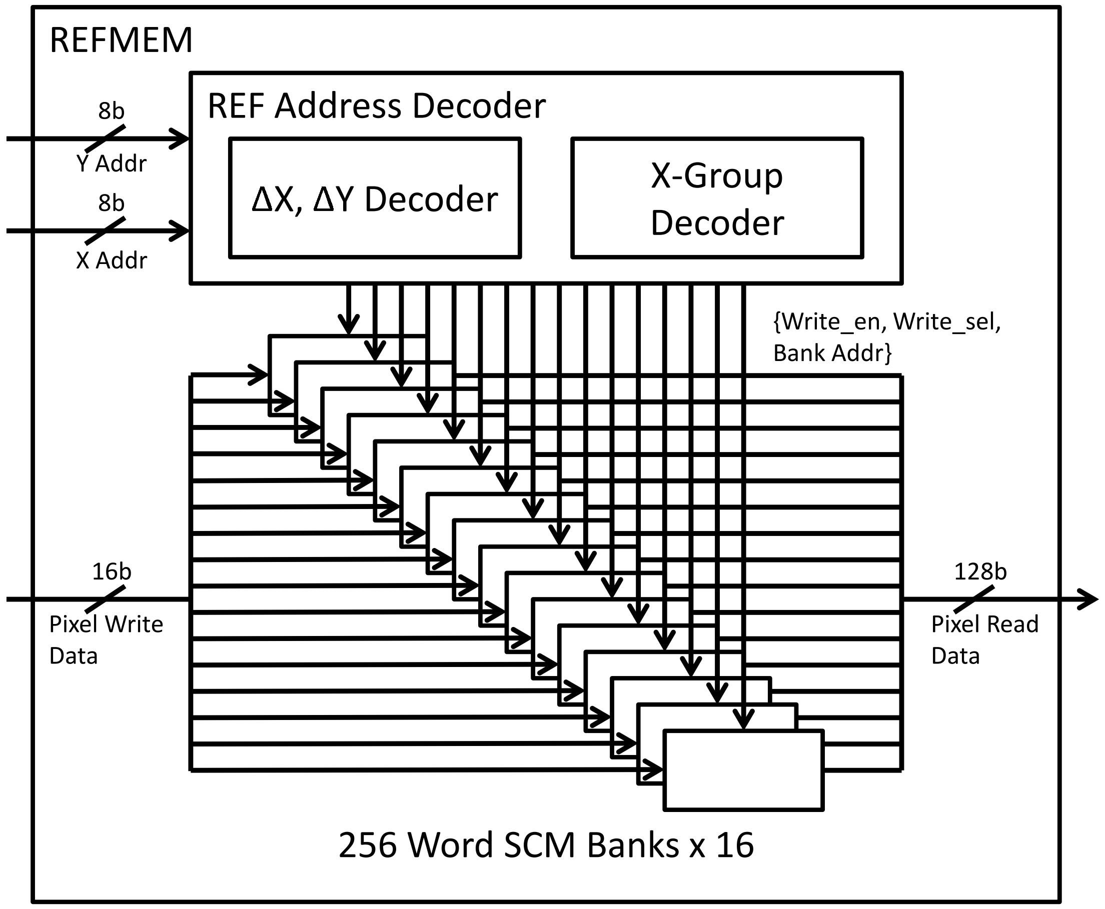


Figure 5.20: Component blocks of the REF frame memory

pixels are rotated based on how their memory address aligns with multiples of 4 in the X and Y direction. Since the compute goal is the sum of absolute differences between *all* of the corresponding pixel pairs, pixels are not rotated completely back to a zero rotation, but rather only enough so that each pixel is aligned with its partner. In each address decoder, a component of the address called *Sr* is computed, which locates the bank of the pixel, but also the rotation of the block if that pixel is the top left corner of the block access. These *Sr* values can be compared, and then only the output of the ACT pixels are pushed through a rotator to align them with their REF counterparts. The ACT pixels are rotated because the simpler design of the ACT memory decoder results in a shorter critical path through the ACT memory than through the REF memory. Therefore, given the choice between which pixels to rotate, the design should rotate the pixels not already on the critical path through the stage.

5.3.3 REF Memory Access Patterns

The REF and ACT frame memories are not sufficient to contain the whole image at once. The ACT memory is only loaded all at once, because each CTU block is stored in ACT memory only until its SAD is computed, and then the ACT can be refilled with new CTU blocks after all the current CTU blocks have been checked. The REF memory, however, can be loaded partially, and in the middle of a search. An out of bounds unit in the top level controller maintains the current REF memory state in the complete image, and make requests outside of MEACC2 for additional pixel memory if necessary. These pixel requests must be fulfilled in order, as the unit transitions into a pixel-receptive state once the request is issued. Based on the type of frame motion required to bring the next checking point into memory range, the request may be given in either one or two requests.

Cardinal directions only require a single burst write to complete, and so only generate one set of request words (remembering that each pixel request consists of a total of four words). This kind of frame pattern movement is shown in Figure 5.21. Diagonal directions require two burst writes to complete, and so generate two set of requests words. This kind of pattern movement is shown in Figure 5.22. The order of the requests is also shown in the Figure. If a pixel request completely replaces the REF memory (for example,

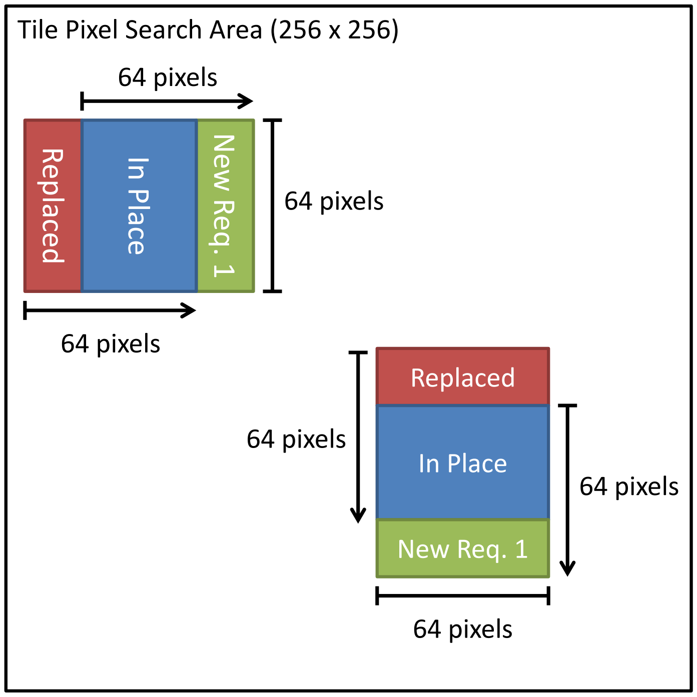


Figure 5.21: Memory replacement scheme for cardinal frame shifts

. New pixels are in green, retained pixels are in blue, and pixels cleared to make room for new pixels are in red. The pixel request is broken into two requests to take advantage of the burst pixel-write mode in the execution controller.

during a 64x64 block compare, the next checking point may require a complete refresh of the REF memory), the request is given in only one request.

5.3.4 A Smart Full Search Pattern leveraging Pixel Frame Locality

Since the memory subsystem maintains a coherent frame of pixels at all times, pixels close to each other in frame are brought into the memory system together. A full search must visit all possible points in the search area, but there is no given restriction on the order those points must be visited. Therefore, it is possible to reorder the visiting order of full search to take advantage of this inherent pixel locality. A diagram of a sector based full search pattern is shown in Figure 5.23. This smart full-search is what is implemented by MEACC2. This scheme prevents the MEACC2 from having to request the same pixels from the large frame memory multiple times, instead checking all valid blocks which are

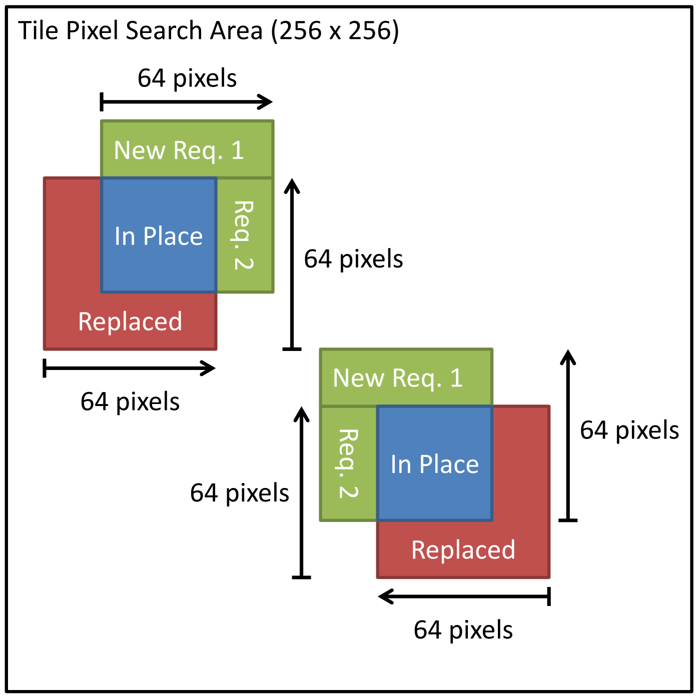


Figure 5.22: Memory replacement scheme for diagonal frame shifts

. New pixels are in green, retained pixels are in blue, and pixels cleared to make room for new pixels are in red. The pixel request is broken into two requests to take advantage of the burst pixel-write mode in the execution controller.

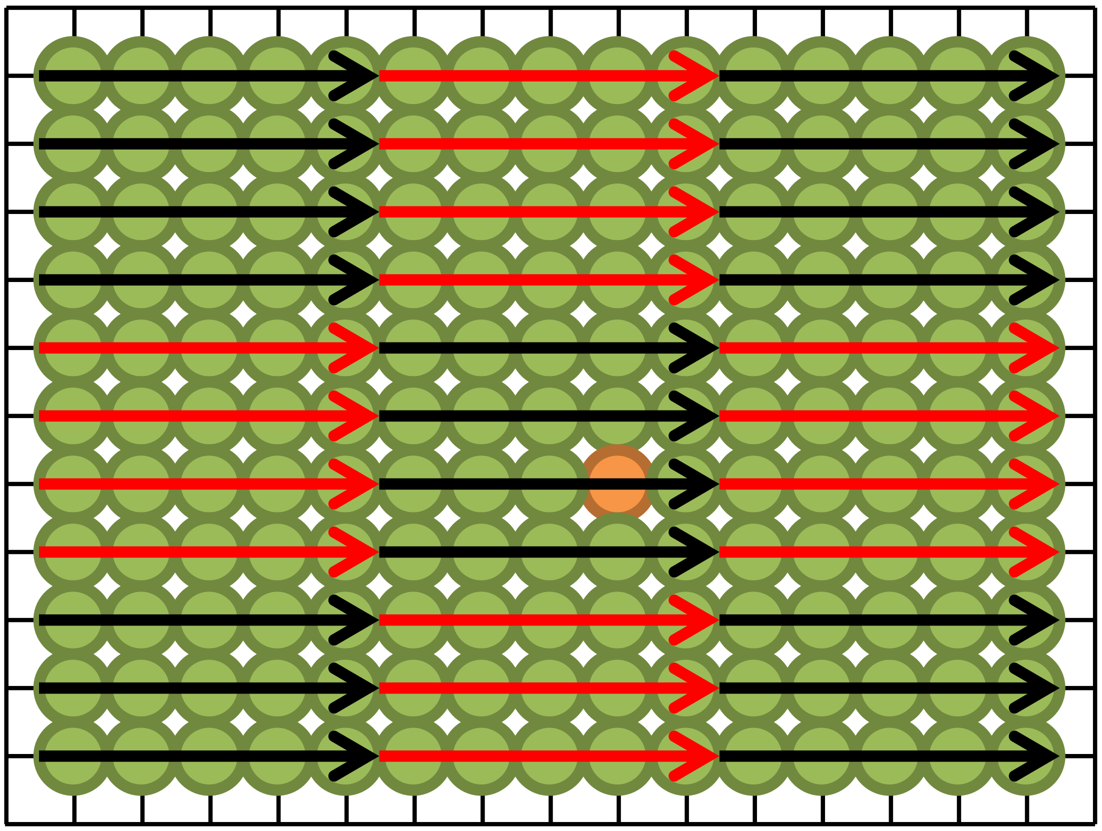


Figure 5.23: The pixel checking pattern of a sector based full search

. A block frame memory aware full search pattern will check points within the current cached memory before moving on. This full search has 6 sectors and 3 partial sectors. Once the first row in a sector has been checked, the rest of the pixels needed to check the row have already been brought into frame memory.

covered by its reference frame memory before moving on in the search.

This smart-search does require additional hardware at the full-search controller level, but saves many thousands of cycles transferring pixels from the larger memory system into the reference frame memory.

5.4 Pattern Memory

The pattern memory has two main components, an SCM and a ROM. The SCM and ROM are addressed together, with the top most bit of the address determining whether or not the data comes from the SCM or the ROM. This allows us to store common pattern endings in the ROM, as well as containing a full pattern search for debug and default purposes. Each row in a pattern contains the following fields: x offset, y offset, jump address, and valid bits. The overall structure of the Pattern Memory is shown in Figure 5.24. The x and y offsets are in relation to the current center of the pattern. The jump address is where in pattern memory the controller should jump to if this point is picked as the new center for the pattern, and the valid bits include the valid settings for the search stage if the current point is picked as the next center. These valid bits are used to skip repeated search locations when the same pattern stage is exercised multiple times as the center moves. These repetitions can be known ahead of time, and so removed. Each of the offsets are signed, and resolve to the full range of pattern addresses, and so take up 9 bits. The jump address points to an address in the overall pattern memory (either ROM or SCM), and so requires 6 bits. The valid bit field does not have a required size, but for this design I chose a size of 16 bits. This is sufficient to contain the 12-point stage pattern, and all the other most commonly used search patterns have fewer points per stage. A step by step walk through of how to store patterns in pattern memory is given in Section 5.4.1, using the built in ROM pattern as an example.

5.4.1 ROM Pattern

The ROM contains a set of useful patterns for either finishing a search, or as a stand alone. As a stand along it contains a three-stage search using a Diamond-8, Diamond-4 and Cross-1 pattern. It also contains a Diamond-2 Pattern that leads into a Cross-1 pattern as well. Any pattern stored in pattern memory can be directed to jump into the ROM, which allows configurable patterns to inherit the stages already stored in ROM. A decimal representation of the pattern ROM is shown in Table 5.36, and its binary equivalent is given in Table 5.37. A graphical representation of the pattern stored is shown in Figure 5.25. The ROM occupies the top 32 words of the pattern memory. Patterns which are configurable are located in words 0 - 31.

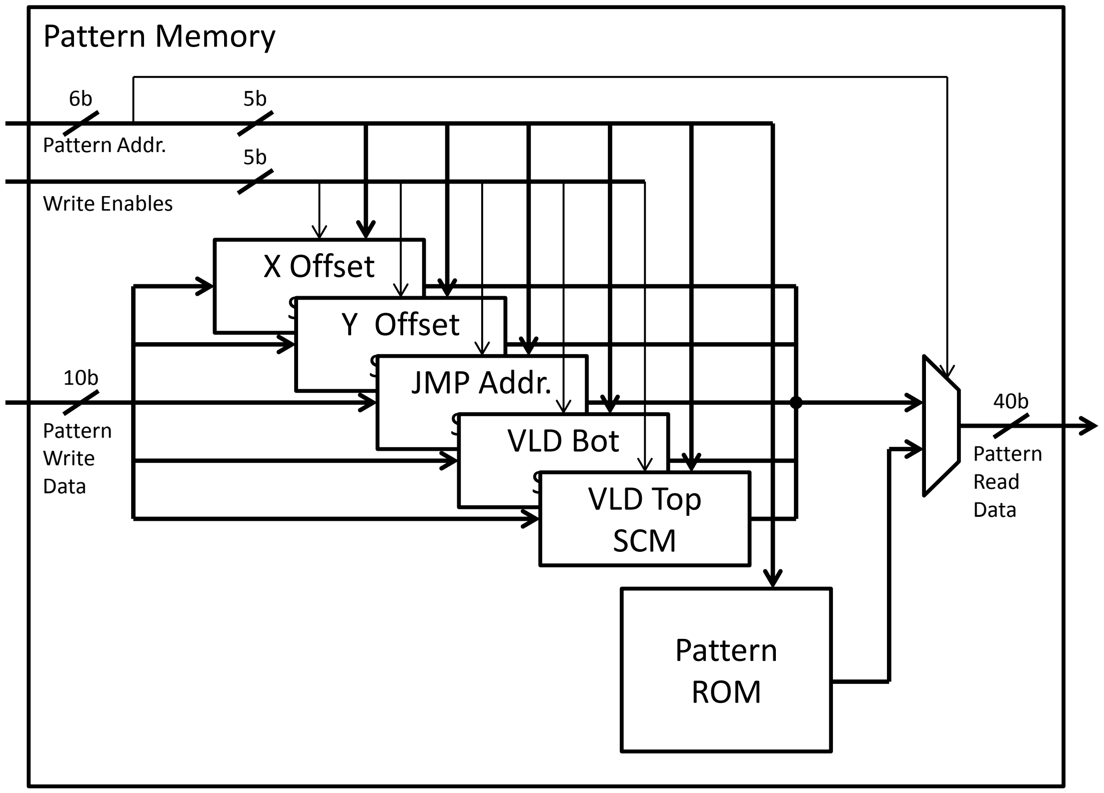


Figure 5.24: Component Blocks of the pattern memory

Table 5.36: Pattern ROM Contents in decimal

ADDR	X Offset	Y Offset	JMPADR	TopVLD	BotVLD	Stage	Loc
32	0	0	41	00000000	11111111	D8	C
33	0	-8	32	00000000	11000111	D8	1
34	-4	-4	32	00000000	00000111	D8	2
35	-8	0	32	00000000	00011111	D8	3
36	-4	4	32	00000000	00011100	D8	4
37	0	8	32	00000000	01111100	D8	5
38	4	4	32	00000000	01110000	D8	6
39	8	0	32	00000000	11110001	D8	7
40	4	-4	32	00000000	00111110	D8	8
41	0	0	59	00000000	00001111	D4	C
42	0	-4	41	00000000	11000111	D4	1
43	-2	-2	41	00000000	00000111	D4	2
44	-4	0	41	00000000	00011111	D4	3
45	-2	2	41	00000000	00011100	D4	4
46	0	4	41	00000000	01111100	D4	5
47	2	2	41	00000000	01110000	D4	6
48	4	0	41	00000000	11110001	D4	7
49	2	-2	41	00000000	00111110	D4	8
50	0	0	59	00000000	00001111	D2	C
51	0	-2	50	00000000	11000111	D2	1
52	-1	-1	50	00000000	00000111	D2	2
53	-2	0	50	00000000	00011111	D2	3
54	-1	1	50	00000000	00011100	D2	4
55	0	2	50	00000000	01111100	D2	5
56	1	1	50	00000000	01110000	D2	6
57	2	0	50	00000000	11110001	D2	7
58	1	-1	50	00000000	00111110	D2	8
59	0	0	59	00000000	00000000	C1	C
60	0	-1	59	00000000	00001011	C1	1
61	-1	0	59	00000000	00000111	C1	2
62	0	1	59	00000000	00001110	C1	3
63	1	0	59	00000000	00001101	C1	4

