Chapter 3
Low-Power Design Approach

3.1 Overview

In recent years, portable devices such as cellular phones, video camcorders, personal digital assistants and handheld digital TVs are becoming increasingly popular. The portability requirement implies an important issue on power reduction. However, the increased power consumption generally comes from the sophisticated algorithms and architectural challenges. H.264/AVC is the dominant video coding algorithm and requires high speed or high throughput solutions for real-time decoding demands. Many solutions [40]-[42] feature to improve coding throughput and reduce data bus bandwidth. However, this level of power consumption is still not applicable when multimedia capabilities are offered in portable systems.

This chapter thoroughly describes low-power design techniques applicable to a portable multimedia system. Figure 3.1 shows the power profiling of H.264/AVC video decoding for mobile applications [43]. Only power profiling in H.264/AVC is shown here since its power requirements are much higher than that in MPEG-2. In this figure, the numbers in brackets represent the memory size used in that module. Motion compensation and deblocking filter occupy large portions of this pie chart. The reasons are that the motion compensation is the most computationally intensive process and the deblocking filter uses large internal memory to remove long data dependencies. Therefore, there are two issues we have noticed. First, reducing the storage size, such as register and memory, is a key to achieving low-power consumption. Second, most of computations require memory accesses to accomplish a data exchange between logic and memory, leading to the increment of working frequency to meet the real-time decoding demand. Hence, it is obvious that reducing the working frequency cuts data switching activity, resulting in less dynamic power consumption. In the following, we will introduce a collection of low-power techniques, including the reduced register number, improved memory system, and several low-power building blocks.

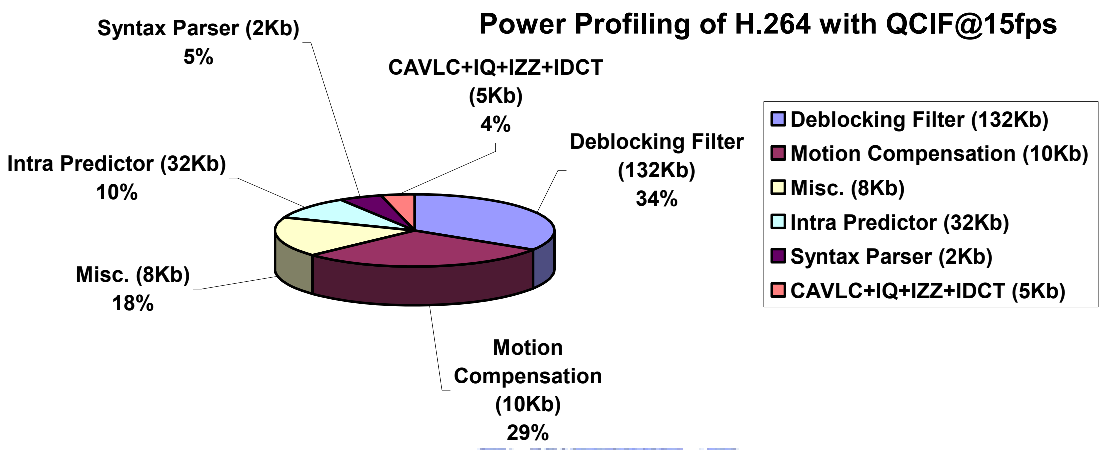


Figure 3.1: Power profiling and memory usage in [43].

3.2 *Reducing Pipeline Registers*

It is obvious that reducing the number of registers cuts the logic and clock-tree power dissipation. To this end, this section presents a new pipeline methodology to reduce the number of pipelined registers at a cost of slight cycle increment. We first present the traditional pipeline methodology and find that the number of pipeline registers relates to the processing cycles of modules between two pipeline stages. After that, we propose a domain-pipelined scalability (DPS) to reduce the numbers of registers.

3.2.1 Pipeline Methodology

Video processing tasks are a time-consuming process and usually computationally challenging because the amount of data to be processed is voluminous. To improve the processing times, pipelining is widely used for exploiting temporal concurrency. It allows a chain of tasks to be divided into stages, with each stage handling results obtained from the previous stage. Pipelining increases the processing throughput – the number of processes completed per unit of time. The increase in processing throughput means that a procedure runs faster and has lower total execution time. However, limitations on practical pipelining arise from additional delay and power on pipelined registers. Those overheads relate to the granularity of pipelining and thereby adversely impact the system power consumption. Considering the video processes in combined MPEG-2 and H.264/AVC, the granularity means the size of 4×4, 8×8 or 16×16 pixels for data transaction between modules. For example, Figure 3.2 shows a data path from stream input to pixel output. CAVLC decoder, inverse zig-zag/quantizer, and IDCT are three main functional units in this path. To decide the granularity in Figure 3.2, Table 3.1 lists two performance indexes with different levels of pipelined granularities from 4×4 sub-blocks to 16×16 macroblock. A fine-level pipelining, 4×4 sub-block levels, requires lesser registers but introduces many stalls or bubbles in each 4×4 level, leading to the increment of processing cycles. On the contrary, a coarse-level pipelining, 16×16 macroblock levels, can reduce the pipelined stalls and bubbles but needs numerous registers to accomplish the pipelining procedures. Hence, we can conclude that the processing cycles and numbers of pipelined registers are often at odds. However, traditional pipelined methods [41][44] didn't consider the processing cycles and used fixed number of pipelined registers over the whole design, leading to additional usage of pipelined registers. In the following, we will present a domain-pipelined scalability (DPS) to alleviate this problem. It takes the processing cycles into account when developing a pipelined methodology in a video decoding system.

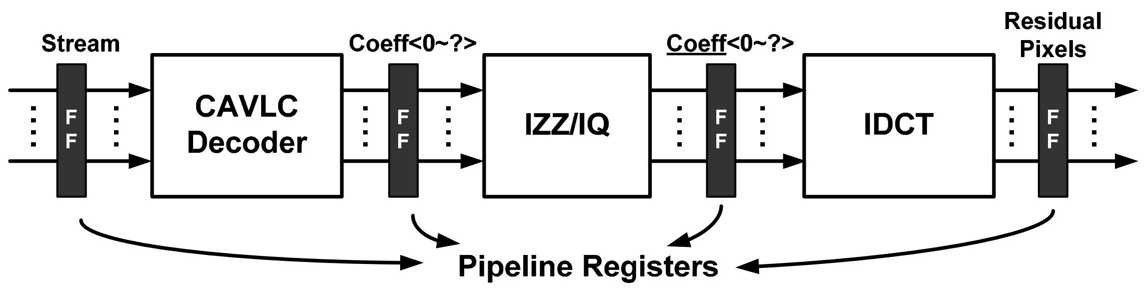


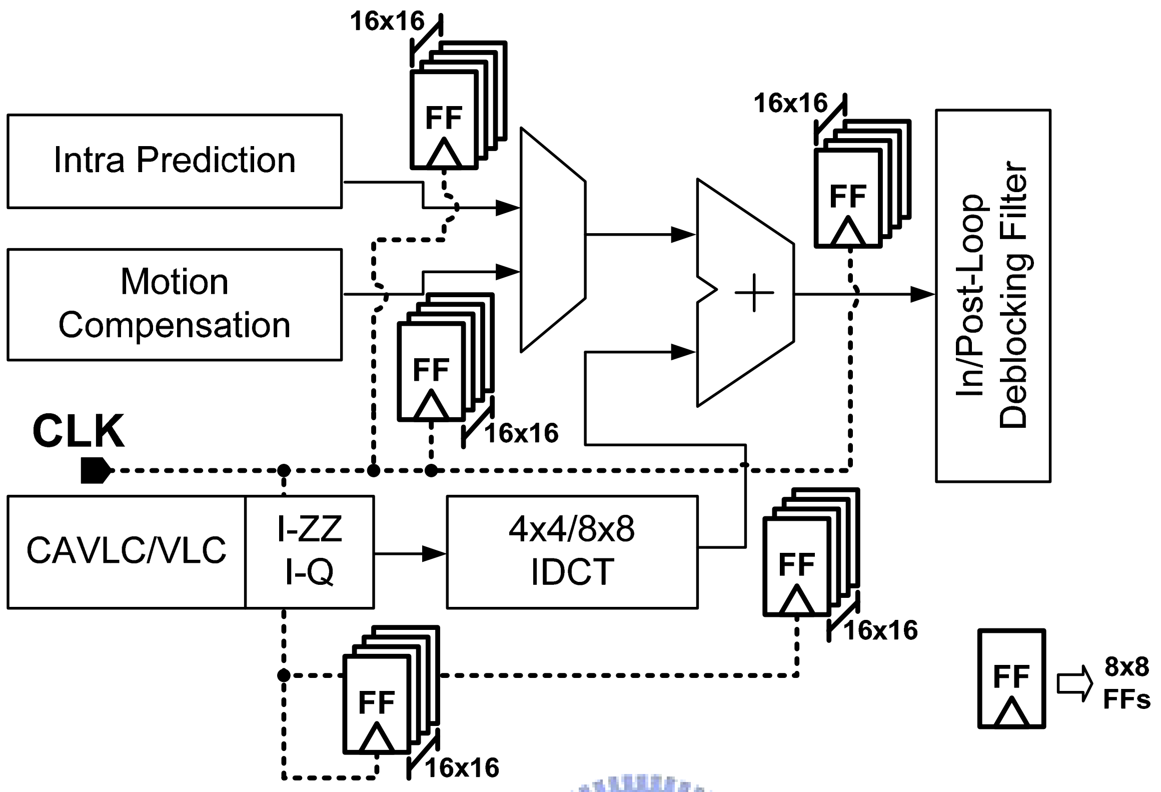
Figure 3.2: Pipelined path between stream inputs and residual pixels.