Table 5.37: Pattern ROM contents in binary

ADDR	X Offset	Y Offset	JMPADR	TopVLD	BotVLD	Stage	Loc
100000	000000000	000000000	101001	00000000	11111111	D8	C
100001	000000000	111111000	100000	00000000	11000111	D8	1
100010	111111100	111111100	100000	00000000	00000111	D8	2
100011	111111000	000000000	100000	00000000	00011111	D8	3
100100	111111100	000000100	100000	00000000	00011100	D8	4
100101	000000000	000001000	100000	00000000	01111100	D8	5
100110	000000100	000000100	100000	00000000	01110000	D8	6
100111	000001000	000000000	100000	00000000	11110001	D8	7
101000	000000100	111111100	100000	00000000	00111110	D8	8
101001	000000000	000000000	111011	00000000	00001111	D4	C
101010	000000000	111111100	101001	00000000	11000111	D4	1
101011	111111110	111111110	101001	00000000	00000111	D4	2
101100	111111100	000000000	101001	00000000	00011111	D4	3
101101	111111110	000000010	101001	00000000	00011100	D4	4
101110	000000000	000000100	101001	00000000	01111100	D4	5
101111	000000010	000000010	101001	00000000	01110000	D4	6
110000	000000100	000000000	101001	00000000	11110001	D4	7
110001	000000010	111111110	101001	00000000	00111110	D4	8
110010	000000000	000000000	111011	00000000	00001111	D2	C
110011	000000000	111111110	110010	00000000	11000111	D2	1
110100	111111111	111111111	110010	00000000	00000111	D2	2
110101	111111110	000000000	110010	00000000	00011111	D2	3
110110	111111111	000000001	110010	00000000	00011100	D2	4
110111	000000000	000000010	110010	00000000	01111100	D2	5
111000	000000001	000000001	110010	00000000	01110000	D2	6
111001	000000010	000000000	110010	00000000	11110001	D2	7
111010	000000001	111111111	110010	00000000	00111110	D2	8
111011	000000000	000000000	111011	00000000	00000000	C1	C
111100	000000000	111111111	111011	00000000	00001011	C1	1
111101	111111111	000000000	111011	00000000	00000111	C1	2
111110	000000000	000000001	111011	00000000	00001110	C1	3
111111	000000001	000000000	111011	00000000	00001101	C1	4

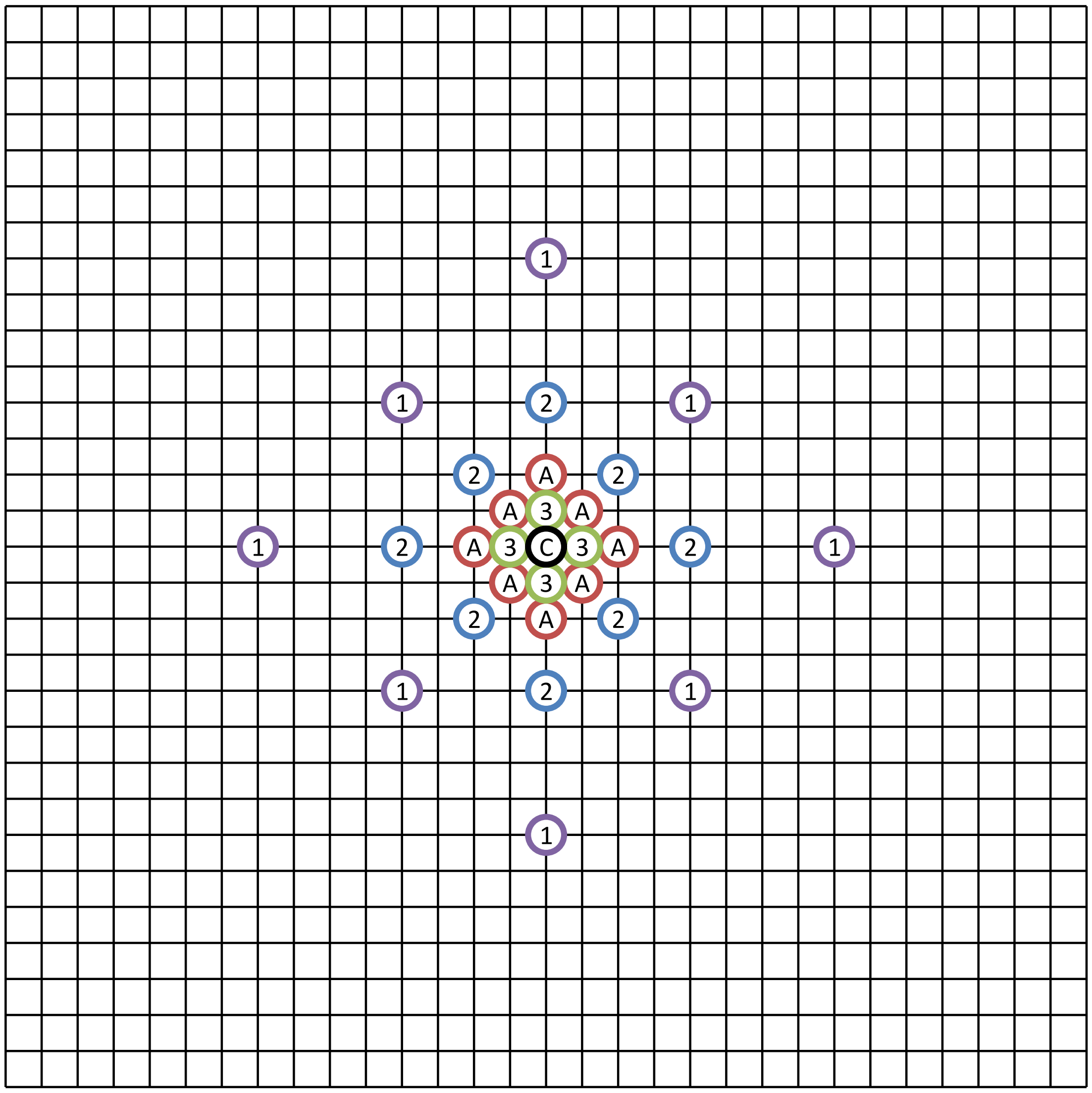


Figure 5.25: 4-Stage pattern stored in ROM

5.4.2 A 12 Point Circular Search Pattern

Patterns try to capture the full range of motion within an image, in the minimum number of points. A cross pattern, for instance, captures motion in only the cardinal directions, while a diamond pattern captures motion in both the cardinal and diagonal directions. Hexagonal patterns capture motion, biased in either the horizontal or vertical direction depending upon the type of hexagon (type A or type B). All of these search patterns were developed in the context of H.264 and previous standards, where the maximum image size only went to 1080p. Direction in the cardinal direction and the diagonals, then, would capture most of the movement possible in a particular frame. With larger image sizes, up to 4x the size of 1080p to start, motion within the image may fall within the areas missed by cardinal and diagonal motion vectors. At the same time, H.265 brings in additional motion vectors as possible candidates and with process shrink, the actual *computation* of a candidate SAD, once its relevant pixels have been brought into memory, is also less expensive. Therefore, additional patterns which contain more search points (and require more compute), but cover more possible motion vectors, can become relevant. A 12 point circular pattern, with a three-stage example shown in Figure 5.26, balances keeping the total number of points searched low, while still covering more possible motion directions. It also has the same overlapping characteristics of diamond, cross, and hexagonal patterns, where repeated searches at the same stage have overlapping check points which can be skipped, as shown in Figure 5.27, Figure 5.28, and Figure 5.29. The rest of the re-use movements are symmetrical about the X and Y axis. This reuse of 3 points is less than the reuse of the diamond pattern, which reuses either 3 or 5 points depending upon the movement type, comparable to hexagonal patterns which also reuse 3 points, and results in less distortion on average than the cross pattern, which reuses only 1 point. Table 5.38 gives a breakdown of points reuse in different patterns, excluding the center point of the pattern. Even as a percentage measure, the Circular pattern compares favorably to the cross, while checking 3 times the total number of points.

Table 5.38: Point reuse between stages in various search patterns

Pattern	NumPts	Reuse	Reuse Pct.
Cross	4	1	25%
Diamond	8	3 or 5	38% - 50%
HexA	6	3	50%
HexB	6	3	50%
Circular	12	3	25%

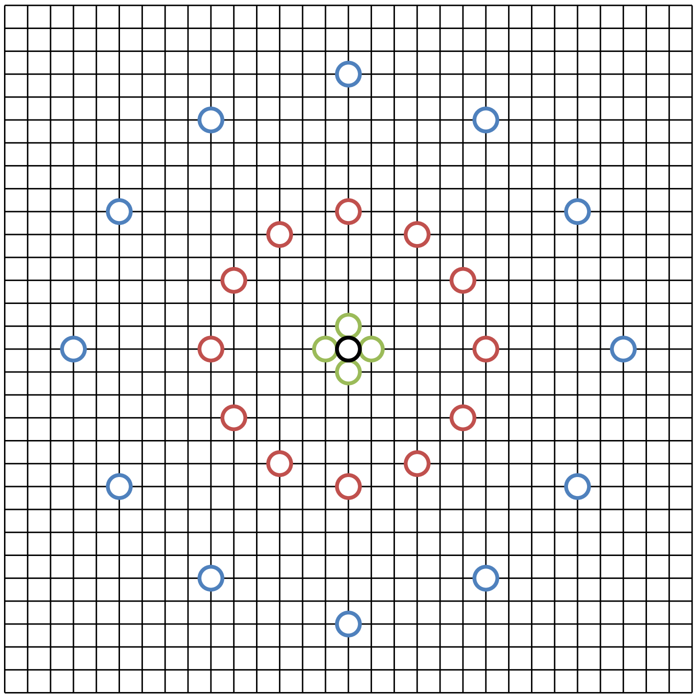


Figure 5.26: 3-Stage, 12-point circular pattern

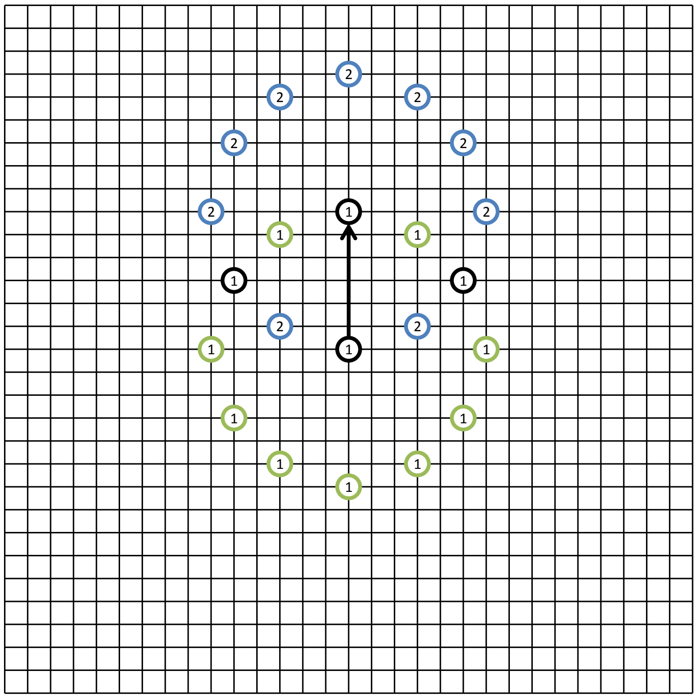


Figure 5.27: Circular pattern type I reuse

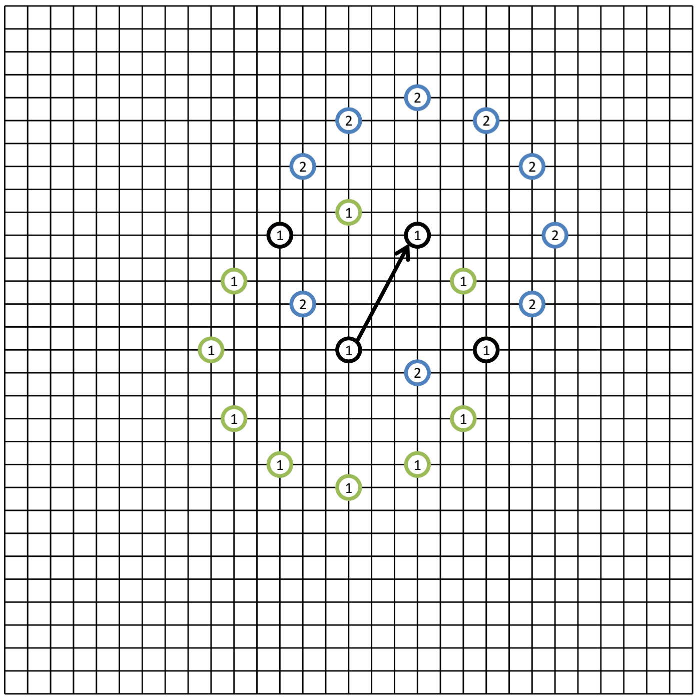


Figure 5.28: Circular pattern type II reuse

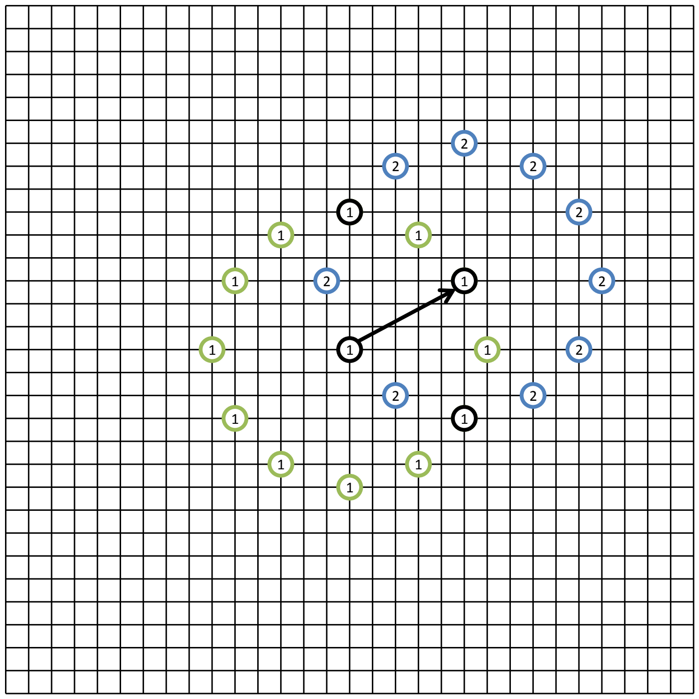


Figure 5.29: Circular pattern type III reuse

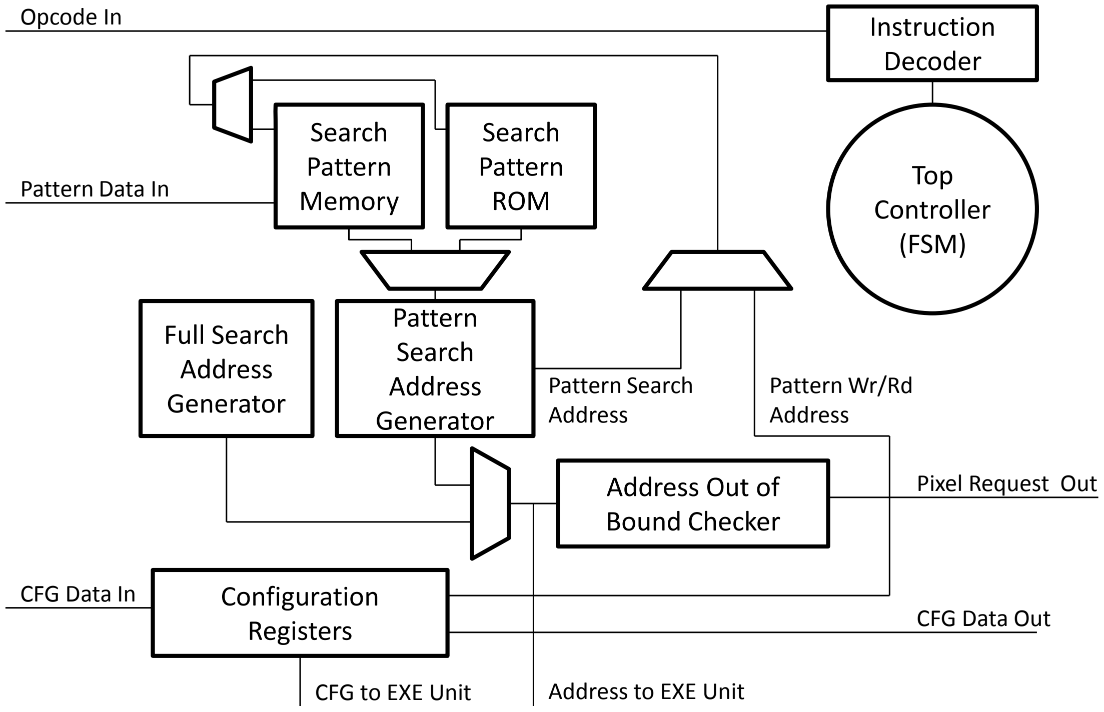


Figure 5.30: Controller circuitry

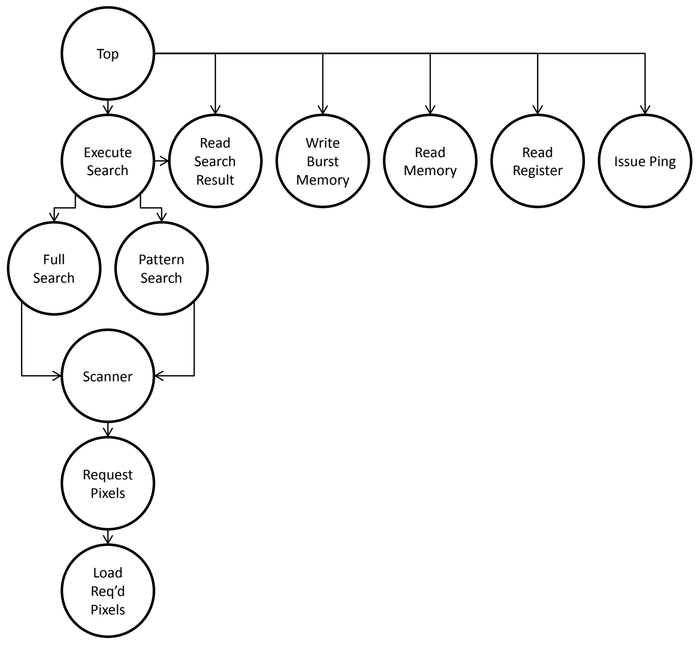


Figure 5.31: Hierarchy of the top control unit

5.5 Control Units

The control unit consists of the configuration registers, pattern memory, full-search address generator, pattern-memory address generator, out of bounds point checker, the controller FSM, and an instruction decoder, as shown in Figure 5.30. The instruction decoder samples the op-code bits of every input word and translates these into control signals for the controller FSM. In order to prevent random bits in the pixel transfers from being misinterpreted, all instruction decode signals pass through the controller FSM, where they are masked if the controller is not in an instruction-receiving state. Both address generators can generate the next inspection point for either a smart full-search of a pattern search run out of the pattern memory. The address out of bound checker, combined with the controller FSM handles pixel replacement.

The top FSM controller is not a single FSM. Instead it is a series of hierarchical FSMs. These hierarchical FSMs are built so that there is no latency lost when traveling down the hierarchy, which requires careful handling of the idle states in each machine. This allows us to retain the full efficiency of a fully integrated top level FSM, without paying

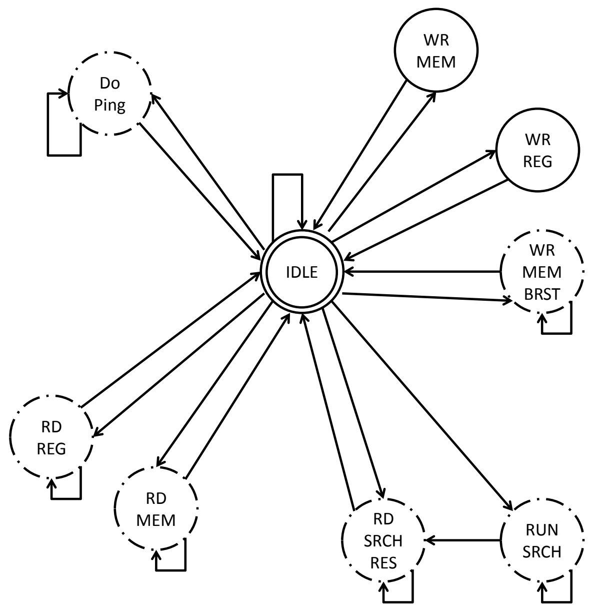


Figure 5.32: State diagram of the top level controller

. States which trigger other FSMs are given in dashed circles, and the reset state is shown with a double circle.

as much of the complexity price in terms of machine analysis and difficulties in correct implementation. The list of the component FSMs, and the relational hierarchy, is shown in Figure 5.31. Since both full search and pattern search make use of pixel replacement, the actual implementation of the execute search contains mux logic to arbitrate between which FSM has control of the scanner FSM. The state transition diagram is shown in Figure 5.32 with the hierarchical FSMs marked in dashed borders. The return to IDLE behavior adds latency to the rare register and pattern memory writes. Searches and their associated memory operations are handled by a lower level state machine and are set up to be pipelined. The read out commands have their own state machines so that MEACC2 can stall correctly if its output FIFO is full.

5.6 Output Block

The output block captures signals of interest from MEACC2, and outputs them to the FIFO in a fixed order depending upon the operation required. It contains the multiplexor tree, counters to manage word-by-word output operations (such as 4-word pixel requests, or 8-word paired pixel requests). It takes its control signals from the overall MEACC2 controller. Data is universally 16 bits wide, to conform with the width of the output FIFO. There is space in the output control for up to 9 more 16-bit registers. Right now, unassigned register values are configured to return a register read value of 1.

Chapter 6

ME2 Physical Data

MEACC2 went through place and route targeting a 65 nm CMOS technology node. At an expected supply voltage of 1.3 V, MEACC2 operates at a maximum frequency of 812 MHz while dissipating 79.8 mW. By scaling the supply voltage to 0.9 V, MEACC2 operates at a maximum frequency of 158 MHz and dissipates 8.06 mW. Table 6.1 summarizes the results of place and route, and Figure 6.1 shows the dieplot with the major memory areas outlined.

Table 6.1: MEACC2 at a Glance

Name	MEACC2
Frequency	812–158 MHz
Power	79.8–8.06 mW
Supply	1.3–0.9 V
Total Area	1.041 mm^2 (3 AsAP Tiles)
Block Dimensions	1020.25µm^2×1020.30µm^2
OnDie Memory	10 KB SCM blocks
Pixels Compares per Cycle	16, in a 4x4 block
Supported Block Sizes	8×8 to 64×64 pixels
Largest Supported Tile Size	256×256 pixels

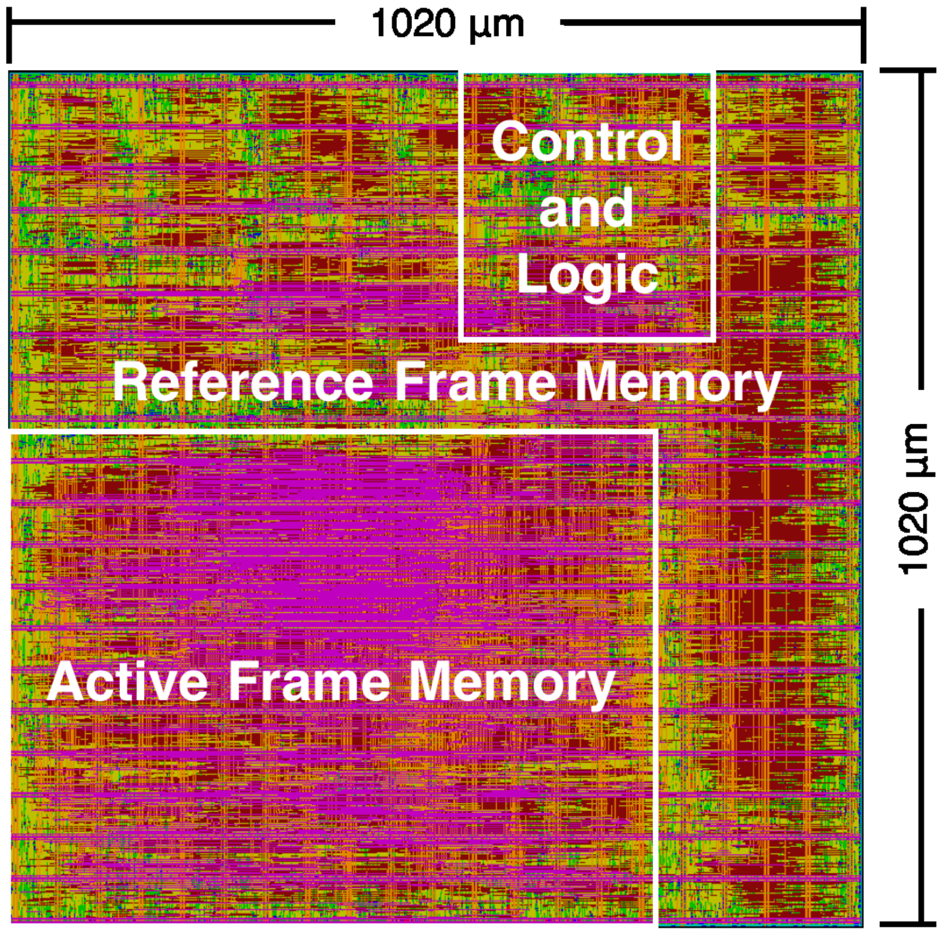


Figure 6.1: A plot of the physical layout of the MEACC2.

Chapter 7

Matlab Model

Modeling can be a way of quickly estimating the value of architectural changes, and models can also be extended to act as the basis for initial pre-silicon validation of a device. A model was designed and implemented in matlab to both verify the operation of the device, and also produce test vectors for both pre and post silicon validation.

7.1 Model

The model is designed to be a helper tool for future implementers of an AsAP based Codec. Additionally, it is possible for later AsAP generations to have multiple instantiations of MEACC2 in their core network. To enable both of these functions, the model is written as a Matlab class, which can be instantiated multiple times. The model does *not* directly emulate a cycle-by-cycle operation, but instead computes the expected cycle latency and reports that, as well as a cost function, at the completion of an instruction. This saves time, and also keep the model's code at a higher level of abstraction, so that the actual operation is not obfuscated.

Due to the level of abstraction necessary for fast operation, the model does not actually emulate pixel memory operations. Instead, the model tracks the current position of the box that contains the entirety of current pixel memory, moving that tracking when appropriate (when it is changed in the course of a search, or if new pixels are loaded into pixel memory). This means that the model *does not* cover the case where a new frame begins processing, but the new pixel data has not yet been loaded into MEACC2. Future modelers must be careful to appropriately handle the model when transitioning between frames. In my own modeling, I made sure to reload the pixel memory when changing active or reference frames.

7.2 Implementation as a Class

The model is built around the sad_comp function, which computes the SAD between two given blocks of pixels in the reference and active frame memories. This function is then progressively wrapped by additional functionality. There are no regressive calls, but the model *is not* coded in a purely functional manner. There are object level variables which are modified by functions, even though they do not appear in that function's call.

The model can be instantiated with differing sizes of active frame memory, reference frame memory, pattern memory, and image sizes. The image size is a virtual construct, but its inclusion means that the model can be expanded to handle arbitrarily large images. The actual physical MEACC2 device is limited by its IO width constraints and memory address space, but the model should still be a usable tool for further exploring the design space of motion estimation accelerators.

7.3 Automatic Test Generation and Transcription

The model has a property, CREATE_TRANSCRIPT, which defaults to a value of 0, but if set enables four different kinds of logging. These logs persist as long as the model object does, are public properties, and are implemented as matrices. The four different kinds of logs are:

- Setup Logging
- Pixel Request Logging
- Search Result Logging
- Points Checked Logging

Setup logging generates a row every time a model starts a search. The row contains the values of all pertinent registers for the search. In the same order as given in Table 7.1, the expanded names are: *predicted motion vector dx, predicted motion vector dy, block id, threshold value, active frame x coordinate, active frame y coordinate, output register value, pixel memory frame position x coordinate, pixel memory position y coordinate, decimation factor in the x direction, decimation factor in the y direction, and the search pattern address.* Not all of these values are pertinent for both pattern search and full search, for example pattern searches do not care about decimation factors. If a value is not pertinent for a particular search, then those values are recorded as zero in the transcript.

Table 7.1: Setup transcript format

pmvdx	pmvdy	blkid	thresh	ax	ay	oreg	imx	imy	decx	decy	pat_addr

Pixel request logging tracks the pixel requests made by MEACC2 through its output FIFO. Those requests, while coming out over multiple FIFO words, are condensed into a single transcript row. In the same order as given in Table 7.2, the expanded names are: *x coordinate of the top left corner, y coordinate of the top left corner, width, and height.* In diagonal pixel memory movements, MEACC2 issues a pair of pixel requests, and therefore adds two rows to the pixel request logging transcript.

Table 7.2: Pixel request transcript format

X	Y	W	H

Search result logging tracks the final search result output that MEACC2 places on its output FIFO. That report, as with the pixel requests, is made over multiple FIFO words, but is condensed into a single row for the transcript. In the same order as given in Table 7.3, the expanded names are: *the x coordinate of the top left corner of the matched pixel block, the y coordinate of the top left corner of the matched pixel block, and the SAD value for the match.* The SAD value is modeled in matlab as an integer, but should never exceed the value of a 20 b unsigned number. If it does, something has gone wrong.

Points checked result logging tracks which points are considered over the course of a search pattern. In this case, the SAD value of any pair of points was not of interest to

Table 7.3: Search result transcript format

X	Y	SAD

me in my debugging, so it is not logged. That would be a useful extension of the logging functionality, if future students want to experiment with more rigorous regression testing. These log only contains the X and Y coordinates of each checked point in absolute terms.

Table 7.4: Points checked transcript format

X	Y

Points checked logging was used primarily for debugging purposes during RTL validation. The other three logs can be consumed by calling "generate_test_from_model_run", which creates a testbench ready pair of stimulus files, one file for input and the other for expected output, on a model instance that has valid transcripts. The top level function leverages a pair of helper functions named "test_input_gen" and "test_output_gen" which can also be used independently to generate valid MEACC2 instruction words.

7.4 Cost Functions

The Matlab model maintains a cost function which tracks the cost of each search. The cost is meant to be a high-level estimate of the total number of cycles required to process a pixel block. The cost function is broken into two parts, cycles spent on transferring pixels into the block, and cycles spent computing the SADs. The pixel throughput is fixed at 2 pixels per cycle, so every two pixels transferred cost 1 cycle. Computes are done 4x4 pixels at a time, but depending upon block sizes a SAD compute could cost 4 (for an 8x8 block) or 256 cycles for a 64x64 block compare. This cost function should systematically underestimate the total number of cycles in both categories. In pixel transfers, the cost is underestimated because it assumes 100% FIFO utilization and no latency from pixel request to pixels being available. In SAD computes, it only counts the cycles necessary to perform the SAD, but does not take into account start latency, nor empty pipeline cycles due to decisions being made for the top level search. These error rates should be similar between different not-full pattern searches. Full-searches have slightly different behavior due to the different decision unit used to run the search. So our strategy for estimating the performance of MEACC2 then, is to use the model to generate costs, correlate those costs by simulating those transcripts in Modelsim and developing cost modifiers for both pixel transfer and SAD compute, and then use those correlations to make larger predictions using our body of generated costs.

Chapter 8

Simulation Results

The matlab model of MEACC2, combined with the RTL and Modelsim lets us begin to analyze the potential performance of the architecture.

8.1 Cost Function Correlation

Our cost function properly has two missing factors, the efficiency of the FIFOs and the efficiency of the computes. FIFO performance can be predicted based on bit width and operating frequency, but the FIFO efficiency is based on the overall system's ability to handle pixel requests from the large memory. The compute efficiency can be estimated as a function of block size and the number of points checked per pattern stage.

8.1.1 FIFO Limits

Assuming perfect memory usage (no repeated memory accesses), perfect FIFO usage (the FIFOs are transmitting useful data 100% of the time), and perfect pixel latency from the external memory establishes an upper bound of performance based purely on inter-chip communication block, and see that is should be sufficient for the current data throughput target. Table 8.1 gives effective FIFO throughput at various usage levels and operating frequencies, while Table 8.2 gives the throughput required to transfer at the specified framerate for each video format.

Taking Table 8.1 and Table 8.2 together, the 16b FIFO with 50% utilization can

Table 8.1: 16b FIFO throughput

	Throughput, Mpix /s			
Freq (GHz)	100%	75%	50%	25%
0.45	900	675	450	225
0.55	1100	825	550	275
0.65	1300	975	650	325
0.75	1500	1125	750	375
0.85	1700	1275	850	425
0.95	1900	1425	950	475
1.05	2100	1575	1050	525
1.15	2300	1725	1150	575
1.25	2500	1875	1250	625
1.35	2700	2025	1350	675
1.45	2900	2175	1450	725
1.55	3100	2325	1550	775
1.65	3300	2475	1650	825

Table 8.2: Video format throughput requirements

				Mpix / s	
Name	X	Y	Pixel Count	30 FPS Req	60 FPS Req
QCIF	176	144	25344	1	2
CIF	352	288	101376	3	6
480p	640	480	307200	9	18
720p	1280	720	921600	28	55
1080p	1920	1080	2073600	62	124
2160p	3840	2160	8294400	249	498
4320p	7680	4320	33177600	995	1991
Digital 4K	4096	2160	8847360	265	531
IMAX	5616	4096	23003136	690	1380

produce sufficient memory throughput to handle all video formats in 30 FPS if the device operating frequency can hit 1.05 GHz. Lower frequencies, or lower utilization result in MEACC2 operation being FIFO bound.

8.1.2 Compute Limits

Given that the compute datapath executes a 4x4 pixel search with a latency of 6 cycles and a throughput of 1 4x4 block per cycle, what is the maximum bound on our performance if the pipeline is kept constantly full. This establishes an upper bound on the expected performance based purely on the pixel datapath. How does this relate to the compute requirements of various target performance points, what does it suggest for future devices? Compare pattern search using cost functions from the matlab model.

The stage efficiency, E is a function of how many cycles it takes to process a block, the amount of latency to begin processing, and how long it takes to make a decision at the end of a stage. This implies, that given a perfect memory subsystem, the whole efficiency is determined by the relationship between start up latency, decision latency, and how many cycles are taken to process a block.

$$E_{Stage} = \frac{N \times (Cycles/Block)}{L_{Startup} + N \times (Cycles/Block) + L_{Decision}}$$

Table 8.3 demonstrates the pattern of efficiency usage across various points per stage and block sizes. Effectively, the larger each particular unit of computation gets, the less of an impact the initial latency of the datapath and decision.

8.2 Pattern Search Performance

MEACC2 is configurable, and can support arbitrary stage-based patterns. Previous work called for the pattern memory to be resynthesized for each type of pattern, but by fixing the maximum pattern size at 16 points, MEACC2 can be implemented with a configurable pattern memory. This also allows for a direct performance comparison between different search pattern algorithms on the same set of video data and making use of the same memory and datapath architectures. Each of the pattern shown in Tables 8.4 through 8.9

Table 8.3: Compute efficiency of a 16xSAD 6 cycle pipeline, 2 cycle decision unit

	Block Size		Stage Efficiency					
Cycles / Blk	X	Y	3	5	7	9	13	FS Eff
2	4	8	42.86%	55.56%	63.64%	69.23%	76.47%	99.994%
2	8	4	42.86%	55.56%	63.64%	69.23%	76.47%	99.994%
4	8	8	60.00%	71.43%	77.78%	81.82%	86.67%	99.997%
8	8	16	75.00%	83.33%	87.50%	90.00%	92.86%	99.998%
8	16	8	75.00%	83.33%	87.50%	90.00%	92.86%	99.998%
16	16	16	85.71%	90.91%	93.33%	94.74%	96.30%	99.999%
32	16	32	92.31%	95.24%	96.55%	97.30%	98.11%	100.000%
32	32	16	92.31%	95.24%	96.55%	97.30%	98.11%	100.000%
64	32	32	96.00%	97.56%	98.25%	98.63%	99.05%	100.000%
128	32	64	97.96%	98.77%	99.12%	99.31%	99.52%	100.000%
128	64	32	97.96%	98.77%	99.12%	99.31%	99.52%	100.000%
256	64	64	98.97%	99.38%	99.56%	99.65%	99.76%	100.000%

were run with a fixed block size of 8x8, and each pattern is a three stage pattern with similar spreads. The hybrid pattern uses diamond patterns in the first two stages, and then a cross pattern in the last stage. The circular pattern uses a circular pattern for the first two stages and then finishes with a cross pattern as well, to match the pattern proposed earlier. The MAE is the *mean absolute error* across all frames and candidate blocks. Percentage differential is computed from the smallest error-case of each class, for example in Table 8.5, the pattern that finds the smallest MAE is a circular pattern, while the pattern that checks the least number of points per block is the cross pattern.

The MEACC2 can only handle a tile of up to 256x256 pixels, and these images are significantly larger, so the first step is to divide the image into tiles, and then process each of those tiles in turn. This does mean that the search only takes place within a particular tile, so the results differ from a straightforward full-search across the entire image. Overall, the 832x480 video streams are partitioned into 8 tiles (3 full tiles and 5 partial tiles), and the 1280x720 streams are partitioned into 15 tiles (8 full tiles, and 7 partial tiles). When considering performance limitations, these tiles could be each spread to a different instance of MEACC2, provided the system can afford the area and energy, the scaling is perfect up to the full tiles, but partial tiles only offer some fraction of speedup instead of full speedup. The reference frame is taken as the first frame in the video stream, and then the remaining frames are processed. Therefore, the higher framerate videos: Fourpeople, Johnny, and Kristen and Sara, have less overall movement per frame than the lower framerate videos: BasketballDrill, BQMall, and Flowervase.