Table 3.1: Various pipelined granularities.

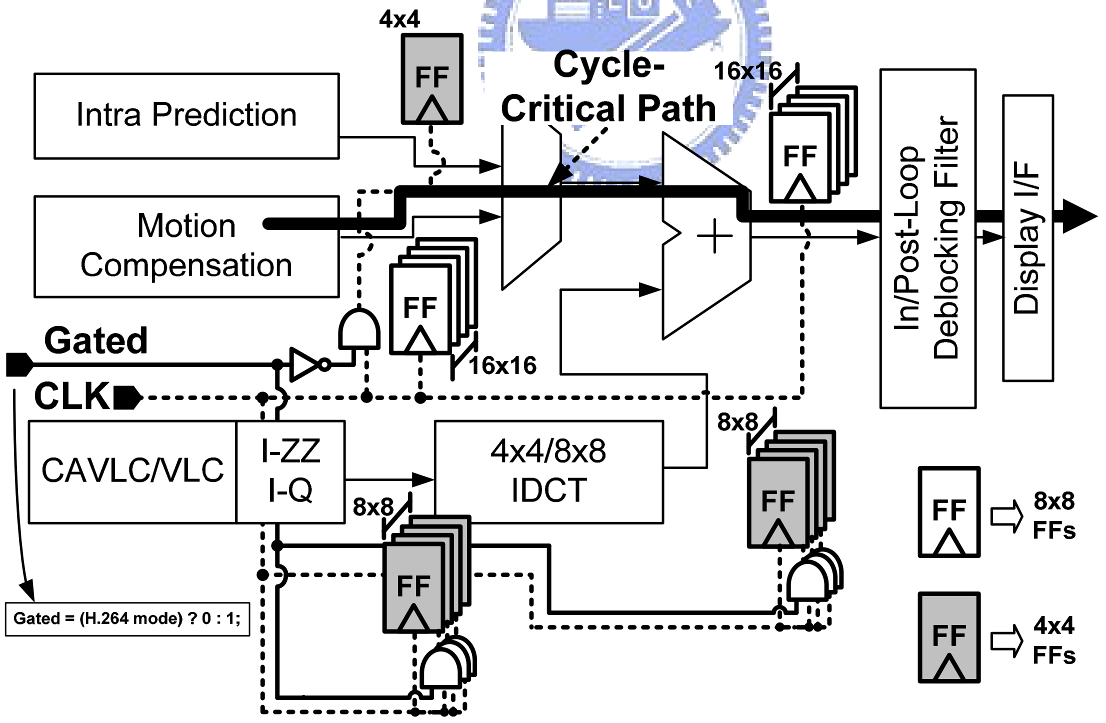
Parallelism	***Unit of Data***	***Buffer Cost***	***Processing Cycles***
MB-Level	16×16 pixels	×16	Y cycles/MB
Block-Level	8×8 pixels	×4	1.19×Y cycles/MB
Subblock-Level	4×4 pixels	×1	1.26×Y cycles/MB

3.2.2 Domain-Pipelined Scalability

Figure 3.3 shows the data paths of H.264/AVC decoder prior to the deblocking filter. The proposed DPS method partitions the paths into two pipelined domains. One of them includes the cycle-critical path that consumes a great number of processing cycles from stream inputs to outputs (e.g. motion compensation, deblocking filter, display I/F, as the thick line of Figure 3.3(b) indicates), and the other includes the non-cycle-critical path that demonstrates the path except for cycle-critical one (e.g. entropy decoder, IDCT, intra prediction). Compared to the fixed 16×16 macroblock-level pipelining [41][44] in Figure 3.3(a), the proposed DPS method in Figure 3.3(b) allocates different pipelined granularities with cycle-awareness. Actually, the numbers of processing cycles will impact the pipeline methodology. For instance, we consider the pipeline level on 4×4 sub-blocks and 16×16 macro-blocks (MB) in Figure 3.4. The shaded and dotted regions represent the previous and next decoding MB respectively. In the non-cycle-critical path, the introduced waiting cycles or bubbles occur frequently on a 4×4 sub-block level. On the other hand, the motion compensation, deblocking filter, and display I/F are performed on 16×16 MB level, in the sense that it improves processing cycles since the waiting cycles are reduced and occur only on each 16×16 level. Therefore, we choose 4×4 sub-block level pipeline to reduce the pipelined register size in non-cycle-critical path. As for the cycle-critical path, we apply 16×16 MB level pipeline to reduce the processing cycles at the cost of acceptable increment on pipeline registers. Therefore, we only apply the fine-level pipeline into the non-cycle-critical path for reducing the pipeline registers. By contrast, the coarse-level pipeline is utilized to eliminate the waiting cycles on the cycle-critical path. With regard to the integration issue, a 4×4 sub-block is the smallest pipelined element in H.264/AVC while an 8×8 block size is adopted by MPEG-2 video standard. Due to different pipelined sizes, we utilize AND gates to disable un-used flip-flops according to the pipelined operation of each standard in Figure 3.3(b). As a result, we optimize the pipeline granularities according to the characteristics of processing cycles [43]. It is suitable for determining the optimized pipelined levels during the system development. The detailed pipeline granularities are listed in Table 3.2. Compared to the un-optimized 16×16-level pipelines [44], the proposed DPS method reduces the number of pipelined registers by 37.5%, resulting in less power dissipation.



(a)



(b)

Figure 3.3: (a) Traditional pipelined method and (b) two pipelined domains by using the DPS method.

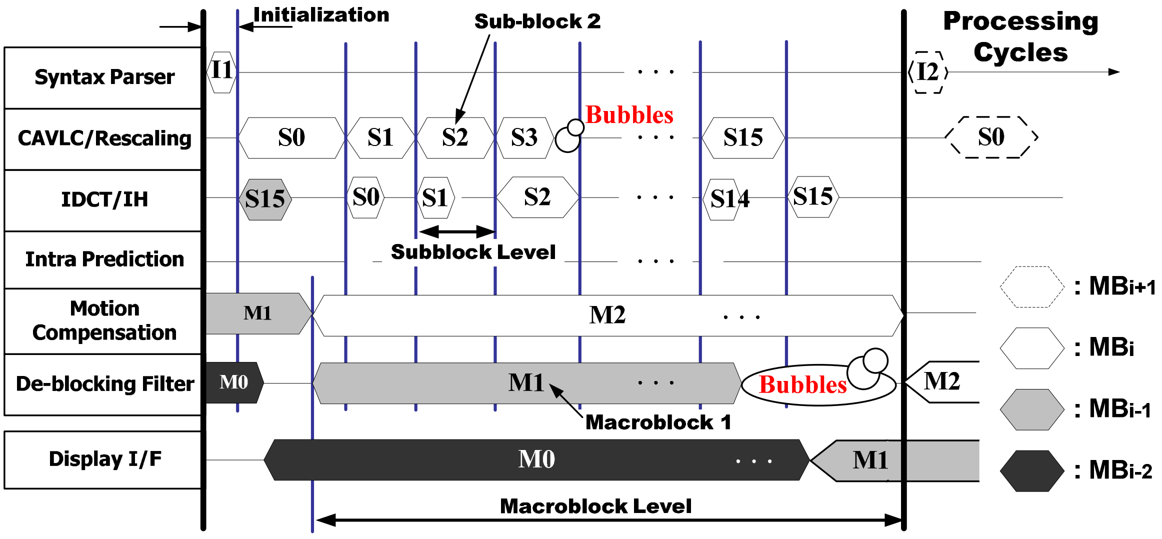


Figure 3.4: A data-path diagram of residual path by DPS.

Table 3.2: Different pipeline levels in each module.

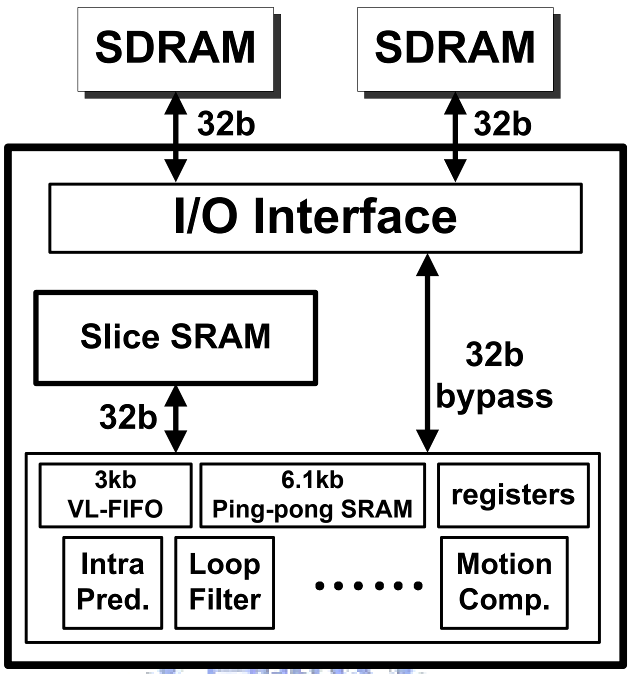
Cycle Characteristics	Key Module	Proposed	
		MPEG-2	H.264/AVC
Non-cycle-critical path	Intra Prediction	N/A	4×4
	VLD/CAVLD	8×8	
	4×4/8×8 IDCT		
Cycle-critical path	Motion Compensation	16×16	16×16
	In/Post-Loop Filter		
	Display I/F		