Table 8.4: Pattern performance on BasketballDrill 832x480, 30 frames

pattern	mae	avg pts	mae pct	pts pct
cross	19.11	16.43	139.51%	0.00%
diamond	10.35	28.65	29.71%	74.42%
hybrid	10.83	28.01	35.76%	70.47%
hex aaa	14.95	20.36	87.37%	23.95%
hex bbb	15.22	20.20	90.71%	22.93%
hex aba	14.88	20.40	86.48%	24.18%
circular	7.98	38.02	0.00%	131.42%

Table 8.5: Pattern performance on BQMall 832x480, 30 frames

pattern	mae	avg pts	mae pct	pts pct
cross	17.15	19.41	127.54%	0.00%
diamond	9.47	33.86	25.61%	74.46%
hybrid	10.17	32.45	34.91%	67.18%
hex aaa	15.07	22.72	99.97%	17.06%
hex bbb	13.07	23.90	73.45%	23.11%
hex aba	14.13	23.33	87.52%	20.18%
circular	7.54	44.95	0.00%	131.56%

Table 8.6: Pattern performance on Flowervase 832x480, 30 frames

pattern	mae	avg pts	mae pct	pts pct
cross	3.10	15.15	134.47%	0.00%
diamond	1.60	24.45	21.03%	61.38%
hybrid	1.81	23.30	37.00%	53.82%
hex aaa	3.46	17.17	161.83%	13.33%
hex bbb	2.65	17.90	100.60%	18.17%
hex aba	3.18	17.79	140.17%	17.45%
circular	1.32	31.82	0.00%	110.03%

Across all 6 video streams, the circular pattern delivers the smallest MAE, while checking up to 139% more points per block. This reflects the predicted tradeoff, a smaller SAD match is found, but at the cost of more compute. The smaller SAD match has a

Table 8.7: Pattern performance on FourPeople 1280x720, 60 frames

pattern	mae	avg pts	mae pct	pts pct
cross	5.75	16.62	137.03%	0.00%
diamond	3.21	29.43	32.33%	77.02%
hybrid	3.31	28.77	36.36%	73.05%
hex aaa	4.69	20.47	93.07%	23.14%
hex bbb	4.62	20.63	90.43%	24.12%
hex aba	4.68	20.48	92.73%	23.20%
circular	2.43	38.95	0.00%	134.29%

Table 8.8: Pattern performance on Johnny 1280x720, 60 frames

pattern	mae	avg pts	mae pct	pts pct
cross	4.12	18.66	148.86%	0.00%
diamond	2.32	32.18	40.05%	72.45%
hybrid	2.45	30.97	47.85%	65.97%
hex aaa	3.35	22.79	102.20%	22.12%
hex bbb	3.28	23.32	97.67%	24.99%
hex aba	3.35	22.80	102.04%	22.20%
circular	1.66	44.60	0.00%	139.00%

Table 8.9: Pattern performance on Kristen and Sara 1280x720, 60 frames

pattern	mae	avg pts	mae pct	pts pct
cross	3.84	18.83	149.48%	0.00%
diamond	2.17	32.14	40.81%	70.67%
hybrid	2.30	30.83	49.60%	63.68%
hex aaa	3.14	22.79	103.77%	21.02%
hex bbb	3.06	23.43	98.88%	24.40%
hex aba	3.14	22.79	103.87%	21.01%
circular	1.54	44.88	0.00%	138.29%

MAE of approximately half that of previously introduced patterns. Therefore, the circular pattern can be used in applications whose video throughput requirements fall short of the capacity of MEACC2, in order to produce a smaller compression while still making full use of all the hardware already implemented in a fixed video-coding system, or simple patterns can be used to maximize pixel throughput, at a quantifiable cost to the MEA.

8.3 Performance Prediction

A performance prediction is made from simulation, based on the cost functions built into the model, as well as scaling factors derived from previous work on the AsAP platform.

8.3.1 From Cost Function to Performance Prediction

The cost function is used to capture a predicted cycle count during simulation and is divided into two parts, the cost associated with moving the reference memory frame, and the cost to perform the necessary computes. Compute is normally a raw number of SAD pairs computed, so an 4x4 block SAD would count for a cost of 16. The performance simulations were run with a block size of 8x8, since when considering the compute pipeline, the 8x8 block was the least efficiently used by the pipeline. This means that the cost function would be scaled by 64/16, or 4, to account for the number of cycles the datapath actually takes to compute the required SAD block.

This separation of the cost function also allows us to evaluate the overall split in cycles spent between pixel movement and SAD compute. In Table 8.10, the hybrid diamond-diamond-cross search split 30%/70% compute and move. This effect was similarly constant across all tested patterns, though the more points in a pattern the more the breakdown favored the compute side. At the most extreme, a circular pattern spends about 45% of the cost doing compute.

8.3.2 Performance Prediction Across Video Streams

Across 6 different video streams, the hybrid search gives a good balance between search time (performance) and final distortion (MAE). Nominal performance can be estimated by taking the sum of the two cost-function components and translating them to expected cycles. A more realistic estimation of the final performance is gained by scaling the nominal value by 0.3. This scaling factor is derived from the work of Landge, when she developed the accelerator [83], and Xiao, Le, and Baas when they developed an encoder using Landge's accelerator. The pixel throughput achieved by the overall system was only about 30% what Landge projected. A similar process takes place to build a workable encoder from MEACC2 so it should be fair to use a similar scaling factor to translate nominal throughput into actual expected performance. Xiao, Le, and Baas' work also provided the expected power number, scaling a $1.1W$ system at 400 MHz, to a $1.375W$ system at 500 MHz, and a $2.75W$ system at 1 GHz, which should be workable assumptions when fabricating in $65nm$. The power numbers, along with expected performance for both operating frequencies is used to project our performance and efficiency against other designs in Table 4.4 and Table 4.5.

Table 8.10: Hybrid search performance from simulation

	Work Breakout		Throughput (FPS)			Throughput (Mpix/s)		
Video	Comp%	MV%	Nom.	1 GHz	0.5 GHz	Nom.	1 GHz	0.5 GHz
BBall	32.92%	66.88%	471.01	141.30	70.65	188.10	56.43	28.22
BQMall	30.89%	68.95%	381.37	114.41	57.20	152.30	45.69	22.85
Flower	29.41%	70.38%	505.68	151.70	75.85	201.95	60.58	30.29
4 Ppl.	35.40%	64.51%	213.65	64.09	32.05	196.90	59.07	29.53
Johnny	32.74%	67.19%	183.52	55.05	27.53	169.13	50.74	25.37
K & S	34.15%	65.77%	192.35	57.71	28.85	177.27	53.18	26.59
480p	31.08%	68.74%	452.68	135.81	67.90	180.78	54.24	27.12
720p	34.10%	65.82%	196.50	58.95	29.48	181.10	54.33	27.16

8.3.3 Performance Scalability

One of the advantages of building MEACC2 to tile sizes is that multiple instances of MEACC2 can work in parallel. Each image stream is divided into 256x256 tiles, and each tile can be processed separately. For an 832x480 image, the partitioning fills 3 tiles completely, and 5 partial tiles. Since our simulations was run in series for each tile, this means that the work can be sped up at least 3 times, as 3 tiles can be kept at full utilization, while partial tiles have less utilization. Similarly, for a 1280x720 stream, there are 8 full tiles and 7 partial tiles, resulting in, at minimum, an 8x speedup. This additional silicon area is not free, especially in power and memory bandwidth terms, but if a system calls for maximum throughput, the option exists. Table 8.11 show the top line performance increase from a conservative scaling with tiles.

Table 8.11: Hybrid search performance with tiling scalability

	Tiles		Throughput (FPS)			Throughput (Mpix/s)		
Video	Full	Partial	Nom.	1 GHz	0.5 GHz	Nom.	1 GHz	0.5 GHz
BBall	3.00	5.00	1413.02	423.91	211.95	564.31	169.29	84.65
BQMall	3.00	5.00	1144.10	343.23	171.61	456.91	137.07	68.54
Flower	3.00	5.00	1517.04	455.11	227.56	605.84	181.75	90.88
4 Ppl.	8.00	7.00	4045.43	1213.63	606.81	1615.58	484.68	242.34
Johnny	8.00	7.00	1709.17	512.75	256.38	1575.17	472.55	236.28
K & S	8.00	7.00	1468.13	440.44	220.22	1353.03	405.91	202.95
480p	3.00	5.00	1358.05	407.42	203.71	542.35	162.71	81.35
720p	8.00	7.00	2407.58	722.27	361.14	1514.59	454.38	227.19

Chapter 9

Conclusions

9.1 Contributions

I have designed and implemented an new motion estimation engine, MEACC2, for a future AsAP platform, verified that it functions correctly, and projected it to have a worst-case throughput on par with ASIC designs, 2x the pixel/joule efficiency of the previous accelerator, and capable of 3.5x the pixel throughput, sufficient for real-time processing in 720p in the worst case at 110 FPS. Additionally, an extension on the full-search algorithm was proposed, smart-full-search to save memory bandwidth for minimal additional hardware, and a performance analysis of a novel 12-point center-biased search pattern was performed, where the pattern was found to improve MAE of a search by 2x, in exchange for searching 120% more points.

9.2 Non-Video Compression Applications

The primary focus of the research was in the development of a block for video compression purposes, but some of the circuit could be adapted for other uses.

9.2.1 Pattern Matching

A possible uses case is for matching patterns in a database, using the pixel array as a database of patterns. Run a decimated full search, based on the size of the patterns that have been stored. Very fast if used with proper thresholding and a slice of database small enough to fit in all of Reference Memory.

9.2.2 Motion Stabilization

Motion estimation for compression produces motion vectors which give a good sense of how objects in the image have moved over the sampling period. A system which took a set of blocks and computed their motion could make use of the generated vectors to estimate the overall system movement. This estimate could be fed into a stabilization system to stabilize the camera or platform, or could be applied to a video stream encoding as additional input, so that the video itself appears stable.

9.2.3 Burst Memory

Since there are instructions to allow the retrieval of pixel data that has been stored in either active or reference frame memory, it is possible to make use of the frame memory as a general purpose memory. If it is being used that way, it should be noted that the memory reads have a native locality to them, since pixels are produced in a 4x4 block. However, this locality would not necessarily be correlated unless the target application made an effort to handle the rotation of memory words. If such a rotation was desired in operation, it should be possible to use the MEACC2 memory to perform that operation.

9.3 Future Research

The 30/70 breakdown between pixel compute and pixel movement implies that there is additional performance to be found in executing more than one search in parallel, and using those parallel searches to fill the compute pipeline while waiting for pixel information to return. It also implies that the information bottleneck in these designs continues to be the memory system. The beginnings of an interesting solution to this problem might tile out multiple instances of the control modules, to conduct multiple searches which each request access to the datapath. An overall controller would decide when to move the memory, probably when there were no more points to be checked. This would allow the device to unroll the search patterns, similar to loop unrolling. A greedy search algorithm, which computes all the different divisions of a CTU, while attempting to move the memory as infrequently as possible, could help bridge the gap between these serial pattern processors and the 2d systolic arrays which compute everything in parallel.

There is also, of course, the goal of building an actual encoder around the bones provided by the MEACC2. Video encoding, in a low power and configurable context, remains a wide-open domain for novel solutions, and the AsAP platform, with its scalable mesh and the MEACC2 with its scalable design for tiles, seem ideally positioned to create a high performance, low power, real-time HEVC system.

Chapter 10

Glossary

ACT Mem. Active Frame Memory. In block motion algorithms, the memory being repeatedly operated on against a previous reference frame. Sometimes referred to as the current frame, or current pixels.

AMP Asymmetric Motion Prediction

AMVP Asymmetric Motion Vector Prediction

AsAP Asynchronous Array of Processors, the UC Davis VCL fine-grain many-core processing platform, originally for DSP and the demonstration platform for MEACC2.

ASIC Application Specific Integrated Circuit (IC), a circuit designed for a specific application.

ASIP Application Specific Instruction Processor, and processor whose instruction set is design for a specific application.

B-Frame Bidirectionally-predictive-coded Frames. These frames are never ref. frames, and use information from both temporal directions. They tend to compress more than P frames, all else being equal.

BMA Block Motion Algorithm. An algorithm primarily concerned with estimating the motion of blocks of pixels, not the motion of individual pixels.

CABAC Context-adaptive binary arithmetic coding [4]

CAVLC Context-adaptive variable length coding [5]

Center Biased Patterns BMAs which have a bias towards the center of the search area.

Chroma Color information is referred to as Chroma.

CPU Central Processing Unit

CSA Carry Save Adder, shorthand for a type of adder architecture making use of various kinds of compressors to "'save"' the carry bit of an addition for a final compression step, while the rest of the addition is done in parallel, independent of carrys from further down the adder chain [88].

CTB Coding Tree Block, a block of pixels which will be processed together.

CTU Coding Tree Unit, analogous to the macroblocks of H.264, these are groups of CTBs which represent the same frame area, but can be processed separately as they contain separate Luma/Chroma information.

CUDA NVidia's (a GPU manufacturer) proprietary GPGPU compute language for its GPUs

DRAM Dynamic Random-Access Memory, a memory which requires periodic refreshes to maintain its value, but more dense than SRAM. Typically used when large amounts of data storage are required for an application.

DSP Digital Signal Processing

FFT Fast Fourier transform, a mathematical operation frequently used in digital signal processing (DSP) applications.

FIFO First In First Out, describing the order in which messages are passed through the interface

FPGA Field Programmable Gate Array, and ASIC which can be configured to emulate different types of hardware using a HDL.

FPS Frames Per Second, a common measure of throughput in video encode and decode applications

FSM Finite State Machine

Full Search BMA which is not center biased, and checks every possible block location.

GoP Group of Pictures, a collection of I, P, and B frames which make up a subset of a video stream to be encoded.

GPGPU General Purpose GPU, used to describe compute applications which us a GPU for general purpose processing.

GPU Graphics processing unit, typically a highly parallel SIMD architected ASIC.

H.264/AVC Advanced Video Compression, a coding standard, introduced in 2003 for video compression and playback standardization.

H.265/HEVC High Efficiency Video Coding, a coding standard, introduced in 2012, to replace H.264/AVC with new features and targeted at reducing encoded video size by 50%.

HDL Hardware description language such as Verilog or VHDL. Used to describe hardware as the first step in both ASIC and FPGA design flows.

HFSM Hierarchical Finite State Machine. A state machine which can be decomposed into a set of constituent FSMs [89].

I-Frame Intra-coded Frames is a compressed version of a raw frame containing information from only a single frame, and can therefore be decoded independently of its neighbors.

Inter-Frame Between Frames

Intra-Frame Within Frames

Luma Light intensity information is referred to as Luma, a picture can be decomposed into its Luma and Chroma parts.

MAE Mean Absolute Error, a figure of merit when comparing the distortion between two blocks of pixels.

MEACC2 The 2nd generation Motion Estimation Accelerator

MV Motion Vector, a pair of (dX, dY) coordinates denoting the offset of the best match of a block of pixels from its starting location.

OpenCL Open source C-based framework for parallel computing, sometimes used as a substitute for CUDA, or to achieve GPGPU with non-NVidia GPUs.

P-Frame Predictive-coded Frame, are encoded using data from previous I and P frames allowing for more efficient compression.

Pattern Search A search which follows a particular pattern, rather than checking all the possibilities as in Full Search.

PB Prediction Block, the block of pixels which will be evaluated and compressed together.

PE Processing Elements

PMV Predicted Motion Vector

REF Mem. Reference Frame Memory. In block motion algorithms, the memory from a previous frame.

ROM Read Only Memory, used to store values which are fixed at design time.

SAD Sum of Absolute Differences, a figure of merit when comparing the distortion between two blocks of pixels.

Search Area The area in a frame which will be searched for the best-match candidate.

SIMD Single Instruction Multiple Data, describing compute architectures where a single instruction operates on multiple data simultaneously.

SCM Standard Cell Memory. Memory built primarily with latches or registers.

SRAM Static Random-Access Memory, on chip memory used to store data during operation.

Tile The unit of division in H.265 for parallel processing. Individual tiles can be processed independently of each other with a final merge step to bring the whole encoded image together.

TSS Three Step Search, a type of center-biased search pattern with a Diamond-Diamond-Cross pattern progression.

VBSME Variable Block Size Motion Estimation, describes encodings which have macroblocks (if H.264) or CTBs (if H.265) of multiple sizes.