3.3 Improving the Memory Hierarchy

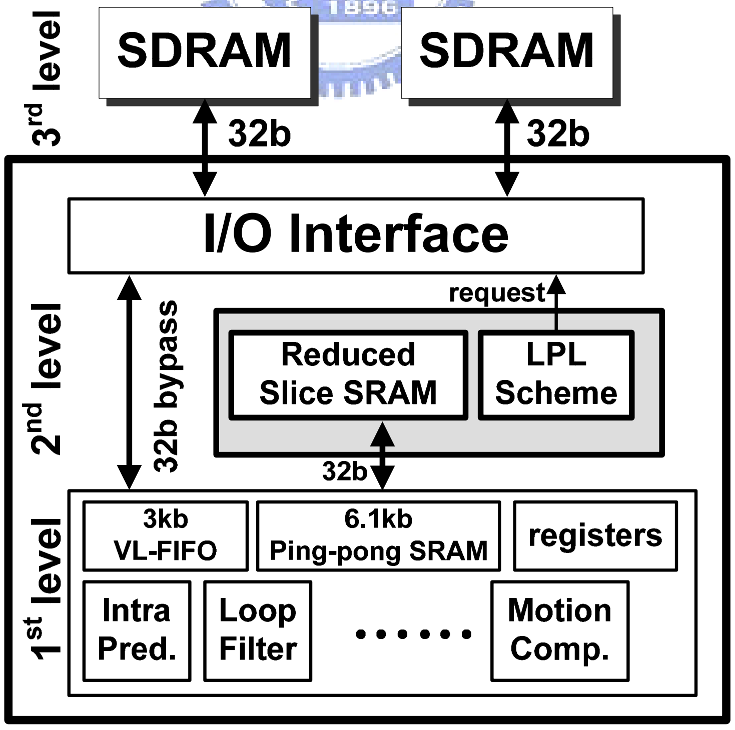
3.3.1 Background

While there has been much work studying memory performance for scientific and general-purpose applications, this dissertation focuses on the need of H.264/AVC video applications. H.264/AVC achieves high compression ratio since it adequately utilizes the neighboring pixels to obtain a reliable predictor and reduce the prediction errors. Compared to prevalent MPEG-x and H.26x video standards, H.264/AVC has a dependency on a long past history of data and need much intermediate storage. Therefore, this highly data correlation also leads to the design challenge on VLSI implementation in terms of memory power as well as cost.

Improving the memory hierarchy or reducing the embedded SRAM size is very effective for achieving low power dissipation because internal memories occupy about 70% of core power in H.264/AVC video decoder [45]. To reduce power consumption on memory modules, we aim at a memory hierarchy where copies of data from larger memories that exhibit high data-correlation are stored to additional layers of smaller memories. In this way, the great part of data accesses is moved to smaller memories and the significant power savings can be achieved since accesses to smaller level of memory hierarchy are less power consumption [46]. Thus, we propose three-level memory hierarchy named as content, slice memory and synchronous DRAM (SDRAM) as depicted in Figure 3.5. Figure 3.5(a) shows a three-level memory hierarchy where a slice SRAM is allocated for the storage with rows of pixels since H.264/AVC features to access logically adjacent pixels in the vertical direction. However, storing all pixels in rows of vertical pixels is unnecessary when the following decoding procedures are unrelated to the upper neighboring pixels. Hence, we further propose a line-pixel-lookahead (LPL) scheme in Figure 3.5(b) to eliminate the un-used pixels, leading to the reduction of memory size. In the following, we address the three-level of memory hierarchy and thereby illustrate how to exploit LPL scheme to further reduce the memory size.



(a)



(b)

Figure 3.5: The (a) conventional [43] and (b) proposed memory hierarchy.

3.3.2 Content Memory

A content memory, the first level of memory hierarchy, includes ping-pong SRAM and variable length FIFO (VL-FIFO) in Figure 3.6. Particularly, the ping-pong SRAM stores the unfiltered pixels in one 16×16 MB prior to the deblocking filter. Its data word size is 32-bit (4-pixel) and the address depth is decided by the YUV format (4:4:4, 4:2:2 or 4:2:0). For 4:2:0 format, there are 16 luma sub-blocks and 8 chroma sub-blocks. Moreover, it adopts the ping-pong structure and stores two macro-blocks (MBs) to resolve the structural hazard when reading and writing processes occur simultaneously. Hence, the content memory is of size (16+8)×4×32×2 (6.1k). On the other hand, VL-FIFO stores the pixels value with a size of one macroblock and is required to coordinate different data paths. In H.264/AVC, the prediction path (e.g. intra prediction or motion compensation) and residual path (e.g. CAVLC/IZ/IQ/IDCT) produces the output data with different timing budgets. The synchronization problem is introduced since both of them should be synchronously added in a pixel-wise manner. We develop a VL-FIFO to synchronize two paths prior to the deblocking filter. As a result, they construct the first level of memory hierarchy in the H.264/AVC decoding system.

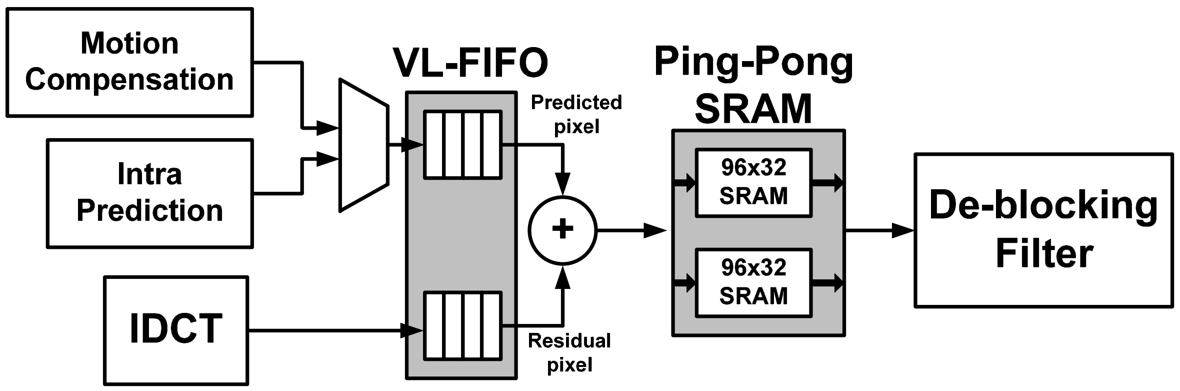
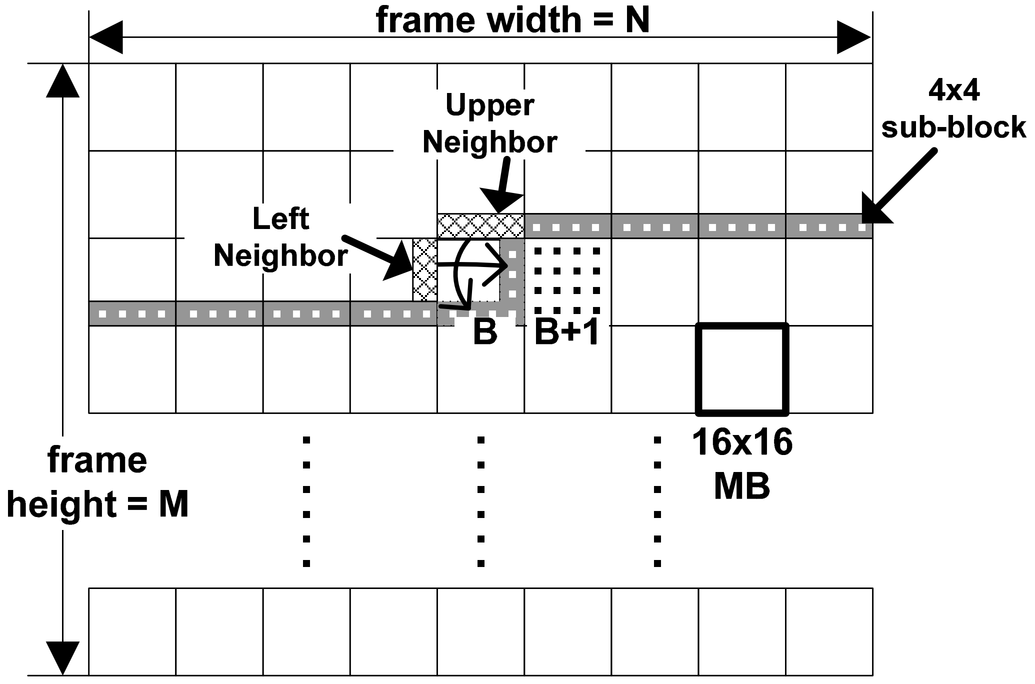


Figure 3.6: VL-FIFO and ping-pong SRAM in the content memory.

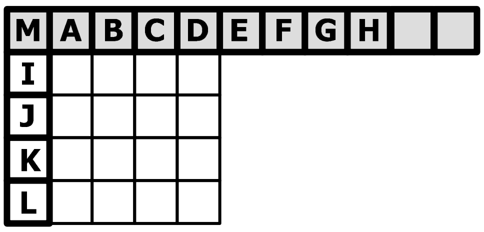
3.3.3 Slice Memory

A slice memory, the second level of memory hierarchy, is required to store neighboring pixels and prevent the data re-access from SDRAM since current decoded pixels in H.264/AVC relate to the pixels previously decoded. Considering a frame size of N×M in Figure 3.7(a), each square represents the 16×16 MB. Each MB contains 16 points and 4×4 pixels within each point. When following decoding procedures are performed from the MB index B to B+1, the pixel data within upper and left neighbors will be updated as the arrows indicate. Therefore, the shaded region should be kept when the decoding index is B+1. To give a specific explanation, we take intra prediction and deblocking filter as an example. As we know, intra predicted pixels rely on the upper one-pixel for a reliable predictor when near-vertical prediction modes apply. As a result, pixels M and A~H should be kept when decoding this 4×4 sub-block in Figure 3.7(b). Figure 3.7(c) indicates that deblocking filter requires neighboring four-pixel (i.e. p0~p3) to execute filtering procedures. In sum, the intra prediction and deblocking filter require one-row and four-row of pixels which construct the data organization in the slice memory.

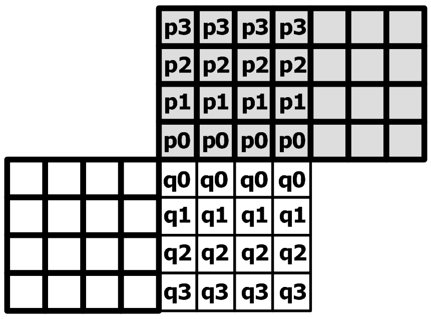
In addition to the intra prediction and deblocking filter, several modules require neighboring pixels or syntax to construct the results, such as CAVLC, CABAC, motion compensation, etc. Table 3.3 exhibits overall data organization in the slice memory. It consists of pixel data or syntax flags per unit of one pixel, 4×4 sub-blocks or 16×16 MB. In the pixel data, the size of slice memory depends on the frame width *W* as the shaded region of Figure 3.7(a) indicates. Moreover, the size of pixel relates to the chrominance format *C.F.* and pixel depth. Here, a 4:2:0 format and 8bits/pixel is assumed for simplicity. As for syntax flags, it is required keeping the flag since the decoding procedures of H.264/AVC will reference the previously decoded neighboring syntax flags. For instance, motion vector will be generated from the previously decoded one. Besides, H.264/AVC develops adaptive entropy coding method to achieve high compression ratio. In particular, CAVLC adaptively selects coding table by *nC* calculated from the *Coeff_Tokens* of neighboring blocks. CABAC adaptively estimates a large number of conditional probabilities through the neighboring syntax flags such as *MB_Type*, *CBP* …etc. These adaptive schemes lead to the highly data dependency and extra data storage for hardware implementation. In sum, the total memory size achieves 190.8kb under the H.264/AVC video decoding with 1080HD resolution and 4:2:0 chrominance formats.



(a)



(b)



(c)

Figure 3.7: The (a) slice memory in (b) intra prediction and (c) deblocking filter.

Table 3.3: Data organization in slice memory.

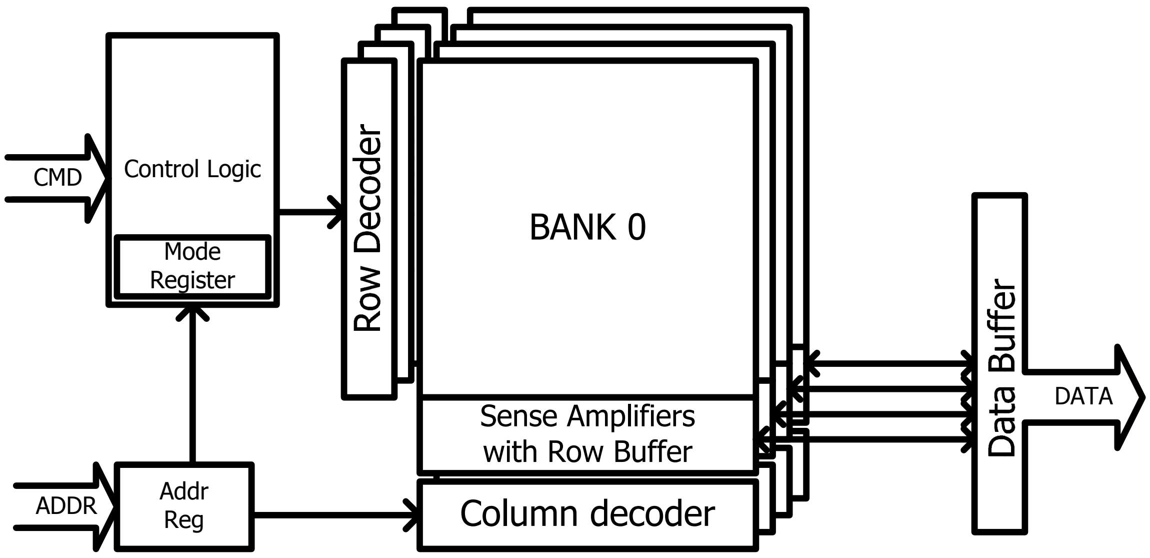
Level (data per unit)	***Content***	***Memory Size***	
		Formula(bits) W= Frame Width	***1080HD 4:2:0***
Pixel	Deblocking filter's upper neighbor *pixel*	4×W×C.F.[9]×8	122.8Kb
	Intra predictor's upper neighbor *pixel*	W×C.F.×8	30.7Kb
Sub-block	Motion vector's upper neighbor *flag*	2×(W/4)×10	9.6Kb
	nC's (CAVLC) upper neighbor *flag*	(W/4)×5	2.4Kb
Macroblock	CBP, MBType, ..etc (CABAC) upper neighbor *flag*	(W/16)×10×16	19.2Kb
		Total	184.7Kb

[9] C.F. means chrominance format.
{4:4:4, 4:2:2, 4:2:0} → C.F. = {3, 2, 2}

3.3.4 Synchronous DRAM

A synchronous DRAM (SDRAM), the third level of memory hierarchy, is widely used for storing frame data. A 64Mb quad-bank SDRAM model adopted is Micron's MT48LC2M32B2 [47]. A simplified architecture is shown in Figure 3.8(a) where four banks share the address and command buses and each bank has individual row decoder, sense amplifier, and column decoder. The mode register stores several SDRAM operation modes including burst length (BL), column address strobe (CAS) latency (abbreviated as CL), or burst type (sequential/interleave). The content of mode register updates according to commands issued from address buses. In sum, SDRAM can be treated as a 3-D structure with the dimensions of bank, row, and column.

On the other hand, Figure 3.8(b) lists the features and configurations of this SDRAM model. This SDRAM uses an internal pipelined architecture to achieve high-speed operation and is a dynamic random access memory containing 67,108,864-bit. It is internally configured as a quad-bank DRAM with a synchronous interface. Each of the 16,777,216-bit is organized as 2,048 rows by 256 columns by 32 bits. Moreover, read and write accesses to the SDRAM are burst oriented; accesses start at a selected location and continue for a programmed number of locations. This SDRAM provides programmable READ and WRITE burst lengths of 1, 2, 4, or 8 locations, or the full page, with a burst terminate option. This SDRAM offer substantial advances in DRAM operating performance including high-speed and low-cost implementation. Finally, we adopt it as the external frame buffer for subsequent frame decoding in H.264/AVC.



(a)

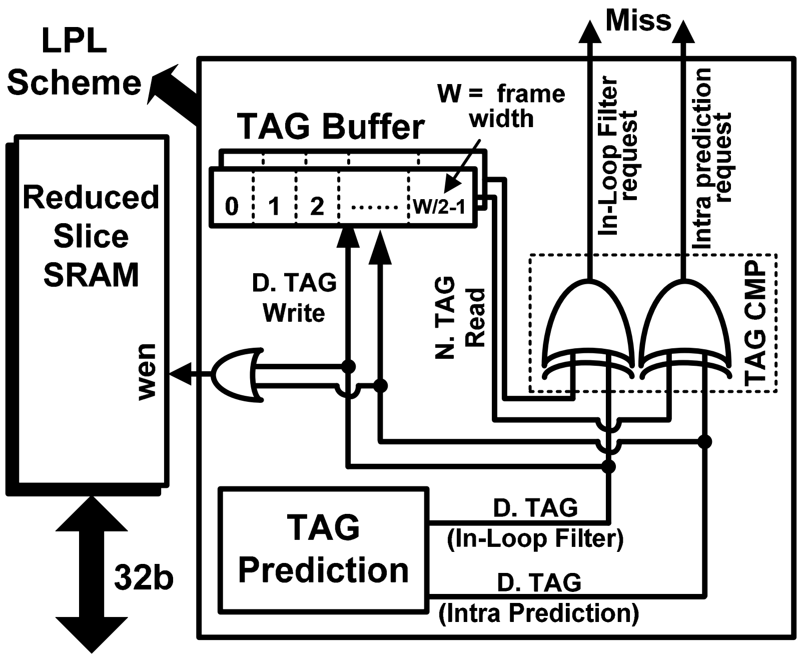
MT48LC2M32B2	**2Meg x 32**
Configuration	512K x 32 x 4banks
Row Addressing	2K (A_0 ~ A_{10})
Column Addressing	256 (A_0 ~ A_7)
Bank Addressing	4 (BA_0, BA_1)
CAS Latency (CL)	1, 2, and 3
Burst Length	1, 2, 4, 8, or full page
Timing (Cycle Time)	5ns (200MHz)
	5.5ns (183MHz)
	6ns (166MHz)
	7ns (143MHz)

(b)

Figure 3.8: (a) Simplified architecture and (b) configuration of a 4-bank SDRAM.

3.3.5 Line-Pixel-Lookahead Method

Line-Pixel-Lookahead (LPL) scheme is proposed to exploit spatial pixel locality in vertical direction and looks ahead before decoding the next line of pixel in order to improve the access efficiency. Figure 3.9 depicts the LPL scheme and the related pseudo codes. In particular, a reduced slice SRAM caches the pixels of upper neighbors, and a LPL scheme predicts whether the follow-up pixel data should be kept or not. In the scope of this LPL scheme, we only focus on the pixel-level of storage content since it occupied a large portion of overall slice memory in Table 3.3. Therefore, only deblocking filter and intra prediction will be considered in this dissertation. In the LPL scheme, the TAG prediction issues a decoding *TAG* (*D. TAG*) that contains a pair of signals for the purpose of deblocking filter and intra prediction units, and the *D. TAG* is equal to the neighboring TAG (*N. TAG*) after buffering one row of TAGs. Two W/2-bit TAG buffers record each *D. TAG*, where *W* means frame width. A TAG CMP (compare) unit perceives the contrast between *N. TAG* and *D. TAG*. A prediction miss will be noticed via a *request* signal from the output of TAG CMP when current *D. TAG* differs from *N. TAG*.