Appendix A

Matlab Model Code

A.1 motion_estimation_engine.m

```
classdef motion_estimation_engine_model < handle
    % A bit accurate model of Michael's Motion Estimation Engine.
    properties
        % Transcript Values
        CREATE_TRANSCRIPT = 0;
        OFILE = '';
        IFILE = '';
    % Setup Matrix: pmvdx, pmvdy, blkid, thresh, ax, ay,
    %               oreg, imx, imy, decx, decy, pat_addr
        SU_MAT = [];
    % Pixel Request Matrix: X, Y, W, H
        PR_MAT = [];
    % Search Result Matrix: X, Y, SAD
        RS_MAT = [];
    % Matrix of Points Checked, X/Y
        PT_MAT = [];
        % Physical Characteristics
        REF_MEM_SIZEXY = [64, 64];
        ACT_MEM_SIZEXY = [64, 64];
        PAT_MEM_SIZE = 32;
```

```
% Memories
REF_MEM = [];
ACT_MEM = [];
PAT_MEM_OFFSETS = [];
PAT_MEM_VLDS = [];
PAT_MEM_JMPADDR = [];
% Frames (Virtual Constructs, not present in actual Unit)
FRAME_SIZEXY = [256, 256];
REF_FRAME = [];
ACT_FRAME = [];
PAT_MEM_FORMAT = [];
% Flags (Virtual Constructs)
STEP_EN = 0;
PLOT_EN = 0;
REPORT_EN = 0;
PLOT_NEW_EN = 0;
ENABLE_LAST_CHECK_REG = 0;
% Logs
LOG_NUM_MOVES = uint64(0);
LOG_MOVE_COST = uint64(0);
LOG_SAD_COST = uint64(0);
LOG_NUM_PTS_CHECKED = uint64(0);
%Externally Visible Registers
BURST_REF_ORIGIN_X_REG = 0;
BURST_REF_ORIGIN_Y_REG = 0;
BURST_REF_HEIGHT_REG = 0;
BURST_REF_WIDTH_REG = 0;
PAT_WRITE_ADDR_REG = 0;
PMV_DX_REG = 0;
PMV_DY_REG = 0;
BLK_ID_REG = 0;
THRESH_TOP_REG = 0;
THRESH_BOT_REG = 0;
ACT_PT_X_REG = 0;
ACT_PT_Y_REG = 0;
REF_PT_X_REG = 0;
```

```
        REF_PT_Y_REG = 0;
        IM_SZ_X_REG = 256;
        IM_SZ_Y_REG = 256;
        MV_X_REG = 0;
        MV_Y_REG = 0;
    end % properties

    % Internal Properties / Registers.
    properties (SetAccess = private)
        BLK_WIDTH = 0;
        BLK_HEIGHT = 0;
        SAD_THRESHOLD = 0;
        REF_ORIGIN_X_REG = 1; % index starts at 1 in matlab
        REF_ORIGIN_Y_REG = 1; % index starts at 1 in matlab
        MEM_MOVED_REG = 0;
        CURR_VLDS_REG = '0000000000000000';
        SEARCH_CENTER_X_REG = 0;
        SEARCH_CENTER_Y_REG = 0;
        NEW_SEARCH_CENTER_X_REG = 0;
        NEW_SEARCH_CENTER_Y_REG = 0;
        BEST_SAD_VAL_REG = 1048576; % max value of 20 unsigned bits + 1
        SEARCH_BASE_PAT_ADDR = 0;
        NEW_SEARCH_BASE_PAT_ADDR = 0;
        PAT_SEARCH_FINISHED = 0;
        LAST_CHECK_X = 0;
        LAST_CHECK_Y = 0;
    end

    methods
    % Need a method for each command word and a constructor.
    % Also a reset function.

    function enable_transcription(obj, name)
        obj.CREATE_TRANSCRIPT = 1;
        obj.OFILE = ['trans_' name '_outputs'];
        obj.IFILE = ['trans_' name '_inputs'];
```

```
end

function obj = motion_estimation_engine_model(ref_szxy, act_szxy, ...
        patsz, framesxy)
obj.REF_MEM_SIZEXY =  ref_szxy;
obj.REF_MEM = zeros(ref_szxy(1), ref_szxy(2));
obj.ACT_MEM_SIZEXY =  act_szxy;
obj.ACT_MEM = zeros(act_szxy(1), act_szxy(2));
obj.PAT_MEM_SIZE =  patsz;
obj.PAT_MEM_OFFSETS = zeros(patsz, 2);
    obj.PAT_MEM_JMPADDR = zeros(patsz, 1);
    obj.PAT_MEM_VLDS = repmat('0000000000000000', patsz, 1);
    obj.PAT_MEM_FORMAT = repmat('.g', patsz, 1);
    obj.FRAME_SIZEXY = framesxy;
    obj.IM_SZ_X_REG = framesxy(2);
    obj.IM_SZ_Y_REG = framesxy(1);
    obj.REF_FRAME = zeros(framesxy(1), framesxy(2));
    obj.ACT_FRAME = zeros(framesxy(1), framesxy(2));
end % end motion_estimation_engine_model (constructor)

function set_burst_ref_x(obj, value)
    obj.BURST_REF_ORIGIN_X_REG = value;
end % end set_burst_ref_x

function set_burst_ref_y(obj, value)
    obj.BURST_REF_ORIGIN_Y_REG = value;
end % end set_burst_ref_y

function set_burst_ref_height(obj, value)
    obj.BURST_REF_HEIGHT_REG = value;
end % end set_burst_ref_height

function set_burst_ref_width(obj, value)
    obj.BURST_REF_WIDTH_REG = value;
end % end set_burst_ref_width
```

```
function set_pmv_x(obj, value)
    obj.PMV_DX_REG = value;
end % end set_pmv_x

function set_pmv_y(obj, value)
    obj.PMV_DY_REG = value;
end % end set_pmv_y



% Set Block ID will also set the internal values blk_width and
% blk_height.  This mimics the lookup table the unit uses.
function set_blkid(obj, value)
    obj.BLK_ID_REG = value;
    switch value
        case 0
            obj.BLK_WIDTH = 64;
            obj.BLK_HEIGHT = 64;
        case 1
            obj.BLK_WIDTH = 32;
            obj.BLK_HEIGHT = 64;
        case 2
            obj.BLK_WIDTH = 64;
            obj.BLK_HEIGHT = 32;
        case 3
            obj.BLK_WIDTH = 32;
            obj.BLK_HEIGHT = 32;
        case 4
            obj.BLK_WIDTH = 16;
            obj.BLK_HEIGHT = 32;
        case 5
            obj.BLK_WIDTH = 32;
            obj.BLK_HEIGHT = 16;
        case 6
            obj.BLK_WIDTH = 16;
            obj.BLK_HEIGHT = 16;
        case 7
```

```
                obj.BLK_WIDTH = 8;
                obj.BLK_HEIGHT = 16;
            case 8
                obj.BLK_WIDTH = 16;
                obj.BLK_HEIGHT = 8;
            case 9
                obj.BLK_WIDTH = 8;
                obj.BLK_HEIGHT = 8;
            case 10
                obj.BLK_WIDTH = 4;
                obj.BLK_HEIGHT = 8;
            case 11
                obj.BLK_WIDTH = 8;
                obj.BLK_HEIGHT = 4;
            otherwise
                obj.BLK_WIDTH = 4;
                obj.BLK_HEIGHT = 4;
        end
    end % end set_blkid

    function set_thresh_top(obj, value)
        obj.THRESH_TOP_REG = value;
        obj.SAD_THRESHOLD = value * 2^10 + obj.THRESH_BOT_REG;
    end % end set_thresh_top

    function set_thresh_bot(obj, value)
        obj.THRESH_BOT_REG = value;
        obj.SAD_THRESHOLD = obj.THRESH_TOP_REG * 2^10 + value;
    end % end set_thresh_top

    function set_act_pt_x(obj, value)
        obj.ACT_PT_X_REG = value;
    end % end set_act_pt_x

    function set_act_pt_y(obj, value)
        obj.ACT_PT_Y_REG = value;
```

```
end % end set_act_pt_y

function set_ref_pt_x(obj, value)
    obj.REF_PT_X_REG = value;
end % end set_ref_pt_x

function set_ref_pt_y(obj, value)
    obj.REF_PT_Y_REG = value;
end % end set_ref_pt_y

function log_setup(obj, decx, decy, pat_base, isFS)
    % Setup Matrix: pmvdx, pmvdy, blkid, thresh, ax, ay, oreg, imx,
    % imy, decx, decy, pat_base, isFS
    SU = [obj.PMV_DX_REG, obj.PMV_DY_REG, obj.BLK_ID_REG, ...
    obj.SAD_THRESHOLD, obj.ACT_PT_X_REG, obj.ACT_PT_Y_REG, ...
    obj.REF_PT_X_REG, obj.REF_PT_Y_REG, 0, ...
    obj.IM_SZ_X_REG, obj.IM_SZ_Y_REG, decx, decy, ...
    pat_base, isFS];
    obj.SU_MAT = [obj.SU_MAT; SU];
end % end log_setup

function log_result(obj)
    % Search Result Matrix: X, Y, SAD
    RS = [obj.SEARCH_CENTER_X_REG, obj.SEARCH_CENTER_Y_REG, ...
    obj.BEST_SAD_VAL_REG];
    obj.RS_MAT = [obj.RS_MAT; RS];
    % put a 0 pixel request into the log (which is illegal) to show end
    % of log.
    obj.PR_MAT = [obj.PR_MAT; [0 0 0 0]];
end

function [] = run_simple_full_search(obj, decx, decy)
    % Given a decimation count, use the blk_size and image_sizes to run
    % through all possible points in the search frame.  Decimation in
    % this case means the skip value in x and y directions.  A standard
    % full search therefore has decimation of 1.
```

```
        obj.LOG_NUM_MOVES = 0;
        obj.LOG_MOVE_COST = 0;
        obj.LOG_SAD_COST = 0;
        obj.LOG_NUM_PTS_CHECKED = 0;
        obj.BEST_SAD_VAL_REG = 1048576;

        if(obj.CREATE_TRANSCRIPT)
            obj.log_setup(decx, decy, 0, 1);
        end

        % check first point
        sad = obj.sad_comp_memcheck(1, 1, obj.ACT_PT_X_REG, obj.ACT_PT_Y_REG);
        obj.BEST_SAD_VAL_REG = sad;
        obj.SEARCH_CENTER_X_REG = 1;
        obj.SEARCH_CENTER_Y_REG = 1;

        % check the rest of the points
        for x = 1:decx:obj.IM_SZ_X_REG - obj.BLK_WIDTH
            for y = 1:decy:obj.IM_SZ_Y_REG - obj.BLK_HEIGHT
                sad = obj.sad_comp_memcheck(x, y, ...
            obj.ACT_PT_X_REG, obj.ACT_PT_Y_REG);
                if sad < obj.BEST_SAD_VAL_REG
                    obj.BEST_SAD_VAL_REG = sad;
                    obj.SEARCH_CENTER_X_REG = x;
                    obj.SEARCH_CENTER_Y_REG = y;
                end
            end
        end

        if(obj.CREATE_TRANSCRIPT)
            obj.log_result();
        end

    end

    function [] = run_smart_full_search(obj, decx, decy)
```

```
obj.LOG_NUM_MOVES = 0;
obj.LOG_MOVE_COST = 0;
obj.LOG_SAD_COST = 0;
obj.LOG_NUM_PTS_CHECKED = 0;
obj.BEST_SAD_VAL_REG = 1048576;

if(obj.CREATE_TRANSCRIPT)
    obj.log_setup(decx, decy, 0, 1);
end

% check first point
sad = obj.sad_comp_memcheck(1, 1, ...
    obj.ACT_PT_X_REG, obj.ACT_PT_Y_REG);
obj.BEST_SAD_VAL_REG = sad;
obj.SEARCH_CENTER_X_REG = 1;
obj.SEARCH_CENTER_Y_REG = 1;

% check the rest of the points, sector by sector.
for x = 1:obj.REF_MEM_SIZEXY(1)-obj.BLK_WIDTH+1: ...
obj.IM_SZ_X_REG - obj.BLK_WIDTH
    for y = 1:obj.REF_MEM_SIZEXY(2)-obj.BLK_HEIGHT+1: ...
obj.IM_SZ_Y_REG - obj.BLK_HEIGHT
        for dx = 0:decx:obj.REF_MEM_SIZEXY(1)-obj.BLK_WIDTH
            for dy = 0:decy:obj.REF_MEM_SIZEXY(2) - obj.BLK_HEIGHT
                checkx = x + dx;
                checky = y + dy;
                %fprintf('(%i, %i)\n', checkx, checky)
                if and((checkx < ...
                    obj.IM_SZ_X_REG-(obj.BLK_WIDTH-1)), ...
        (checky < obj.IM_SZ_Y_REG-(obj.BLK_HEIGHT-1)))
                    %fprintf('(%i, %i)\n', checkx, checky)
                    sad = obj.sad_comp_memcheck(x + dx, y + dy, ...
        obj.ACT_PT_X_REG, obj.ACT_PT_Y_REG);
                    if sad < obj.BEST_SAD_VAL_REG
                        obj.BEST_SAD_VAL_REG = sad;
                        obj.SEARCH_CENTER_X_REG = x + dx;
```

```
                            obj.SEARCH_CENTER_Y_REG = y + dy;
                        end
                    end
                end
            end
        end
    end

    if(obj.CREATE_TRANSCRIPT)
        obj.log_result();
    end

end

function [] = run_search(obj, pat_addr)

% Run Search
% Do a 3-stage diamond search.
%
% NOTE: Open and hold the picture first if PLOT_EN is 1.

% Reset the logging variables.
obj.LOG_NUM_MOVES = 0;
obj.LOG_MOVE_COST = 0;
obj.LOG_SAD_COST = 0;
obj.LOG_NUM_PTS_CHECKED = 0;
obj.BEST_SAD_VAL_REG = 1048576; % max value of 20 unsigned bits + 1
obj.PAT_SEARCH_FINISHED = 0;

% Set base of search pattern
obj.SEARCH_BASE_PAT_ADDR = pat_addr;

if(obj.CREATE_TRANSCRIPT)
        obj.log_setup(0,0, pat_addr, 0);
end
```

```
% Check Center Point, save SAD value to register, save coordinates
% to register setup the current vlds, and Plot.
[sad] = obj.sad_comp_memcheck(obj.REF_PT_X_REG, obj.REF_PT_Y_REG, ...
obj.ACT_PT_X_REG, obj.ACT_PT_Y_REG);
if obj.ENABLE_LAST_CHECK_REG
    obj.LAST_CHECK_X = obj.REF_PT_X_REG;
    obj.LAST_CHECK_Y = obj.REF_PT_Y_REG;
end
obj.SEARCH_CENTER_X_REG = obj.REF_PT_X_REG;
obj.SEARCH_CENTER_Y_REG = obj.REF_PT_Y_REG;
obj.BEST_SAD_VAL_REG = sad;
%obj.SEARCH_BASE_PAT_ADDR = 0;
obj.CURR_VLDS_REG = obj.PAT_MEM_VLDS(obj.SEARCH_BASE_PAT_ADDR+1,:)-'0';
obj.CURR_VLDS_REG;

if obj.PLOT_EN
    plot(obj.SEARCH_CENTER_X_REG, obj.SEARCH_CENTER_Y_REG, 'r.');
end
if obj.STEP_EN
    waitforbuttonpress;
end

% If the center point doesn't meet threshold, continue.
if obj.BEST_SAD_VAL_REG > obj.SAD_THRESHOLD
    while obj.PAT_SEARCH_FINISHED == 0
        old_center_x = obj.SEARCH_CENTER_X_REG;
        old_center_y = obj.SEARCH_CENTER_Y_REG;
        obj.check_pattern();

        if obj.PLOT_NEW_EN
            imshow(obj.REF_FRAME)
        end

        if obj.PLOT_EN
            plot(obj.SEARCH_CENTER_X_REG, obj.SEARCH_CENTER_Y_REG, ...
                    'r.');
                line([obj.SEARCH_CENTER_X_REG, old_center_x], ...
            [obj.SEARCH_CENTER_Y_REG, old_center_y]);
            end
            if obj.STEP_EN
                waitforbuttonpress;
            end
            if obj.BEST_SAD_VAL_REG <= obj.SAD_THRESHOLD
                break;
            end

        end
    end

    obj.MV_X_REG = obj.SEARCH_CENTER_X_REG - obj.ACT_PT_X_REG;
    obj.MV_Y_REG = obj.SEARCH_CENTER_Y_REG - obj.ACT_PT_Y_REG;

    if obj.REPORT_EN
        fprintf('Block Search Completed for block: [%i, %i]\n', ...
        obj.ACT_PT_X_REG, obj.ACT_PT_Y_REG);
    end

    if(obj.CREATE_TRANSCRIPT)
        obj.log_result();
    end

    end



    end % public methods

methods %(Access = protected)
    function [] = check_pattern(obj)
    % CHECK_PATTERN
    % Given frames, pattern center, the top left corner of the block we're
    % matching, the dimensions of the block, the current best SAD,
    % which points are vld (a vector or 1's and 0s, where 0 means invalid),
    % and a pattern (a vector of (x,y) offsets corresponding to vlds).
    %
    % Check each scan point and return the next center, next SAD,
    % next set of vlds, and whether or not the center moved (1 if it moved).
    %
    % Do saturation checking to make sure the SAD requests stay within the
    % boundary of the image.

    %num_scan_points = length(scan_points);
    % This got more complicated with the movement to a register model.
    % Now we want to loop over pattern memory - not just the simple model
    % of providing each pattern separately.
    last_vld_pt = 0;

    for position = 1:length(obj.CURR_VLDS_REG)
        if obj.CURR_VLDS_REG(position) == 1
            last_vld_pt = position;
        end
    end

    if last_vld_pt == 0
        obj.PAT_SEARCH_FINISHED = 1;
        return;
    end

    center_moved = 0;
    pt_chosen = obj.SEARCH_BASE_PAT_ADDR + 1;

    for j = 1:last_vld_pt
        if obj.CURR_VLDS_REG(j) == 1 ...
        && obj.BEST_SAD_VAL_REG > obj.SAD_THRESHOLD
            px = obj.SEARCH_CENTER_X_REG ...
        + obj.PAT_MEM_OFFSETS(pt_chosen + j, 1);
            py = obj.SEARCH_CENTER_Y_REG ...
        + obj.PAT_MEM_OFFSETS(pt_chosen + j, 2);
```

```
        % saturate if we are about to step outside the image
        % boundary.  Remember that matlab indexes from 1.
        if px > obj.IM_SZ_X_REG - obj.BLK_WIDTH;
            px = obj.IM_SZ_X_REG - obj.BLK_WIDTH;
        end
        if px < 1
            px = 1;
        end
        if py > obj.IM_SZ_Y_REG - obj.BLK_HEIGHT
            py = obj.IM_SZ_Y_REG - obj.BLK_HEIGHT;
        end
        if py < 1
            py = 1;
        end
        % end saturation checks.
        if obj.ENABLE_LAST_CHECK_REG
            % if skip register enabled, check to see if this is repeat
            % work, if it is, skip it, else update the register.
            if (obj.LAST_CHECK_X == px && obj.LAST_CHECK_Y == py)
                continue;
            else
                obj.LAST_CHECK_X = px;
                obj.LAST_CHECK_Y = py;
            end
        end

        if obj.PLOT_EN
            plot(px, py, obj.PAT_MEM_FORMAT(pt_chosen + j, :));
        end
        if obj.STEP_EN
            waitforbuttonpress();
        end

        point_sad = obj.sad_comp_memcheck(px, py, ...
obj.ACT_PT_X_REG, obj.ACT_PT_Y_REG);
```

```
            %if obj.MEM_MOVED_REG
            %    mem_pos_ever_moved = 1;
            %end

            if point_sad < obj.BEST_SAD_VAL_REG
                obj.BEST_SAD_VAL_REG = point_sad;
                obj.NEW_SEARCH_CENTER_X_REG = px;
                obj.NEW_SEARCH_CENTER_Y_REG = py;
                next_pt_chosen = j + obj.SEARCH_BASE_PAT_ADDR + 1;
                center_moved = 1;
            end
        end
    end

    %if pt_chosen == 0
    if center_moved ~= 0
        % if AR is a string of ones and zeros, then AR - '0' results in a
        % one dimensional array with the characters broken out into ones
        % and zeros and now index-able.
        % '11110000' -> [1,1,1,1,0,0,0,0]
        obj.CURR_VLDS_REG = obj.PAT_MEM_VLDS(next_pt_chosen,:)-'0';
        obj.SEARCH_BASE_PAT_ADDR = obj.PAT_MEM_JMPADDR(next_pt_chosen);
        obj.SEARCH_CENTER_X_REG = obj.NEW_SEARCH_CENTER_X_REG;
        obj.SEARCH_CENTER_Y_REG = obj.NEW_SEARCH_CENTER_Y_REG;
    else
        obj.CURR_VLDS_REG = ...
        obj.PAT_MEM_VLDS(obj.SEARCH_BASE_PAT_ADDR+1, :)-'0';
        obj.SEARCH_BASE_PAT_ADDR = ...
        obj.PAT_MEM_JMPADDR(obj.SEARCH_BASE_PAT_ADDR+1);
    end
    end

    function [sad_sum] = sad_comp_memcheck(obj, rx, ry, ax, ay)
    % SAD_COMP_MEMCHECK
    %
```

```
% Compute the sum absolute difference between two blocks.  This version
% has dynamic bounds checkings for memory and then calls SAD_COMP.

% Before doing the SAD, check to make sure all the memory we need is
% in-bound. If it is not, then move the memory's corner to bring
% it in bound.

obj.MEM_MOVED_REG = 0;
mem_pos = [obj.REF_ORIGIN_X_REG, obj.REF_ORIGIN_Y_REG];
new_mem_pos = mem_pos;

x_mem = obj.REF_ORIGIN_X_REG;
y_mem = obj.REF_ORIGIN_Y_REG;
w_mem = obj.REF_MEM_SIZEXY(1);
h_mem = obj.REF_MEM_SIZEXY(2);
w_blk = obj.BLK_WIDTH;
h_blk = obj.BLK_HEIGHT;

blk_oob_r = rx + w_blk > x_mem + w_mem;
blk_oob_l = rx < x_mem;
blk_oob_u = ry < y_mem;
blk_oob_d = ry + h_blk > y_mem + h_mem;

blk_oob_x = or(blk_oob_r, blk_oob_l);
blk_oob_y = or(blk_oob_u, blk_oob_d);
blk_oob = or(blk_oob_x, blk_oob_y);

if blk_oob
    obj.MEM_MOVED_REG = 1;
    if blk_oob_r
        new_mem_pos(1) = rx + obj.BLK_WIDTH - w_mem;
    end
    if blk_oob_l
        new_mem_pos(1) = rx;
    end
    if blk_oob_u
```

```
            new_mem_pos(2) = ry;
        end
        if blk_oob_d
            new_mem_pos(2) = ry + h_blk - h_mem;
        end
        mv_cost = movement_cost(new_mem_pos, mem_pos, [w_mem, h_mem]);
        obj.LOG_MOVE_COST = obj.LOG_MOVE_COST + mv_cost;
        if obj.REPORT_EN
            fprintf( ...
                'memory moved at SAD level, dx: %i, dy: %i cost: %i\n', ...
                new_mem_pos - mem_pos, mv_cost);
            movement_request(obj, new_mem_pos, mem_pos, [w_mem, h_mem]);
        end

        obj.REF_ORIGIN_X_REG = new_mem_pos(1);
        obj.REF_ORIGIN_Y_REG = new_mem_pos(2);

        obj.LOG_NUM_MOVES = obj.LOG_NUM_MOVES + 1;

        if obj.PLOT_EN
            rectangle('Position', [new_mem_pos, obj.REF_MEM_SIZEXY]);
        end
    end

    sad_sum = obj.sad_comp(rx, ry, ax, ay);
    num_sads = obj.BLK_HEIGHT * obj.BLK_WIDTH;
    obj.LOG_SAD_COST = obj.LOG_SAD_COST + num_sads;
    obj.LOG_NUM_PTS_CHECKED = obj.LOG_NUM_PTS_CHECKED + 1;
    if(obj.CREATE_TRANSCRIPT)
        obj.PT_MAT = [obj.PT_MAT; [rx, ry, sad_sum]];
    end

    end

    function sad_sum = sad_comp(obj, rx, ry, ax, ay)
    % SAD_COMP
```

```
% compute the sad between two block of given w_blk and h_blk in the
% given active and reference frames, with starting upper left hand
% corners being given by (rx, ry) and (ax, ay).
%
% WARNING: This function does no bound checking.
w_blk = obj.BLK_WIDTH;
h_blk = obj.BLK_HEIGHT;

%sad_matrix = zeros(w_blk, h_blk);

%for j = 0:h_blk-1,
%    for i = 0:w_blk-1,
%        refpix = int32(obj.REF_FRAME(j + ry, i + rx));
%        actpix = int32(obj.ACT_FRAME(j + ay, i + ax));
%        sad_matrix(j+1, i+1) = abs(refpix - actpix);
%    end
%end

% this is faster
rslice = obj.REF_FRAME(ry:ry+h_blk-1, rx:rx+w_blk-1);
aslice = obj.ACT_FRAME(ay:ay+h_blk-1, ax:ax+w_blk-1);
rslice = int32(rslice);
aslice = int32(aslice);
sad_matrix = abs(rslice - aslice);

% sum() works in one dimension at a time, so first sum each of the
% columns
% and then second sum to sum those sums to a final result.
% sad_sum = sum(sum(sad_matrix));
sad_sum = sum(sad_matrix(:)); % this is faster
end

function [cost] = movement_cost(new_mem_pos, old_mem_pos, mem_sizexy)
% MOVEMENT_COST
% Compute the movement cost for moving refmem to new_mem_pos.
% For now, cost is the cycle cost of bringing in all the pixels, no
% discounts from overlapping operations. Pixels are 8b and our comm
% width is 16b, hence cost = raw_cost / 2
% Built in assumption that x direction will move in multiples of two,
% btw.

delta = abs(new_mem_pos - old_mem_pos);
dx = min(mem_sizexy(1), delta(1));
dy = min(mem_sizexy(2), delta(2));

raw_cost = dx * mem_sizexy(1) + dy * mem_sizexy(2) - dx * dy;

cost = raw_cost / 2;

end

function [] = movement_request(obj, new_mem_pos, ...
            old_mem_pos, mem_sizexy)
% Movment Request
% What pixel requests will the Motion Estimation Unit send to the ASAP
% Controller?

% IMPORTANT: The motion estimation unit can only take pixel pairs, not
% single pixels in the X direction.  This means that the effective
% minimum width is 2, and if we're moving in the right/east direction
% we will need to adjust the X coordinate of the pixel requests.  Since
% the width correction is always done, we catch it at the bottom of the
% function.  Since the X request adjustment is done case-by-case, we
% need to set a flag.

delta = (new_mem_pos - old_mem_pos);
dx = delta(1);
dy = delta(2);
modify_address = 0;
%dx = min(mem_sizexy(1), delta(1))
%dy = min(mem_sizexy(2), delta(2))
```

```
% Consider all possible cases
% For Message we need: X, Y, W, H of the rectangle of pixels we want.
% two commands better than 3 commands.

% Case 1 - Up
if (dx == 0 && dy < 0)
    x1 = new_mem_pos(1);
    y1 = new_mem_pos(1);
    w1 = mem_sizexy(1);
    h1 = abs(dy);
    num_words = 1;
end
% Case 2 - Down
if (dx == 0 && dy > 0)
    x1 = old_mem_pos(1);
    y1 = old_mem_pos(2) + mem_sizexy(2);
    w1 = mem_sizexy(1);
    h1 = dy;
    num_words = 1;
end
% Case 3 - Left
if (dx < 0 && dy == 0)
    x1 = new_mem_pos(1);
    y1 = new_mem_pos(1);
    w1 = abs(dx);
    h1 = mem_sizexy(2);
    num_words = 1;
end
% Case 4 - Right
if (dx > 0 && dy == 0)
    x1 = old_mem_pos(1) + mem_sizexy(1);
    y1 = old_mem_pos(2);
    w1 = dx;
    h1 = mem_sizexy(2);
    num_words = 1;
    modify_address = 1;
```

```
end
% Case 5 - Up & Right
if (dx > 0 && dy < 0)
    x1 = new_mem_pos(1);
    y1 = new_mem_pos(2);
    w1 = mem_sizexy(1);
    h1 = abs(dy);
    x2 = old_mem_pos(1) + mem_sizexy(1);
    y2 = old_mem_pos(2);
    w2 = dx;
    h2 = mem_sizexy(2) - abs(dy);
    num_words = 2;
    modify_address = 1;
end
% Case 6 - Down & Right
if (dx > 0 && dy > 0)
    x1 = new_mem_pos(1);
    y1 = old_mem_pos(2) + mem_sizexy(2);
    w1 = mem_sizexy(1);
    h1 = dy;
    x2 = old_mem_pos(1) + mem_sizexy(1);
    y2 = old_mem_pos(2) + dy;
    w2 = dx;
    h2 = mem_sizexy(2) - dy;
    num_words = 2;
    modify_address = 1;
end
% Case 7 - Down & Left
if (dx < 0 && dy > 0)
    x1 = new_mem_pos(1);
    y1 = old_mem_pos(2) + mem_sizexy(2);
    w1 = mem_sizexy(1);
    h1 = dy;
    x2 = new_mem_pos(1);
    y2 = new_mem_pos(2);
    w2 = abs(dx);
```

```
        h2 = mem_sizexy(2) - dy;
        num_words = 2;
    end
    % Case 8 - Up & Left
    if (dx < 0 && dy < 0)
        x1 = new_mem_pos(1);
        y1 = new_mem_pos(2);
        w1 = mem_sizexy(1);
        h1 = abs(dy);
        x2 = new_mem_pos(1);
        y2 = new_mem_pos(2) + abs(dy);
        w2 = abs(dx);
        h2 = mem_sizexy(2) - abs(dy);
        num_words = 2;
    end

    % Case 9
    % If we move completely out of frame - then we only need one word with
    % the new position.
    if (abs(dx) >= mem_sizexy(1) || abs(dy) >= mem_sizexy(2))
        x1 = new_mem_pos(1);
        y1 = new_mem_pos(2);
        w1 = mem_sizexy(1);
        h1 = mem_sizexy(2);
        num_words = 1;
    end

    % Width Correction Code
    if w1 == 1
        w1 = 2;
        if modify_address == 1
            x1 = x1 - 1;
        end
    end

    if num_words > 1
```

```
        if w2 == 1;
            w2 = 2;
            if modify_address == 1
                x2 = x2 - 1;
            end
        end
    end

    fprintf('Pixel Request 1, X:%i, Y:%i, W:%i, H:%i\n', x1, y1, w1, h1);
    obj.PR_MAT = [obj.PR_MAT; [x1, y1, w1, h1]];
    if num_words > 1
        fprintf('Pixel Request 2, X:%i, Y:%i, W:%i, H:%i\n', ...
            x2, y2, w2, h2);
        obj.PR_MAT = [obj.PR_MAT; [x2, y2, w2, h2]];
    end

    end

end % protected methods


end % classdef
```