(a)

```
for(i=0;i<W/4;i++)
{  Step1: Generate decoding tag and buffered.
   D. TAG(i) = F(pre_mode, bS);
   Step2: write the D. TAG into Buffer
            and Slice memory.    }
   Step3: when encountering the next line of
         pixel.
for(j=0;j<W/4;j++)
{  Step4: reading the N. TAG data from TAG
         buffer.
   N. TAG(j) = (if requiring upper
                  neighbor) ? 1:0;
   Step5: comparing the TAG value.
   if(N. TAG(j) == D. TAG(j)) req. = 0; (hit)
   else   req. = 1; (i.e. miss)
}
```

(b)

Figure 3.9: The (a) data flow and (b) related pseudo code in the LPL scheme.

To illustrate how the LPL scheme works, we partition it into two steps: 1) TAG prediction and 2) TAG buffer. Figure 3.10 depicts the detailed circuit of the TAG prediction. The prediction step forecasts data accesses needed in advance, so a specific piece of data is pre-stored in the SRAM before it is actually desired by the follow-up decoding processes. A key observation is that not all upper neighboring pixels need to be pre-stored when they are determined as a “near-horizontal prediction mode” in intra prediction or a “SKIP mode” in deblocking filter. To realize the above behavior, we extract intra prediction mode, boundary strength (*bS*) and related header information from syntax parser as arrowed input signals. Each non-arrowed input is hard-coded and can be referenced by H.264/AVC [1]. In the case of a TAG buffer, it is of size 2W-bit and implemented by a single read and write port register file. The *N. TAG* will be read first from TAG buffer for the TAG comparison. Afterward, the *D. TAG* signal writes into TAG buffer for follow-up operations in next rows. Both reading and writing procedures are activated in different time slots to ensure that the

TAG signal previously written to the buffer can be read back without error.

Figure 3.11 describes the 4×4 intra prediction behavior of the LPL scheme through an example with a frame size of 48×32. Each square represents a 4×4 sub-block labeled by a 1-bit TAG signal. In the *N. TAG* field, we tag the 4×4 pixel data when a vertical prediction mode is applied. Furthermore, the un-tagged pixel will be discarded via *wen* (see the reduced slice SRAM in Figure 3.9(a)), resulting in reducing memory size. The memory word-length is fixed at 8-bit and a scaling factor *f* is introduced to scale the address depth at design time. Thus, the memory size is scalable and proportional to *W/f* (instead of *W* in Table 3.3) without degrading performance when the prediction hits (i.e. *D. TAG* equals *N. TAG*). However, an error of prediction may occur (i.e. miss) so we need to fetch the missed data from the external memory. The miss rate stands for a probability of missing events and is equal to the number of miss over one row of TAG. Although we reduce the memory size from *W*×8 to (*W/f*)×8 bits, 16.7% (i.e. 2/12) of miss rates are its penalties in this example. Therefore, it is indispensable to making a better compromise between the introduced miss rate and the scaling factor *f*.

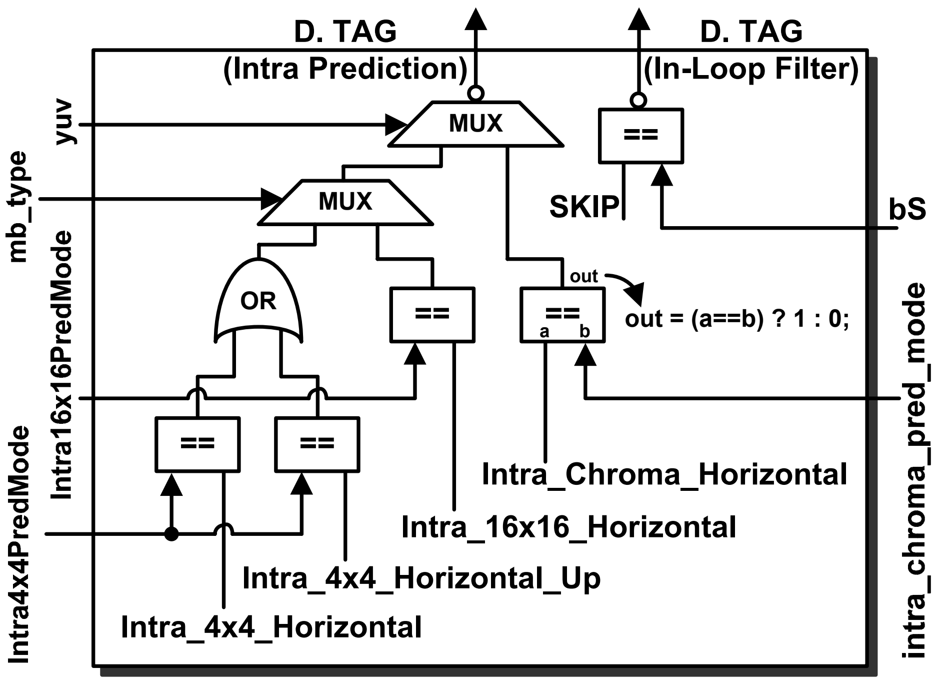


Figure 3.10: The proposed prediction circuit for TAG prediction.

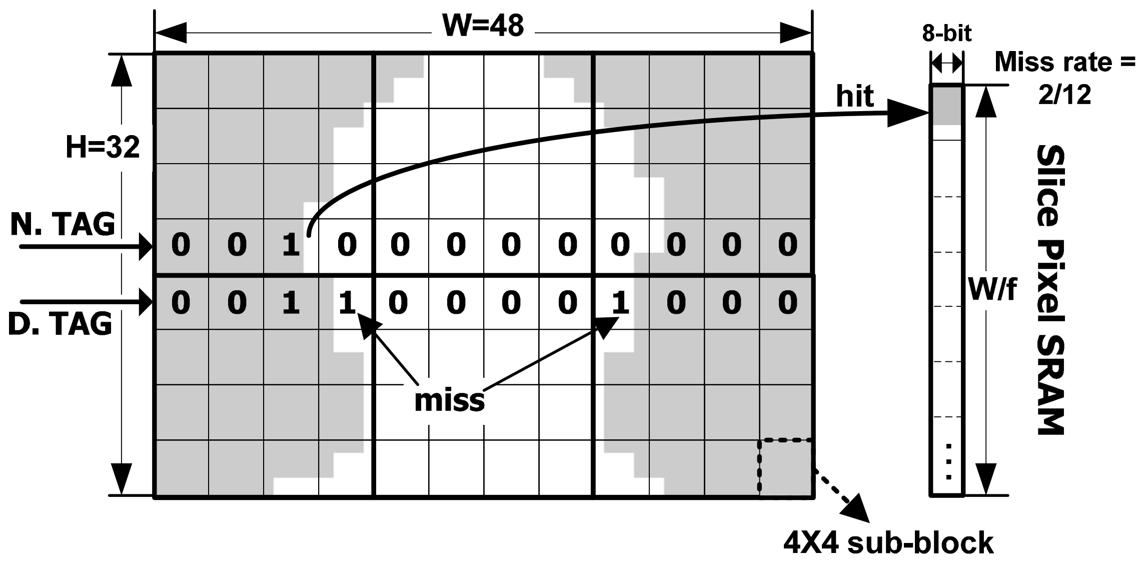


Figure 3.11: Data organization between slice pixel SRAM and TAG prediction.

3.3.6 Performance Evaluation

To apply the proposed LPL scheme to a practical video sequence, we first adopt two different types of video resolution to highlight the probability of missed rate and the reduction of external memory power. After that, we model the external memory power through the power calculator. It proves that the proposed three-level memory hierarchy greatly reduces the external memory power with a cost of slight internal memory power increment. Moreover, the increased memory power can be further reduced by LPL scheme.

3.3.6.1 Miss rate analysis

As compared to the original slice memory in Figure 3.5(a), the internal memory size can be reduced (see Figure 3.5(b)) with a penalty of slight miss rate increment. Because an error of prediction may occur, an additional penalty is introduced to re-fetch the missed data from external memory, leading to the increment of external memory bandwidth as well as power consumption. To alleviate this problem, we first analyze the relationship between the probability of miss events and external memory size. Figure 3.12 depicts the miss rate analysis with high-definition (1920×1088) and CIF (352×288) video resolution, and "αW" means that it's proportional to the frame width *W*. In this figure, we omit two extreme values and only plot the miss rate when scaling factor *f* ranges from 2 to 32. In particular, one extreme case is that miss rate equals zero when the factor *f* is fixed at one [48]. The other case occurs when the design [49] removes the second level of memory hierarchy (i.e. slice memory), leading to a 100% of miss rate. Hence, we can conclude that memory size and miss rate are often at odd and two extreme designs are far from a near-optimal value in terms of memory size and miss rate. Although increasing internal memory size achieves lower miss rate and external bandwidth, it suffers from the high memory cost and manufactured defects. Thus, inappropriate memory size will be harmful to the memory size and power consumption. In our design, we provide several design alternatives via factor *f* when developing a memory hierarchy at the design time. Therefore, we can achieve a better trade-off between miss rate and memory size. However, the goal of this section is to achieve low-power consumption through the memory hierarchy. To this end, we translate the miss rate into the memory bandwidth via Eq. (3.1) and Eq. (3.2). We adopt a memory power calculator to compute the memory power consumption from the available bandwidth in the next sub-section.

$$\text{\# of 4x4 sub-block accesses} = \text{\# of write accesses} + \text{\# of read accesses} \times \text{miss rate} \qquad (3.1)$$

$$\text{Bandwidth(Bytes/sec)} = \frac{\text{\# of 4x4 sub-block accesses}}{\text{frame}} \times 16 \frac{\text{bytes}}{\text{4x4 sub-block}} \times \frac{\text{frame}}{\text{sec}} \qquad (3.2)$$

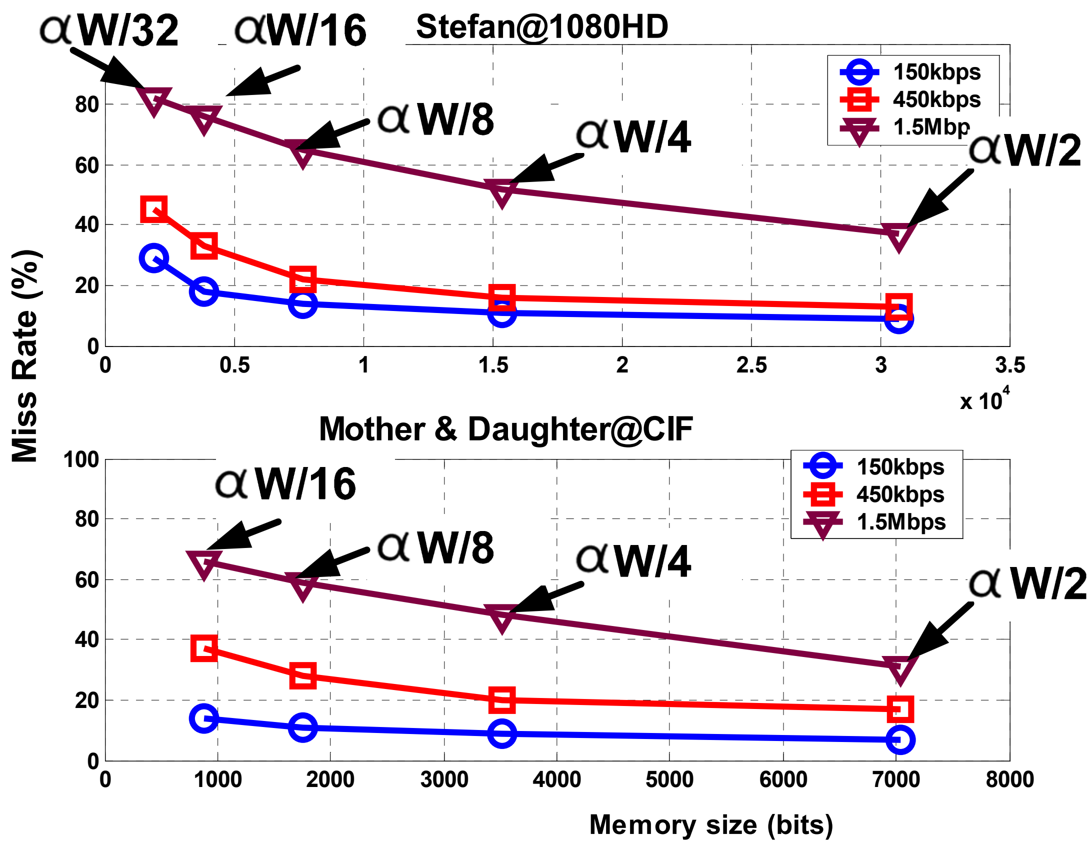


Figure 3.12: Miss rate analysis with different scaling factors *f*.

3.3.6.2 Memory power modeling

To analyze the power consumption of proposed memory hierarchy, we describe the power modeling through a memory power calculator. In general, the power consumption of memory hierarchy can be considered as a summation of on-chip and off-chip memory power in Eq. (3.3). Furthermore, memory accesses are the most power consuming operation [50] and dominate the overall power budget on the internal memory. The power consumed on memory accesses is a function of memory size, the access frequency and technology etc. In the following, we assumed that the on-chip power is linearly proportional to the access frequency [51] and the memory size. Therefore, the simplified power model described by Eq. (3.4) suffices for the purpose of evaluating the power effect on memory hierarchy.

As for off-chip memory, the power modeling becomes more complicated. Not only data access but also I/O and background power should be concerned (see Eq. (3.3)). Specifically, we adopt Micron's system-power calculator [52] to model DRAM power. To estimate the power consumption, it is necessary to understand the basic functionality of DRAM in Figure 3.8. In particular, once the clock enables, commands can be sent to the DRAM module. Typically, the first command is ACTIVE (*ACT*). The *ACT* command selects a bank and transfers the cell data into the sense amplifier. The data stays in the sense amplifiers until a PRECHARGE (*PRE*) command to the same bank restores the data to the cells in the array. When data is stored in the sense amplifier, *READ* and *WRITE* commands may take place. When a *READ* command issues, the data is driven through the I/O gating to the internal read latch. Once in the latch, it is multiplexed onto the output drivers. Moreover, the REFRESH (*REF*) operation is normally distributed evenly over time, because the memory cells of a DRAM store the data information in small capacitors that lose their charge over time. Based on the aforementioned behavior, we can conclude that DRAM power mainly comes from the operations in access, I/O and background modes. The access power dissipation comes from the *READ* and *WRITE* commands. I/O power consumes on I/O pins (i.e. DQ) when *READ* commands issue. Furthermore, *PRE* command in standby and power-down modes and *REF* commands contribute the power dissipation of background operations. Therefore, based on the power modeling in both on-chip and off-chip memories, we will show the simulation results as follows.

$$P_{total} = P_{on-chip} + P_{off-chip} = P_{access} + (P_{access} + P_{IO} + P_{BG}) \tag{3.3}$$

$$P_{on-chip} = P_{access} = f_{access} \times F\left(Length_{word}, \#\ of\ word, V_{dd}\right) \tag{3.4}$$

As mentioned before, we exploit memory size and power calculator as the SRAM and DRAM power indexes. An observation is that the curve between internal SRAM and external DRAM power consumption is shown in Figure 3.13. Let's illustrate this property by choosing Micron's SDRAM model (i.e. MT48LC2M32B2) with CAS latency = 2, BL = 1 and ^{t}CK = 7ns. In X-axis, there are several design alternatives according to the factor *f*. In other words, we provide a scalable solution with a trade-off at the architectural design time. As a result, a better compromise can be constrained by the minimal distance from origin (i.e. *f* = 8) since it achieves smaller SRAM size as well as SDRAM power dissipation. Note that this property can also be applied to DDR/DDR2 memory for high-resolution video decoding when adopting the proposed memory hierarchy with LPL scheme.

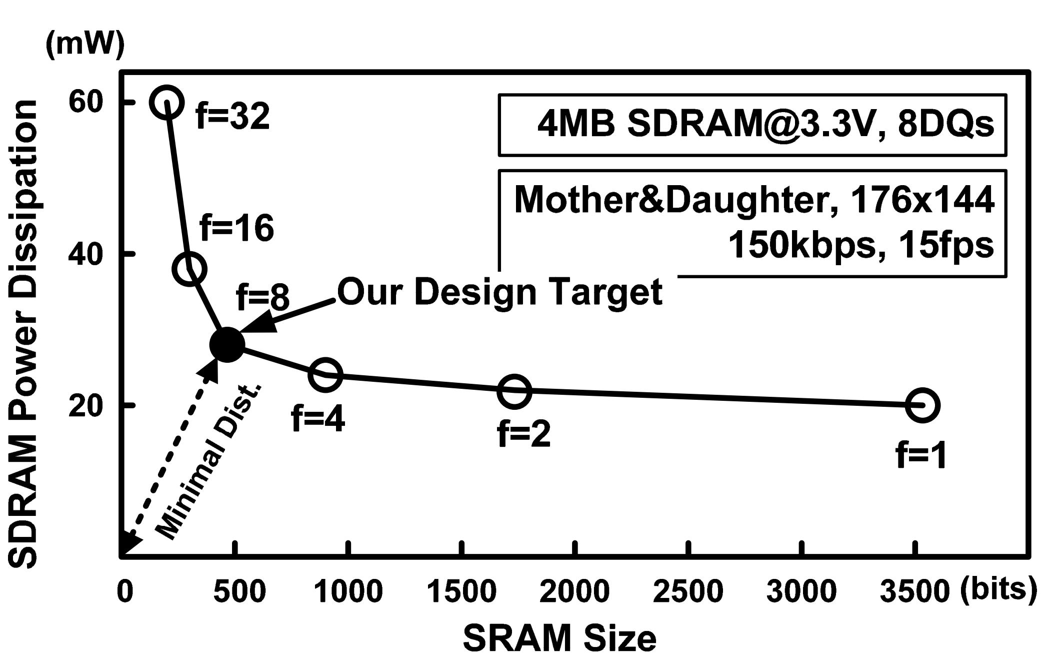
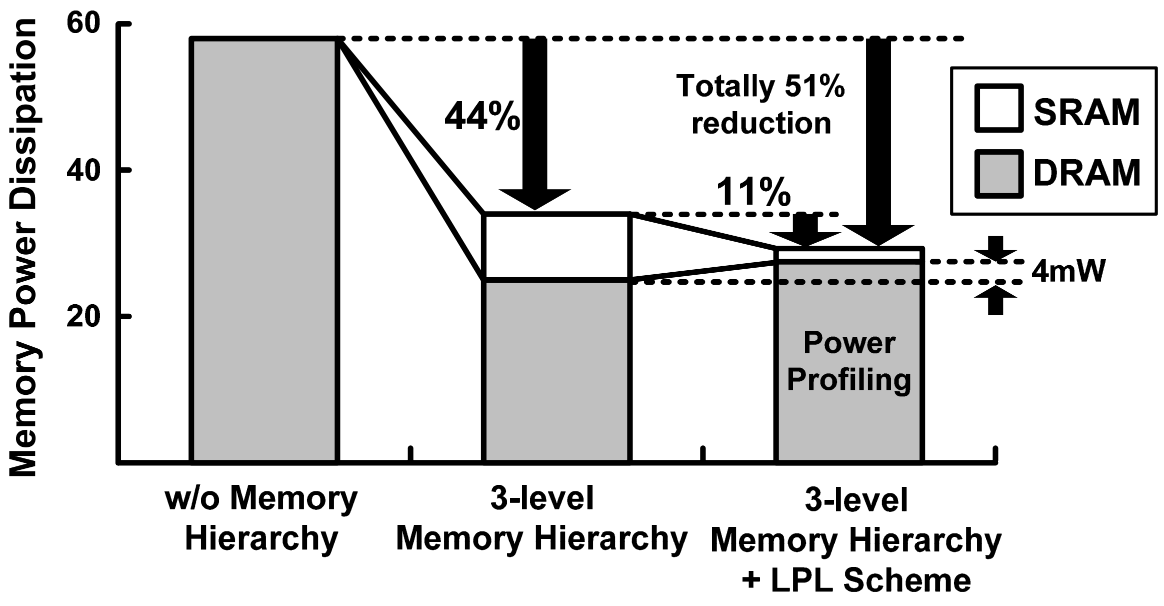
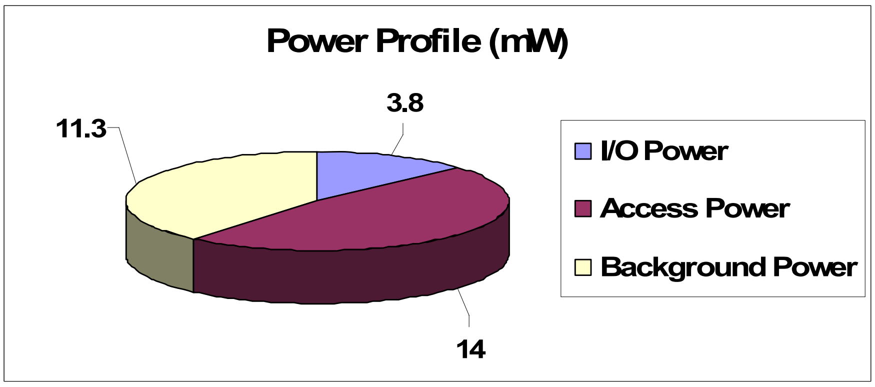