Appendix B

Matlab Instruction Generation Code

B.1 generate_test_from_model_run.m

```
function [] = generate_test_from_model_run(name, model)
%GENERATE_TEST_FROM_MODEL_RUN Create a Verilog Testbench Input/Output Pair
%from the logs of a model run with the given name.
%   When a search is run on the model with generate_trace == 1 (default is
%   0), it will log setup, pixel requests, and result.  This function
%   parses those traces into testvector files suitable for the vt file.
%   Also need to provide the reference and active IMAGES, and the PATTERN
%   MEMORY's 3 vectors (we don't care about the format vector now).
%
%   You know what, just give it the whole model.

SU = model.SU_MAT;
PR = model.PR_MAT;
RS = model.RS_MAT;
AIm = model.ACT_FRAME;
RIm = model.REF_FRAME;
PM_OFF = model.PAT_MEM_OFFSETS;
```

```
PM_VLD = model.PAT_MEM_VLDS;

PM_JMP = model.PAT_MEM_JMPADDR;


i_file_name = ['autogen_' name '_in.tv'];

o_file_name = ['autogen_' name '_out.tv'];


init_refmem = RIm(1:64, 1:64);

init_actmem = AIm(1:64, 1:64);


fprintf('Generating Test from given Traces & Image\n');

i_file = fopen(i_file_name, 'w');

o_file = fopen(o_file_name, 'w');


% load initial REF pixels

fprintf(i_file, test_input_gen('vld', 'set_burst_ref_x',  0));

fprintf(i_file, test_input_gen('vld', 'set_burst_ref_y',  0));

fprintf(i_file, test_input_gen('vld', 'set_burst_width',  63));

fprintf(i_file, test_input_gen('vld', 'set_burst_height', 63));

fprintf(i_file, test_input_gen('vld', 'write_burst_ref'));

fprintf(i_file, test_input_gen('vld', 'burst_pixel_write', init_refmem));


% load initial ACT pixels

fprintf(i_file, test_input_gen('vld', 'write_burst_act'));

fprintf(i_file, test_input_gen('vld', 'burst_pixel_write', init_actmem));


% load configurable part of PAT Memory

for x = 0:31

    r = x + 1; % Matlab indexes by 1

    vlds = fliplr(PM_VLD(r, :)); % endianess reversed for HW

    tvld = bin2dec(vlds(1:8)); % convert to decimal for helper function

    bvld = bin2dec(vlds(8:16)); % convert to decimal for helper function


    fprintf(i_file, test_input_gen('vld', 'set_write_pattern_addr', x));

    fprintf(i_file, test_input_gen('vld', 'write_patt_dx', PM_OFF(r, 1)));

    fprintf(i_file, test_input_gen('vld', 'write_patt_dy', PM_OFF(r, 2)));

    fprintf(i_file, test_input_gen('vld', 'write_patt_jmp', PM_JMP(r, 1)));
    fprintf(i_file, test_input_gen('vld', 'write_patt_top_vlds', tvld));
    fprintf(i_file, test_input_gen('vld', 'write_patt_bot_vlds', bvld));
end

% For each search, parse the register initialization, the pixel requests
% and transfers, and the final output result.
sz_SU = size(SU);
sz_PR = size(PR);
% There is a new row in the setup matrix for each run.
num_searches = sz_SU(1);
pr_pos = 1;

for s = 1:num_searches
    % configure the device
    fprintf(i_file, test_input_gen('vld', 'set_pmv_dx',    SU(s,1)));
    fprintf(i_file, test_input_gen('vld', 'set_pmv_dy',    SU(s,2)));
    fprintf(i_file, test_input_gen('vld', 'set_blkid',     SU(s,3)));
    fprintf(i_file, test_input_gen('vld', 'set_threshold', SU(s,4)));
    fprintf(i_file, test_input_gen('vld', 'set_act_pt_x',  SU(s,5)-1));
    fprintf(i_file, test_input_gen('vld', 'set_act_pt_y',  SU(s,6)-1));
    fprintf(i_file, test_input_gen('vld', 'set_ref_pt_x',  SU(s,7)-1));
    fprintf(i_file, test_input_gen('vld', 'set_ref_pt_y',  SU(s,8)-1));
    fprintf(i_file, test_input_gen('vld', 'set_out_reg',   SU(s,9)));
    fprintf(i_file, test_input_gen('vld', 'set_im_sz_x',   SU(s,10)));
    fprintf(i_file, test_input_gen('vld', 'set_im_sz_y',   SU(s,11)));

    % issue execution, sense last bit of SU for isFS.
    if SU(s, 13) == 1
        fprintf(i_file, test_input_gen('vld', 'start_search', ...
            ['fs', SU(s, 12:13)]));
    else
        fprintf(i_file, test_input_gen('vld', 'start_search', ...
            ['ps', SU(s, 14)]));
    end

    % For each pixel request in the transcript, add the requested pixels
    % onto the input file.  The termination point is the pixel request with
    % all 0s as its operands, as that is an invalid type.  Also add the
    % pixel request onto the output pipe.
    for x = pr_pos:sz_PR(1)
        pr_pos = x;
        rstart = PR(x, 2);
        rend = PR(x, 2) + PR(x, 4) - 1;
        cstart = PR(x, 1);
        cend = PR(x, 1) + PR(x, 3) - 1;

        rx = PR(x,1) -1; %correction by 1 for matlab -> real
        ry = PR(x,2) -1; %correction by 1 for matlab -> real
        rw = PR(x,3);
        rh = PR(x,4);

        if ((rstart == 0) && (rend == -1) && (cstart == 0) && (cend == -1))
            pr_pos = pr_pos + 1;
            break;
        else
            % add inputs
            pixels = RIm(rstart:rend, cstart:cend);
            sz_pixels = size(pixels);
            fprintf(i_file, ...
                test_input_gen('vld', 'burst_pixel_write', pixels));
            % add outputs
            req = [rx, ry, rw, rh];
            fprintf(o_file, test_output_gen('pix_req', req));
        end
    end

    % add search result to output
    % need to process search result to split it into top and bottom values.
    fprintf(o_file, test_output_gen('read_srch_res', ...
        [RS(s, 3) (RS(s, 1)-1) (RS(s, 2)-1)]));
end
```

```
% close out files
fprintf(i_file, test_input_gen('emp', 'input_empty'));
fprintf(i_file, test_input_gen('end', 'input_end'));
fprintf(o_file, test_output_gen('output_end'));

fclose(i_file);
fclose(o_file);
fprintf('Requested Test Generated\n');

end
```

B.2 test_input_gen.m

```
function [ wd ] = test_input_gen( prefix, cmd, operand )
%TEST_OUTPUT_GEN Helper Function to create valid ME2 Instructions
word_sz = 16;
instr_sz = 5;
fill = 0;

    % Parse Prefix
    switch prefix
        case 'vld'
            pfx = '10_';
        case 'emp'
            pfx = '00_';
        case 'end'
            pfx = 'XX_';
    end

    % Parse Command
    switch cmd
        case 'write_burst_act'
            instruction = 0;
            op_sz = 0;
            ops = '__';
        case 'set_burst_ref_x'
            instruction = 1;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];

        case 'set_burst_ref_y'
            instruction = 2;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];

        case 'write_burst_ref'
            instruction = 3;
            op_sz = 0;
            ops = '__';

        case 'set_burst_width'
            instruction = 4;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_burst_height'
            instruction = 5;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_write_pattern_addr'
            instruction = 6;
            op_sz = 6;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'write_patt_dx'
            instruction = 7;
            op_sz = 9;
            ops = [dec2bin_negsup(operand, op_sz), '__'];
        case 'write_patt_dy'
            instruction = 8;
            op_sz = 9;
            ops = [dec2bin_negsup(operand, op_sz), '__'];
        case 'write_patt_jmp'
            instruction = 9;
            op_sz = 6;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'write_patt_top_vlds'
            instruction = 10;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'write_patt_bot_vlds'
            instruction = 11;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_pmv_dx'
            instruction = 12;
            op_sz = 9;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_pmv_dy'
            instruction = 13;
            op_sz = 9;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_blkid'
            instruction = 14;
            op_sz = 4;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_thresh_top'
            instruction = 15;
            op_sz = 10;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_thresh_bot'
            instruction = 16;
            op_sz = 10;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_threshold'
            % special case, compound word.  Operand needs to be split into
            % its top and bottom msbs. Option for recursion here, but since
            % matlab can handle our problem for us natively, why bother?
            bits = dec2bin(operand, 20);
            wd1 = [pfx, dec2bin(15, 5), '_0', bits(1:10), '___\n'];
            wd2 = [pfx, dec2bin(16, 5), '_0', bits(11:20), '___\n'];
            wd = [wd1; wd2];
            wd = wd';
            return;
        case 'set_act_pt_x'
            instruction = 17;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_act_pt_y'
            instruction = 18;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_ref_pt_x'
            instruction = 19;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_ref_pt_y'
            instruction = 20;
            op_sz = 8;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_out_reg'
            instruction = 21;
            op_sz = 6;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'start_search'
            instruction = 22;
            if(strcmp(operand(1:2), 'ps'))
                op_sz = 7;
                ops = [dec2bin(operand(3), 6) '__1'];
            else
                if(strcmp(operand(1:2), 'fs'))
                    op_sz = 11;
                    ops = [dec2bin(operand(3), 5), '_' ...
                        dec2bin(operand(4), 5), '_0'];
                else
                    fprintf(...
                    'ERROR: Search invoked but bad ps/fs for operand1\n');
                    return;
                end
            end
        case 'send_pixels'
            instruction = 23;
            op_sz = 11;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_im_sz_x'
            instruction = 24;
            op_sz = 11;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'set_im_sz_y'
            instruction = 25;
            op_sz = 11;
            ops = [dec2bin(operand, op_sz), '__'];
        case 'read_ref_mem'
            instruction = 28;
            op_sz = 10;
            ops = [dec2bin(operand(1), 5), dec2bin(operand(2), 5), '__'];
        case 'read_act_mem'
            instruction = 29;
            op_sz = 10;
            ops = [dec2bin(operand(1), 5), dec2bin(operand(2), 5), '__'];
        case 'read_reg'
            instruction = 30;
            op_sz = 5;
            % Decode Operand into it's integer value, throw error if not a
            % name.
            switch operand
                case 'burst_ref_x'
                    reg_id = 0;
                case 'burst_ref_y'
                    reg_id = 1;
```

```
case 'burst_height'
    reg_id = 2;
case 'burst_width'
    reg_id = 3;
case 'pattern_write_address'
    reg_id = 4;
case 'pmv_dx'
    reg_id = 5;
case 'pmv_dy'
    reg_id = 6;
case 'blkid'
    reg_id = 7;
case 'thresh_top'
    reg_id = 8;
case 'thresh_bot'
    reg_id = 9;
case 'act_pt_x'
    reg_id = 10;
case 'act_pt_y'
    reg_id = 11;
case 'ref_pt_x'
    reg_id = 12;
case 'ref_pt_y'
    reg_id = 13;
case 'out_reg'
    reg_id = 14;
case 'im_sz_x'
    reg_id = 15;
case 'im_sz_y'
    reg_id = 16;
case 'patt_dx'
    reg_id = 17;
case 'patt_dy'
    reg_id = 18;
case 'patt_jmp'
    reg_id = 19;
```

```
        case 'patt_vld_top'
            reg_id = 20;
        case 'patt_vld_bot'
            reg_id = 21;
        otherwise
            printf(...
                'Warning, Invalid Register Requested for Read\n');
            return;
    end
    ops = [dec2bin(reg_id, op_sz), '__'];
case 'issue_ping'
    instruction = 31;
    op_sz = 0;
    ops = '__';
case 'pixel_pair'
    % special case no instruction header, break out.
    wd = [pfx, dec2bin(operand(1), 8), ...
        '_', dec2bin(operand(2), 8), '___\n'];
    return;
case 'input_empty'
    % special case non-instruction, break out.
    wd = [pfx, '00000_00000000000___\n'];
    return;
case 'input_end'
    % special case non-instruction, break out.
    wd = [pfx, 'XXXXX_XXXXXXXXXXX___\n'];
    return;
case 'burst_pixel_write'
    % special case, compound input.  Expected Operand is an array
    % of pixels, sniff width and height from the given array.
    [height, width] = size(operand);

    % Error Catching, 0 heigth or width is invalid
    if height == 0 || width == 0
        fprintf('Warning, H or W is Zero and should not be.\n');
    end
```

```
            wd = [];
            % BUGFIX: must be raster scan order
            %for x = 1:2:width
            %    for y = 1:1:height
            for y = 1:1:height
                for x = 1:2:width
                    wdx = [pfx, dec2bin(operand(y, x), 8), '_' ...
                        dec2bin(operand(y, x+1), 8), '___\n'];
                    wd = [wd; wdx];
                end
            end
            % transpose matrix to have proper printf behavior.
            wd = wd';
            return;
        otherwise
            printf('Warning, invalid command requested\n');
            return;
    end

    fill_sz = word_sz - instr_sz - op_sz;
    wd = [pfx, dec2bin(instruction, 5) '_' dec2bin(fill, fill_sz) ...
        '_' ops '\n'];


end

function [bin] = dec2bin_negsup(num, bound)
    if(num < 0)
        bin = dec2bin(abs(num), bound);
        %invert bits
        bin = double(~(bin-'0'));
        % recompress
        ibin = [];
        for x = 1:length(bin)
            ibin = [ibin, num2str(bin(x))];
        end
        %add one
        num = bin2dec(ibin);
        num = num + 1;
        bin = dec2bin(num, bound);
    else
        bin = dec2bin(num, bound);
    end
end
```