Figure 3.13: Analysis of power dissipation on external SDRAM and internal SRAM.

Figure 3.14 shows a power saving through the three-level memory hierarchy with the LPL scheme. The traditional three-level memory hierarchy [43] in middle bar is just a special case when a scaling factor *f* equals one. While it reduces power dissipation by 44%, the SRAM power penalty is considerably high. To further reduce the SRAM power consumption, we propose the LPL scheme to make a better compromise from the observation in Figure 3.13. Although the right-hand bar increases the SDRAM power by 4mW, the SRAM power in the right-hand bar can be greatly reduced to $\frac{1}{8}$ $(\because f = 8)$ of that in the middle bar. Moreover, introduced gate count for LPL scheme is very small and occupies only 4% of system area. Hence, the LPL scheme further gains 11% power dissipation. Altogether, three-level memory hierarchy and LPL scheme achieve 51% power saving. To further report the power consumption in the DRAM module, Figure 3.14(b) exhibits the power profiling and shows that there are 14mW of access power excessively consumed in SDRAM. This is because motion compensation requires large read access times from external memory. Additionally, it should be noted that the metric of SDRAM's power consumption can be further optimized through mobile DRAM [53] or other leading-edge DRAM designs [54]-[56]. In our simulation, we just choose a preliminary Synchronous DRAM to verify the power performance in a system point of view. Therefore, the power improvement will become more prominent when the low-power DRAM module can be applied.



(a)



(b)

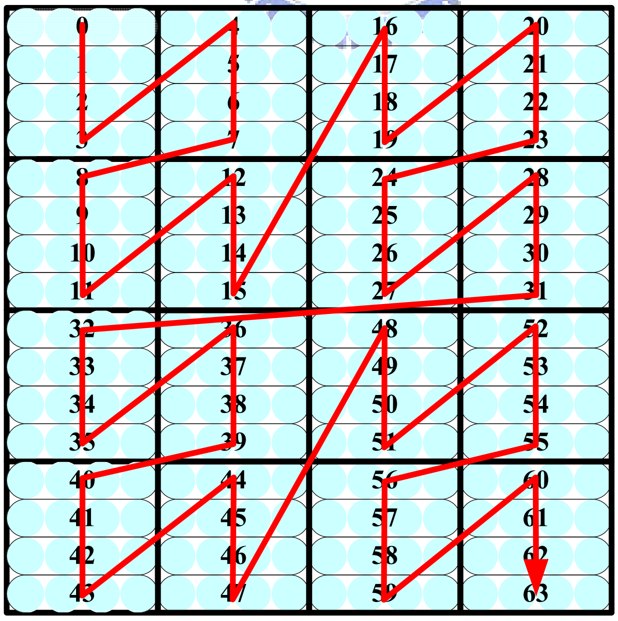
Figure 3.14: (a) Power saving on memory hierarchy and (b) DRAM power profiling.

3.4 *Eliminating Memory Access Times*

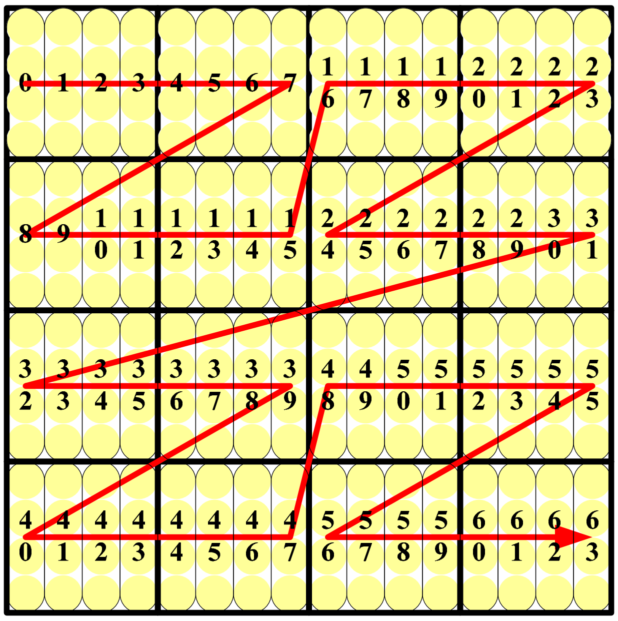
Under an identical design specification, reduction of memory-access times leads to operations with lower system clock rate and voltage and thus lower power consumption. Based on the complexity profiling of H.264/AVC joint model (JM) software via Intel® VTune performance analyzer [57], both motion compensation and de-blocking filter are the most computation-intensive operations in H.264/AVC [58]. To lower the required working frequency for low-power demands, we first analyze the decoding ordering which affects the access efficiency in both intra and inter prediction modules. Then, we propose new architectures to reduce both processing cycles and memory access times in computationally critical modules of H.264/AVC.

3.4.1 1×4 and 4×1 Decoding Ordering

According to the decoding flow in H.264/AVC specification [6], a 4×4 sub-block is the smallest processing unit. To efficiently transfer 4×4 pixel data among modules, we discuss two strategies to organize the word-length for data transactions. One is a 4×1 row-by-row decoding ordering, and the other is a 1×4 column-by-column decoding ordering. Although the 1×4 is similar to the 4×1 within a 4×4 sub-block, the decoding orders in a 16×16 macroblock (MB) are widely different. Figure 3.15(a) and (b) show the 4×1 and 1×4 decoding orderings in one MB, respectively. Compared to the 4×1 row-by-row decoding ordering, the 1×4 column-by-column decoding ordering provides a better data structure, reducing the processing cycles on intra and inter prediction modules. For example, an intra prediction module requires left neighboring pixels to spatially create the predicted pixels. These neighboring pixels can be easily fetched through the 1×4 ordering since they are organized in a column-wise manner. As for the inter prediction module, extra initialization cycles are required when the thread in Figure 3.15 makes a turn. Therefore, the access times are related to the frequency of turning events. The results proved that the 1×4 column-by-column decoding ordering reuses the neighboring pixels and reduces the turning events, yielding the cycle reduction of 17% and 28% in intra and inter prediction modules respectively [59].



(a)



(b)

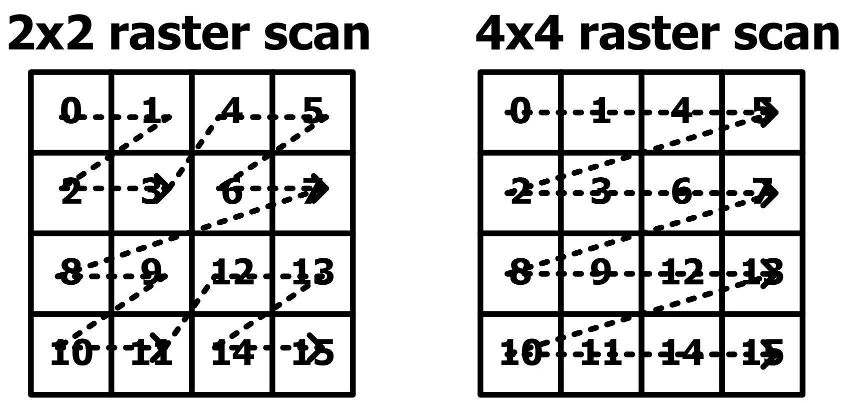
Figure 3.15: Different data orders in (a) 4×1 and (b) 1×4 decoding orders.