B.3 test_output_gen.m

```
function [ wds ] = test_output_gen( cmd, operands )
%TEST_OUTPUT_GEN Create Expected Output Words, given CMD & OPRs
    % multi-word outputs get transposed before being returned, because
    % fprintf reads down a column before reading across a row.  Important
    % to know if you want to write any more of these.

switch cmd
    case 'read_srch_res'
        instr_id = 24;
        %sad_top = operands(1);
        %sad_bot = operands(2);
        sad_v = operands(1);
        sad_x = operands(2);
        sad_y = operands(3);

        sad_v = dec2bin(sad_v, 20);
        sad_vt = sad_v(1:10);
        sad_vb = sad_v(11:20);
        sad_vt = bin2dec(sad_vt);
        sad_vb = bin2dec(sad_vb);

        wd0 = [dec2bin(instr_id, 5), '_', dec2bin(sad_vt, 11), '\n'];
        wd1 = [dec2bin(instr_id, 5), '_', dec2bin(sad_vb, 11), '\n'];
        wd2 = [dec2bin(instr_id, 5), '_', dec2bin(sad_x, 11), '\n'];
        wd3 = [dec2bin(instr_id, 5), '_', dec2bin(sad_y, 11), '\n'];
        wds = [wd0 ; wd1; wd2; wd3];
        wds = wds';

    case 'read_reg_res'
        instr_id = 25;
        wds = [dec2bin(instr_id, 5), '_', dec2bin(operands, 11), '\n'];

    case 'pix_req'
        instr_id = 26;
        rq_1_x = operands(1);
        rq_1_y = operands(2);
        rq_1_w = operands(3);
        rq_1_h = operands(4);
        wd0 = [dec2bin(instr_id, 5), '_', dec2bin(rq_1_x, 11), '\n'];
        wd1 = [dec2bin(instr_id, 5), '_', dec2bin(rq_1_y, 11), '\n'];
        wd2 = [dec2bin(instr_id, 5), '_', dec2bin(rq_1_w, 11), '\n'];
        wd3 = [dec2bin(instr_id, 5), '_', dec2bin(rq_1_h, 11), '\n'];
        wds = [wd0 ; wd1; wd2; wd3];
        if (length(operands) > 4)
            rq_2_x = operands(5);
            rq_2_y = operands(6);
            rq_2_w = operands(7);
            rq_2_h = operands(8);
            wd4 = [dec2bin(instr_id, 5), '_', dec2bin(rq_2_x, 11), '\n'];
            wd5 = [dec2bin(instr_id, 5), '_', dec2bin(rq_2_y, 11), '\n'];
            wd6 = [dec2bin(instr_id, 5), '_', dec2bin(rq_2_w, 11), '\n'];
            wd7 = [dec2bin(instr_id, 5), '_', dec2bin(rq_2_h, 11), '\n'];
            wds = [wds ; wd4; wd5; wd6; wd7];
        end
        wds = wds';

    case 'pix_transmit'
        instr_id = 27;
        % expected operands is a 4x4 block of pixels.
        wd0 = [dec2bin(instr_id, 5), '_' dec2bin(16, 11), '\n'];
        wds = wd0;
        %for x = 1:2:4
        %    for y = 1:1:4
        for y = 1:1:4
            for x = 1:2:4
                wdx = [dec2bin(operands(y, x+1), 8), '_' ...
                    dec2bin(operands(y, x), 8), '\n'];
                wds = [wds; wdx];
            end
        end
        wds = wds';

    case 'out_pix_pair'
        wds = [dec2bin(operand(1), 8), '_', dec2bin(operand(2), 8), '\n'];

    case 'ping_out'
        instr_id = 31;
        wds = [dec2bin(instr_id, 5), '_', dec2bin(0, 11), '\n'];

    case 'output_end'
        wds = 'XXXXX_XXXXXXXXXXX\n';

    otherwise
        printf('Not an output word.\n');
        return;
end


end
```

Appendix C

Testbench Code

C.1 me2_top.vt

```
// Michael Braly - mabraly@ucdavis.edu
// ---------------------------------------------------------------------------
// Description:
// Automated Testbench to verify proper functioning of: me2_top.v
// ---------------------------------------------------------------------------
// History:
// 02/25/2014 - Created
// 09/21/2014 - Modified to Act like FIFOs for Wrapping Tests
// 11/17/2014 - Adapted to Test the top level ME_ACC2
// ---------------------------------------------------------------------------
module automated_testbench_me2();

// Set the target test vector file here:
`define TV_FILE_IN "autogen_two_simple_pat_mem_move_in.tv"
`define TV_FILE_OUT "autogen_two_simple_pat_mem_move_out.tv"
//`define TV_FILE_IN "me2_ref_wr_rd_all_test_in.tv"
//`define TV_FILE_OUT "me2_ref_wr_rd_all_test_out.tv"

// Input and Output widths do not include the clk signal or the async reset
parameter DUT_INPUT_WIDTH = 18;
parameter DUT_OUTPUT_WIDTH = 16;
parameter CLK_PERIOD = 10;
parameter NUM_STAGES = 7;

// Testbench System Signals
reg         clk, reset;
reg [31:0]  invectornum, errors, numstalls;
reg [31:0]  outvectornum;
reg [31:0]  cycle_count;
reg [DUT_INPUT_WIDTH-1:0]   testvectors_in[100000:0];
reg [DUT_OUTPUT_WIDTH-1:0]  testvectors_out[100000:0];
reg [DUT_INPUT_WIDTH-1:0]   inputs_applied;
reg [DUT_OUTPUT_WIDTH-1:0]  outputs_expected;
wire    [DUT_OUTPUT_WIDTH-1:0]  actual_outputs;
wire                full_idle;
wire                stalled_waiting_for_input_rdy;
wire                stalled_waiting_for_output_rdy;

// FIFO Control Wires
wire        data_in_rdy; // TV Driven
wire        get_next_data_in;
wire        data_out_rdy;
wire        data_out_full; // TV Driven

// DUT Inputs (Need a wire for every possible non-clk/rst input)
wire    [15:0]  data_in;

// DUT Outputs (Need a register + wire pair for every output to be tested)
wire    [15:0]  data_out;
wire    [15:0]  data_out_expected;

// Structural assignments so that the testbench logic can be written in a
// general manner for easier reuse.
assign {data_in_rdy, data_out_full, data_in}
    = inputs_applied;
assign {data_out_expected}
```

```
    = outputs_expected;

assign actual_outputs = {data_out};



// Instantiate device under test (hook outputs to the wire part of each pair)
me2_top dut(
    .clk(clk),
    .rst(reset),
    .data_in_rdy(data_in_rdy),
    .get_next_data_in(get_next_data_in),
    .data_in(data_in),
    .data_out(data_out),
    .data_out_rdy(data_out_rdy),
    .data_out_full(data_out_full),
    .full_idle(full_idle),
    .input_stall(stalled_waiting_for_input_rdy),
    .output_stall(stalled_waiting_for_output_rdy));

// ----------------------------------------------------------------------------
// Begin Testbench - Should not need to edit anything below here.
// ----------------------------------------------------------------------------
// Generate clock
always
    begin
        clk = 1;
        #(CLK_PERIOD/2);
        clk = 0;
        #(CLK_PERIOD/2);
    end

// load vectors before beginning testing, and pulse reset as long as it takes
// to flush any existing pipeline.
initial
    begin
        $readmemb( `TV_FILE_IN, testvectors_in);
        $readmemb( `TV_FILE_OUT, testvectors_out);
        invectornum = 0;
        outvectornum = 0;
        numstalls = 0;
        errors = 0;
        reset = 1;
        #(CLK_PERIOD * NUM_STAGES);
        reset = 0;
        cycle_count = 0;
    end

// Apply test vectors on the rising edge of the clock.
always @ (posedge clk)
    begin
    //#1;
    inputs_applied = testvectors_in[invectornum];
    outputs_expected = testvectors_out[outvectornum];
    cycle_count = cycle_count + 1;
    end

// Check results of test vector on falling edge of the clock.
always @ (negedge clk)
if(~reset)
begin // skip during reset
    // if data_out_rdy && ~data_out_full, evaluate output
    if (data_out_rdy && ~data_out_full)
    begin
        if ({actual_outputs} !== {outputs_expected})
        begin
        $display ("Error:");
        $display (" cycle number = %d", cycle_count);
        $display (" vector number = %d", outvectornum);
        //$display (" input    = %b", inputs_applied);
        $display (" outputs  = %h", actual_outputs);
        $display (" expected = %h", outputs_expected);
        errors = errors + 1;
        end
        else
        begin
        $display ("Passed");
        //$display (" cycle Number = %d", cycle_count);
        //$display (" outputs = %b", actual_outputs);
        end
        outvectornum = outvectornum + 1;
    end

    // if unit requesting new input, or we're modelling a full output pipe
    // increment to next test vector.
    if (get_next_data_in || data_out_full)
    begin
        invectornum = invectornum + 1;
        if (data_out_full)
        begin
            numstalls = numstalls + 1;
        end
    end

    if (testvectors_out[outvectornum] === {(DUT_OUTPUT_WIDTH){1'bx}})
    begin
    $display ("----------------------");
        if(errors == 0) $display("Test Status - PASSED |");
        else $display("Test Status - FAILED |");
    $display ("----------------------");
    $display ("%d inputs applied", invectornum);
    $display ("%d input stalls applied", numstalls);
    $display ("%d output words checked", outvectornum);
    $display ("%d errors", errors);
    $display ("%d cycles elapsed", cycle_count);
    $stop;
    end
end
endmodule
```

Appendix D

Top-Level Hierarchical FSM

Hierarchical finite state machines are a technique for managing the complexity of a controller with many separate states, but relatively ordered transitions [89]. For relatively simple state machines with fewer than about 7 states, such as the execution controller which runs the pixel datapath pipeline on MEACC2, and as shown in Figure D.1, the whole block can be held in the active memory of a single designer. As the FSM state space grows eventually it becomes easier to partition the design. In MEACC2 the top level controller of was designed and implemented as a hierarchical finite state machine and Figure D.2 shows the dependencies between each of the constituent FSMs. Since both the pattern and full search FSMs make use of the scan FSM, there is a design choice to either replicate the Scan FSM, or manage the transition to and from idle edges in the Scan FSM so that it can be responsive to both pattern and full search FSMs. Since the device will never have both the pattern and full search FSMs out of their idle states at the same time, it is safe to reuse the same Scan FSM for both HFSMs.

D.1 Transparent Hierarchical FSMs

The goal of the partitioning is to make the block easier to design, without impacting control delays or other timing sensitive paths. As an example of how these partitions can be made, the state transition diagram of the Request Pixel FSM is given in Figure D.3 in flattened form. The collection of states that make up the Load Requested Pixels FSM are

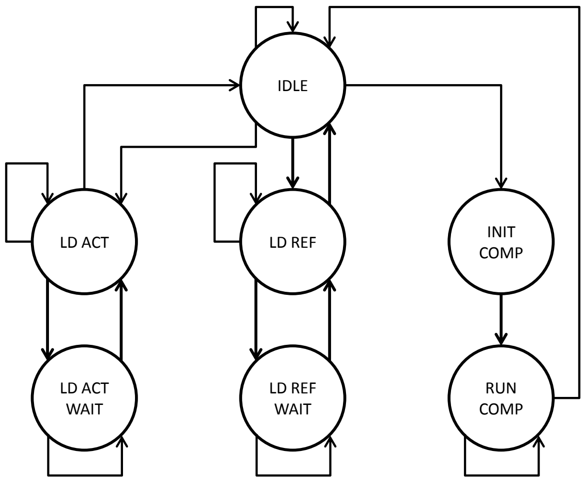


Figure D.1: State diagram for the execution controller

. With only 7 states, the complexity of the FSM is such that the whole design can be kept in the designer's memory at once.

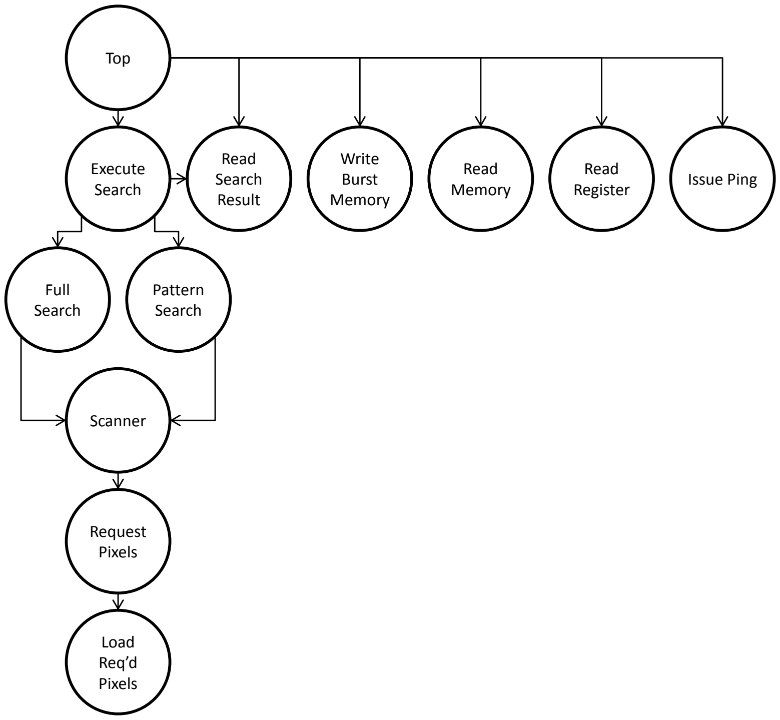


Figure D.2: Dependency diagram for the top level controller

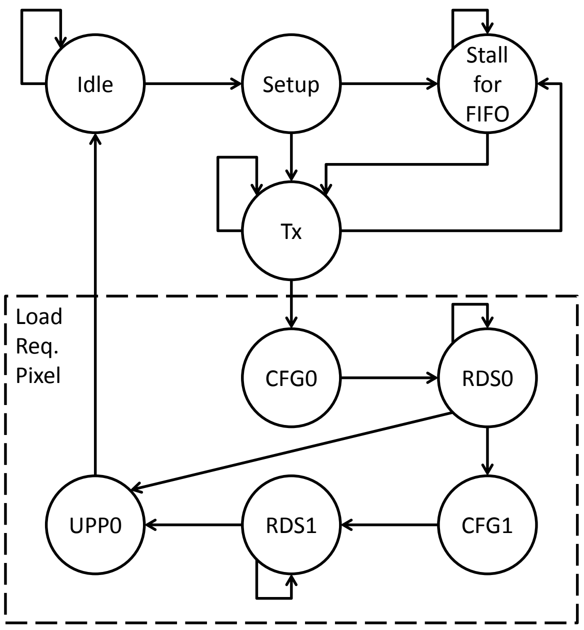


Figure D.3: Flattened state diagram for request pixel FSMs

. This version of the state diagram has all the states for both request pixel and load pixel FSM components of the top level controller.

a sub-graph of the overall Request Pixel FSM, and only have a single entrance and exit path from the rest of the FSM. Therefore, the FSM can be partitioned, as shown in Figure D.4 and Figure D.5, with the only change being the addition of an additional idle state to the Load Requested Pixels FSM, and the addition of a composite state representing the Load Requested Pixel FSM functionality in the Request Pixel FSM. By carefully choosing the transition edges, the new sub-state machine will transition out of its idle state in parallel with the Load Requested Pixels FSM transition into the composite state in its own graph, successfully partitioning the design without adding any additional design latency.

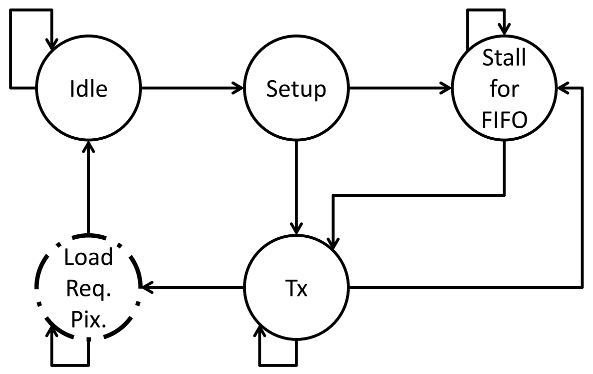


Figure D.4: Hierarchical state diagram for request pixels FSM

. The collection of states which made up load requested pixels are combined into a composite state, simplifying the implementation of the request pixels FSM.

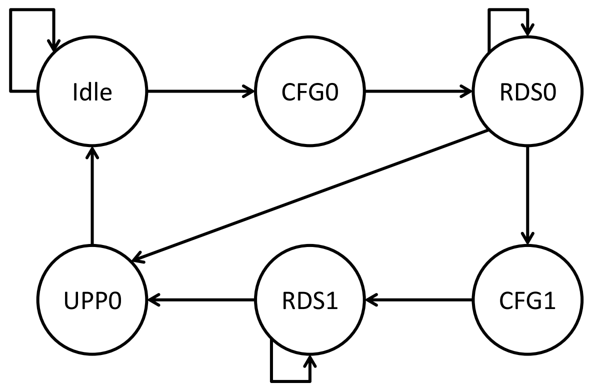


Figure D.5: Hierarchical state diagram for load requested pixels FSM

. The collection of states making up load requested pixels need their own additional IDLE state to be fully self-contained.

Bibliography

[1] S. Vassiliadis, E.A. Hakkennes, J.S.S.M. Wong, and G.G. Pechanek. The sum-absolute-difference motion estimation accelerator. In *Euromicro Conference, 1998. Proceedings. 24th*, volume 2, pages 559–566 vol.2, Aug 1998.

[2] Iain E Richardson. *The H. 264 advanced video compression standard.* John Wiley & Sons, 2011.

[3] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, July 2003.

[4] Detlev Marpe, Gabi Blättermann, and Thomas Wiegand. Adaptive codes for h. 26l. *ITU-T Telecommunications Standardization Sector*, pages 9–12, 2001.

[5] Gisle Bjontegaard and Karl Lillevold. Context-adaptive vlc coding of coefficients. *JVT Document JVT-C028, Fairfax, VA*, 19, 2002.

[6] J. Ohm, G.J. Sullivan, H. Schwarz, Thiow Keng Tan, and T. Wiegand. Comparison of the coding efficiency of video coding standards - including high efficiency video coding (hevc). *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1669–1684, Dec 2012.

[7] Vivienne Sze, Madhukar Budagavi, and Gary J Sullivan. *High Efficiency Video Coding (HEVC).* Springer, 2014.

[8] Il-Koo Kim, Sunil Lee, Min-Su Cheon, T. Lee, and JeongHoon Park. Coding efficiency improvement of hevc using asymmetric motion partitioning. In *Broadband Multimedia Systems and Broadcasting (BMSB), 2012 IEEE International Symposium on*, pages 1–4, June 2012.

[9] J. Ohm and G.J. Sullivan. High efficiency video coding: the next frontier in video compression [Standards in a Nutshell]. *Signal Processing Magazine, IEEE*, 30(1):152–158, Jan 2013.

[10] G.J. Sullivan, J. Ohm, Woo-Jin Han, and T. Wiegand. Overview of the high efficiency video coding (hevc) standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1649–1668, Dec 2012.

[11] H. Koumaras, M. Kourtis, and Drakoulis Martakos. Benchmarking the encoding efficiency of h.265/hevc and h.264/avc. In *Future Network Mobile Summit (FutureNetw), 2012*, pages 1–7, July 2012.

[12] P. Helle, H. Lakshman, M. Siekmann, J. Stegemann, T. Hinz, H. Schwarz, D. Marpe, and T. Wiegand. A scalable video coding extension of hevc. In *Data Compression Conference (DCC), 2013*, pages 201–210, March 2013.

[13] J. Vaisey and A. Gersho. Image compression with variable block size segmentation. *Signal Processing, IEEE Transactions on*, 40(8):2040–2060, Aug 1992.

[14] A. Ahmad, N. Khan, S. Masud, and M.A. Maud. Efficient block size selection in h.264 video coding standard. *Electronics Letters*, 40(1):19–21, Jan 2004.

[15] Hongtao Song, Zhiyong Gao, and Xiaoyun Zhang. Novel fast motion estimation and mode decision for h.264 real-time high-definition encoding. In *Image and Signal Processing (CISP), 2012 5th International Congress on*, pages 43–48, Oct 2012.

[16] S. Oudin, P. Helle, J. Stegemann, C. Bartnik, B. Bross, D. Marpe, H. Schwarz, and T. Wiegand. Block merging for quadtree-based video coding. In *Multimedia and Expo (ICME), 2011 IEEE International Conference on*, pages 1–6, July 2011.

[17] Muhammad Usman Karim Khan, Muhammad Shafique, Mateus Grellert, and Jorg Henkel. Hardware-software collaborative complexity reduction scheme for the emerging hevc intra encoder. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 125–128, March 2013.

[18] A. Fuldseth, M. Horowitz, Shilin Xu, K. Misra, A. Segall, and Minhua Zhou. Tiles for managing computational complexity of video encoding and decoding. In *Picture Coding Symposium (PCS), 2012*, pages 389–392, May 2012.

[19] V. Sze and A.P. Chandrakasan. A highly parallel and scalable cabac decoder for next generation video coding. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 126–128, Feb 2011.

[20] F. Pescador, M.J. Garrido, E. Juarez, and C. Sanz. On an implementation of hevc video decoders with dsp technology. In *Consumer Electronics (ICCE), 2013 IEEE International Conference on*, pages 121–122, Jan 2013.

[21] Dajiang Zhou, Jinjia Zhou, Xun He, Jiayi Zhu, Ji Kong, Peilin Liu, and S. Goto. A 530 mpixels/s 4096x2160, 60fps h.264/avc high profile video decoder chip. *Solid-State Circuits, IEEE Journal of*, 46(4):777–788, April 2011.

[22] B. M. Baas. A parallel programmable energy-efficient architecture for computationally-intensive DSP systems. In *Signals, Systems and Computers, 2003. The Thirty-Seventh Asilomar Conference on*, volume 2, pages 2185–2189, November 2003.

[23] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas. An asynchronous array of simple processors for DSP applications. In *IEEE International Solid-State Circuits Conference (ISSCC)*, volume 49, pages 428–429, 663, February 2006.

[24] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Daniel Gurman, Chi Chen, Jason Cheung, and Tinoosh Mohsenin. Hardware and applications of AsAP: An asynchronous array of simple processors. In *IEEE HotChips Symposium on High-Performance Chips*, August 2006.

[25] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Dean Truong, Tinoosh Mohsenin, and Bevan Baas. AsAP: An asynchronous array of simple processors. *IEEE Journal of Solid-State Circuits (JSSC)*, 43(3):695–705, March 2008.

[26] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *Symposium on VLSI Circuits*, pages 22–23, June 2008.

[27] D. N. Truong, W. H. Cheng, T. Mohsenin, Z. Yu, A. T. Jacobson, G. Landge, M. J. Meeuwsen, A. T. Tran, Z. Xiao, E. W. Work, J. W. Webb, P. Mejia, and B. M. Baas. A 167-processor computational platform in 65 nm CMOS. *IEEE Journal of Solid-State Circuits (JSSC)*, 44(4):1130–1144, April 2009.

[28] Z. Xiao, S. Le, and B. M. Baas. A fine-grained parallel implementation of a h.264/avc encoder on a 167-processor computational platform. In *IEEE Asilomar Conference on Signals, Systems and Computers*, November 2011.

[29] Ryan W. Apperson. A dual-clock FIFO for the reliable transfer of high-throughput data between unrelated clock domains. Master's thesis, University of California, Davis, CA, USA, September 2004. `http://www.ece.ucdavis.edu/cerl/techreports/2004-5/`.

[30] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, and Jason Cheung. AsAP: A fine-grain multi-core platform for DSP applications. *IEEE Micro*, 27(2):34–45, March 2007.

[31] Anthony T. Jacobson. A continuous-flow mixed-radix dynamically-configurable fft processor. Master's thesis, University of California, Davis, CA, USA, July 2007. `http://www.ece.ucdavis.edu/vcl/pubs/theses/2007-3`.

[32] Stephen T. Le. A fine grained many-core h.264 video encoder. Master's thesis, University of California, Davis, CA, USA, March 2010. `http://www.ece.ucdavis.edu/vcl/pubs/theses/2010-03`.

[33] Aaron Stillmaker. *Design of Energy-Efficient Many-Core MIMD GALS Processor Arrays in the 1000-Processor Era.* PhD thesis, University of California, Davis, Davis, CA, USA, Dec. 2015. `http://www.vcl.ece.ucdavis.edu/pubs/theses/2015-1/`.

[34] Eric W. Work. Algorithms and software tools for mapping arbitrarily connected tasks onto an asynchronous array of simple processors. Master's thesis, University of California, Davis, CA, USA, September 2007. `http://www.ece.ucdavis.edu/vcl/pubs/theses/2007-4`.

[35] A.T. Tran, D.N. Truong, and B.M. Baas. A low-cost high-speed source-synchronous interconnection technique for GALS chip multiprocessors. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 996–999, May. 2009.

[36] A. T. Tran, D. N. Truong, and B. M. Baas. A reconfigurable source-synchronous on-chip network for GALS many-core platforms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(6):897–910, Jun. 2010.

[37] Anh Tran, Dean Truong, and Bevan Baas. A complete full-rate 802.11a baseband reciever implemented on an array of programmable processors. In *Asilomar Conference on Signals, Systems and Computers*, October 2008.

[38] A. T. Tran and B. M. Baas. Design of bufferless on-chip routers providing in-order packet delivery. In *SRC Technology and Talent for the 21st Century (TECHCON)*, page S14.3, Sep. 2011.

[39] A. T. Tran and B. M. Baas. RoShaQ: High-performance on-chip router with shared queues. In *IEEE International Conference on Computer Design (ICCD)*, pages 232–238, October 2011.

[40] Michael J. Meeuwsen. A shared memory module for an asynchronous array of simple processors. Master's thesis, University of California, Davis, CA, USA, April 2005. `http://http://www.ece.ucdavis.edu/cerl/techreports/2005-2/`.

[41] Michael Meeuwsen, Zhiyi Yu, and Bevan M. Baas. A shared memory module for asynchronous arrays of processors. *EURASIP Journal on Embedded Systems*, 2007:Article ID 86273, 13 pages, 2007.

[42] Z. Yu and B. Baas. Performance and power analysis of globally asynchronous locally synchronous multi-processor systems. In *IEEE Computer Society Annual Symposium on VLSI*, March 2006.

[43] Z. Yu and B. M. Baas. Implementing tile-based chip multiprocessors with GALS clocking styles. In *IEEE International Conference of Computer Design (ICCD)*, October 2006.

[44] Soheil Ghiasi Bin Liu, Mohammad H. Foroozannejad and Bevan M. Baas. Optimizing power of many-core systems by exploiting dynamic voltage, frequency and core scaling. In *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2015.

[45] D. Larkin, V. Muresan, and N. O'Connor. A low complexity hardware architecture for motion estimation. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4 pp.–, May 2006.

[46] An-Chao Tsai, Kuan-I Lee, Jhing-Fa Wang, and Jar-Ferr Yang. Vlsi architecture designs for effective h.264/avc variable block-size motion estimation. In *Audio, Language and Image Processing, 2008. ICALIP 2008. International Conference on*, pages 413–417, July 2008.

[47] Xuena Bao, Dajiang Zhou, Peilin Liu, and S. Goto. An advanced hierarchical motion estimation scheme with lossless frame recompression and early-level termination for beyond high-definition video coding. *Multimedia, IEEE Transactions on*, 14(2):237–249, April 2012.

[48] G. Sanchez, D. Noble, M. Porto, and L. Agostini. High efficient motion estimation architecture with integrated motion compensation and fme support. In *Circuits and Systems (LASCAS), 2011 IEEE Second Latin American Symposium on*, pages 1–4, Feb 2011.

[49] N. Purnachand, L.N. Alves, and A. Navarro. Fast motion estimation algorithm for hevc. In *Consumer Electronics - Berlin (ICCE-Berlin), 2012 IEEE International Conference on*, pages 34–37, Sept 2012.

[50] S. Wuytack, J.-P. Diguet, F.V.M. Catthoor, and H.J. de Man. Formalized methodology for data reuse: exploration for low-power hierarchical memory mappings. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 6(4):529–537, Dec 1998.

[51] Jen-Chieh Tuan, Tian-Sheuan Chang, and Chein-Wei Jen. On the data reuse and memory bandwidth analysis for full-search block-matching vlsi architecture. *Circuits and Systems for Video Technology, IEEE Transactions on*, 12(1):61–72, Jan 2002.

[52] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis. Multimedia rectangularly addressable memory. *Multimedia, IEEE Transactions on*, 8(2):315–322, April 2006.

[53] J.K. Tanskanen, T. Sihvo, and J. Niittylahti. Byte and modulo addressable parallel memory architecture for video coding. *Circuits and Systems for Video Technology, IEEE Transactions on*, 14(11):1270–1276, Nov 2004.

[54] J. Vanne, E. Aho, T.D. Hamalainen, and K. Kuusilinna. A parallel memory system for variable block-size motion estimation algorithms. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(4):538–543, April 2008.

[55] S. Chandrakar, A. Clements, A. Sudarsanam, and A. Dasu. Memory architecture template for fast block matching algorithms on fpgas. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[56] M.E. Sinangil, A.P. Chandrakasan, V. Sze, and Minhua Zhou. Memory cost vs. coding efficiency trade-offs for hevc motion estimation engine. In *Image Processing (ICIP), 2012 19th IEEE International Conference on*, pages 1533–1536, Sept 2012.

[57] M.E. Sinangil, A.P. Chandrakasan, V. Sze, and Minhua Zhou. Hardware-aware motion estimation search algorithm development for high-efficiency video coding (hevc) standard. In *Image Processing (ICIP), 2012 19th IEEE International Conference on*, pages 1529–1532, Sept 2012.

[58] Yiran Li and Tong Zhang. Reducing dram image data access energy consumption in video processing. *Multimedia, IEEE Transactions on*, 14(2):303–313, April 2012.

[59] P. Meinerzhagen, C. Roth, and A. Burg. Towards generic low-power area-efficient standard cell based memory architectures. In *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pages 129–132, Aug 2010.

[60] P. Meinerzhagen, S.M.Y. Sherazi, A. Burg, and J.N. Rodrigues. Benchmarking of Standard-Cell Based Memories in the Sub- V_T Domain in 65-nm CMOS Technology. *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, 1(2):173–182, June 2011.

[61] P. Meinerzhagen, O. Andersson, B. Mohammadi, Y. Sherazi, A. Burg, and J.N. Rodrigues. A 500 fw/bit 14 fj/bit-access 4kb standard-cell based sub-vt memory in 65nm cmos. In *ESSCIRC (ESSCIRC), 2012 Proceedings of the*, pages 321–324, Sept 2012.

[62] M. Budagavi and Minhua Zhou. Video coding using compressed reference frames. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 1165–1168, March 2008.

[63] A.D. Gupte, B. Amrutur, M.M. Mehendale, A.V. Rao, and M. Budagavi. Memory bandwidth and power reduction using lossy reference frame compression in video encoding. *Circuits and Systems for Video Technology, IEEE Transactions on*, 21(2):225–230, Feb 2011.

[64] D. Silveira, G. Sanchez, M. Grellert, V. Possani, and L. Agostini. Memory bandwidth reduction in video coding systems through context adaptive lossless reference frame compression. In *Programmable Logic (SPL), 2012 VIII Southern Conference on*, pages 1–6, March 2012.

[65] Zhe Wang, D. Chanda, S. Simon, and T. Richter. Memory efficient lossless compression of image sequences with jpeg-ls and temporal prediction. In *Picture Coding Symposium (PCS), 2012*, pages 305–308, May 2012.

[66] Zhan Ma and A. Segall. Frame buffer compression for low-power video coding. In *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pages 757–760, Sept 2011.

[67] H. Kaul, M.A. Anders, S.K. Mathew, S.K. Hsu, A. Agarwal, R.K. Krishnamurthy, and S. Borkar. A 320 mV 56μW 411 GOPS/Watt Ultra-Low Voltage Motion Estimation Accelerator in 65 nm CMOS. *Solid-State Circuits, IEEE Journal of*, 44(1):107–114, Jan 2009.

[68] Jinglin Zhang, J.-F. Nezan, and J.-G. Cousin. Implementation of motion estimation based on heterogeneous parallel computing system with opencl. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 41–45, June 2012.

[69] Xiangwen Wang, Li Song, Min Chen, and Junjie Yang. Paralleling variable block size motion estimation of hevc on cpu plus gpu platform. In *Multimedia and Expo Workshops (ICMEW), 2013 IEEE International Conference on*, pages 1–5, July 2013.

[70] Yeong-Kang Lai and Liang-Gee Chen. A data-interlacing architecture with two-dimensional data-reuse for full-search block-matching algorithm. *Circuits and Systems for Video Technology, IEEE Transactions on*, 8(2):124–127, Apr 1998.

[71] M. Elgamel, A.M. Shams, and M.A. Bayoumi. A comparative analysis for low power motion estimation vlsi architectures. In *Signal Processing Systems, 2000. SiPS 2000. 2000 IEEE Workshop on*, pages 149–158, 2000.

[72] Yu-Wen Huang, Tu-Chih Wang, Bing-Yu Hsieh, and Liang-Gee Chen. Hardware architecture design for variable block size motion estimation in mpeg-4 avc/jvt/itu-t h.264. In *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, volume 2, pages II–796–II–799 vol.2, May 2003.

[73] Lei Deng, Wen Gao, Ming Zeng Hu, and Zhen Zhou Ji. An efficient hardware implementation for motion estimation of avc standard. *Consumer Electronics, IEEE Transactions on*, 51(4):1360–1366, Nov 2005.

[74] Ching-Yeh Chen, Shao-Yi Chien, Yu-Wen Huang, Tung-Chien Chen, Tu-Chih Wang, and Liang-Gee Chen. Analysis and architecture design of variable block-size motion estimation for h.264/avc. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 53(3):578–593, March 2006.

[75] Zheng Zhaoqing, Sang Hongshi, Huang Weifeng, and Shen Xubang. High data reuse vlsi architecture for h.264 motion estimation. In *Communication Technology, 2006. ICCT '06. International Conference on*, pages 1–4, Nov 2006.

[76] J. Byun, Y. Jung, and J. Kim. Design of integer motion estimator of hevc for asymmetric motion-partitioning mode and 4k-uhd. *Electronics Letters*, 49(18):1142–1143, August 2013.

[77] A. Akin, O.C. Ulusel, T.Z. Ozcan, G. Sayilar, and I. Hamzaoglu. A novel power reduction technique for block matching motion estimation hardware. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 269–272, Sept 2011.

[78] H. Niitsuma and T. Maruyama. Sum of absolute difference implementations for image processing on fpgas. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 167–170, Aug 2010.

[79] Zhang Chun, Yang Kun, Mai Songping, and Wang Zhihua. A dsp architecture for motion estimation accelerating. In *Intelligent Multimedia, Video and Speech Processing, 2004. Proceedings of 2004 International Symposium on*, pages 583–586, Oct 2004.

[80] M.R.H. Fatemi, H.F. Ates, and R. Salleh. A bit-serial sum of absolute difference accelerator for variable block size motion estimation of h.264. In *Innovative Technologies in Intelligent Systems and Industrial Applications, 2009. CITISIA 2009*, pages 1–4, July 2009.

[81] J. Vanne, E. Aho, K. Kuusilinna, and T.D. Hamalainen. A configurable motion estimation architecture for block-matching algorithms. *Circuits and Systems for Video Technology, IEEE Transactions on*, 19(4):466–477, April 2009.

[82] Zhibin Xiao, S. Le, and B. Baas. A fine-grained parallel implementation of a h.264/avc encoder on a 167-processor computational platform. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pages 2067–2071, Nov 2011.

[83] Gouri Landge. A configurable motion estimation accelerator for video compression. Master's thesis, University of California, Davis, CA, USA, December 2009. `http://www.ece.ucdavis.edu/vcl/pubs/theses/2009-4`.

[84] Sung Dae Kim and Myung Hoon Sunwoo. Mesip: A configurable and data reusable motion estimation specific instruction-set processor. *Circuits and Systems for Video Technology, IEEE Transactions on*, 23(10):1767–1780, Oct 2013.

[85] Shengqi Yang, W. Wolf, and N. Vijaykrishnan. Power and performance analysis of motion estimation based on hardware and software realizations. *Computers, IEEE Transactions on*, 54(6):714–726, Jun 2005.

[86] J. Vanne, E. Aho, T.D. Hamalainen, and K. Kuusilinna. A high-performance sum of absolute difference implementation for motion estimation. *Circuits and Systems for Video Technology, IEEE Transactions on*, 16(7):876–883, July 2006.

[87] S.K. Chatterjee and I. Chakrabarti. Power efficient motion estimation algorithm and architecture based on pixel truncation. *Consumer Electronics, IEEE Transactions on*, 57(4):1782–1790, November 2011.

[88] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective (3rd Edition)*. Addison Wesley, 3 edition, 5 2004.

[89] Michael Keating. *The Simple Art of SoC Design: Closing the Gap Between RTL and ESL*. Springer Science & Business Media, 2011.