3.4.2 Motion Compensation

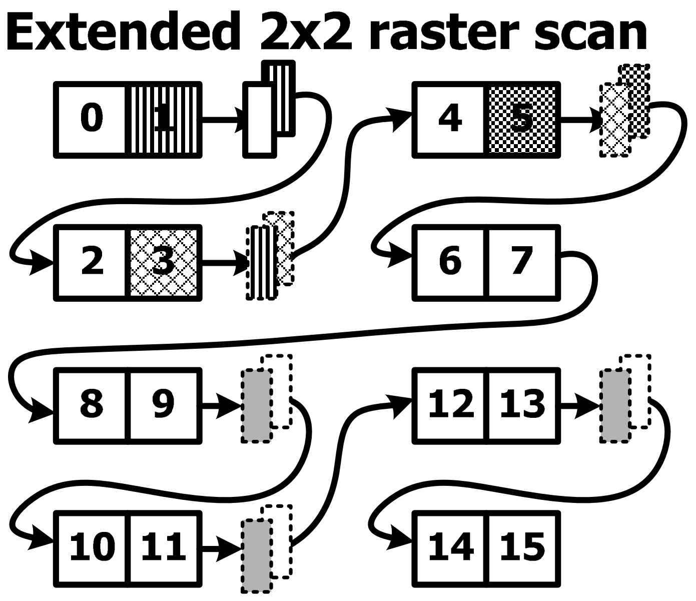
It is obvious that reducing the processing cycles in the motion compensation (MC) module can improve the system performance since it is a system bottleneck compared to other modules. In particular, the interpolation unit in MC is always the most time-consuming module. In the following, we propose a horizontal-switch approach to reuse the neighboring pixels for the interpolator design so as to reduce the external memory access times. More detailed discussion about motion compensation can be found in the master thesis of my junior member [139].

3.4.2.1 Horizontal-switch technique

Interpolation unit is always the most time-consuming module in whole motion compensation core. A great deal of memory accesses degrade decoding throughput especially in the features of variable block size and quarter-pel resolution. To reduce memory access times, it is necessary to increase data reuse probability for overlapped regions of neighboring interpolation windows. To this end, we apply a data buffer to each shift register for luma interpolators. Furthermore, a chroma interpolator has a similar behavior with a luma interpolator and both of them can be simplified into a multiplication-free design approach [30]. Specifically, in Figure 3.16(a)(b), a dotted line shows the different scan orders in one luma macroblock, and the number in each square represents the decoding orders defined by H.264/AVC [6]. In Figure 3.16(a), a 2×2 raster scan is compliant to H.264/AVC, but extra cycles for data initialization are required when the dotted line turns. Compared to the 2×2 raster scan, a 4×4 raster scan features less turning events but violates H.264/AVC standard. Since standard-limitation and cycle-efficiency are often at odds, an extended 2×2 raster scan has been proposed to improve decoding performance in Figure 3.16(c). In particular, content registers, 6×9 pixel buffers, attached to shift registers for the interpolator are adopted. When sub-block #1 is decoded, overlapped windows in the right-hand side are stored into background of pixel buffers. These buffers switch into foreground when decoding index moves to sub-block #3. Therefore, in the decoding of sub-block #4, the left overlapped window can be reused from the content registers instead of external memory, and the overall processing cycles can be reduced. Compared to the conventional design [41], the proposed motion compensation requires additional 6×9 pixel buffers (1% cost of MC) but saves 30% of access times.



(a) (b)



(c)

Figure 3.16: Different scan orders in motion compensation: (a) 2×2 raster scan; (b) 4×4 raster scan; (c) extended 2x2 raster scan.

3.4.2.2 Efficient Memory Interface

Although we increase the reuse probability to reduce the access frequency to the external memory, the external memory interface has to cooperate with the MC to improve the overall access efficiency. In our design, two external frame memories are allowed for writing decoded data and reading reference data reciprocally at the same time. Compared with SRAM, SDRAM is adopted due to the cost and power issues. However, SDRAM also introduces the longer access latency and degrades the decoding throughput because of the internal pipeline architectures and 3D structure of banks, rows and column characteristics (see Figure 3.8). To solve the above problems, an efficient memory interface is introduced to overlap and re-schedule each access commands to improve the bandwidth utilization. A 64Mb quad-bank SDRAM model adopted is Micron's MT48LC2M32B2 [47] and Figure 3.17 illustrates our memory interface design. This interface is composed of the bank controller, memory scheduler, and several read/write buffers. Each bank controller generates suitable commands for read/write processes. The memory scheduler collects these commands then sends re-scheduled commands to external SDRAM. Read and write data buffers store burst data, and read/write command queues are designed to hold successive commands. The read and write processes are activated in a parallel fashion and switch at frame levels. Based on this memory interface, the experimental results exhibit that the processing cycles per MB for QCIF@30fps decoding range from 288 to 512 for various video sequences.

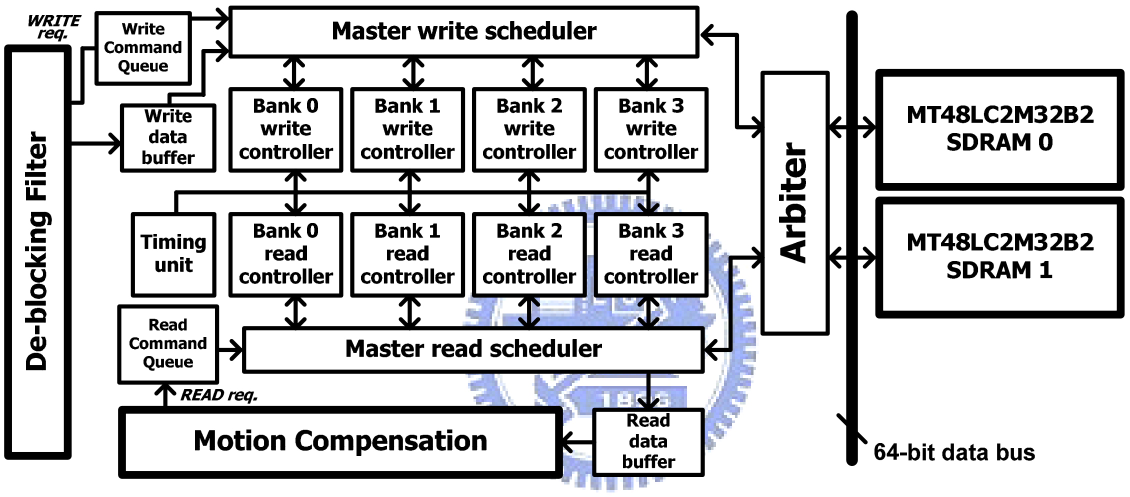


Figure 3.17: A block diagram of the proposed memory controller.

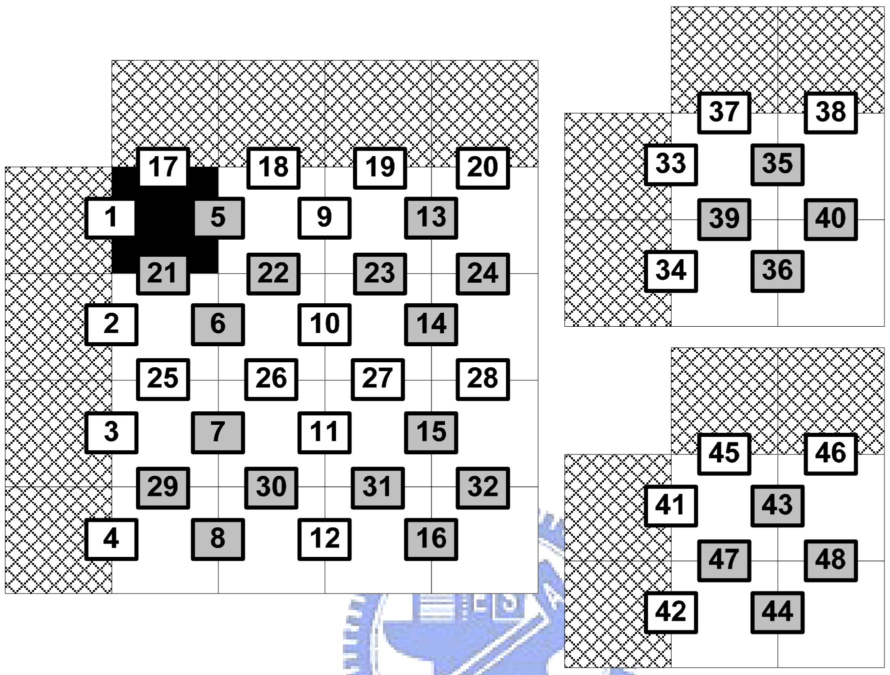
3.4.3 Deblocking Filter

In addition to the motion compensation, deblocking filter should be considered in terms of complexity and memory accesses. In general, deblocking filter contributes about one-third of the computational complexity at the H.264/AVC video decoder [137]. It operates each filtering process on the 4×4 boundaries instead of the 8×8 boundaries in filters of H.263 or MPEG-4 video standards. Therefore, a large number of memory accesses are its penalty for the low-power portable applications. In the following, we present a hybrid filtering schedule to reduce the access frequency to memory modules for improving the access efficiency.

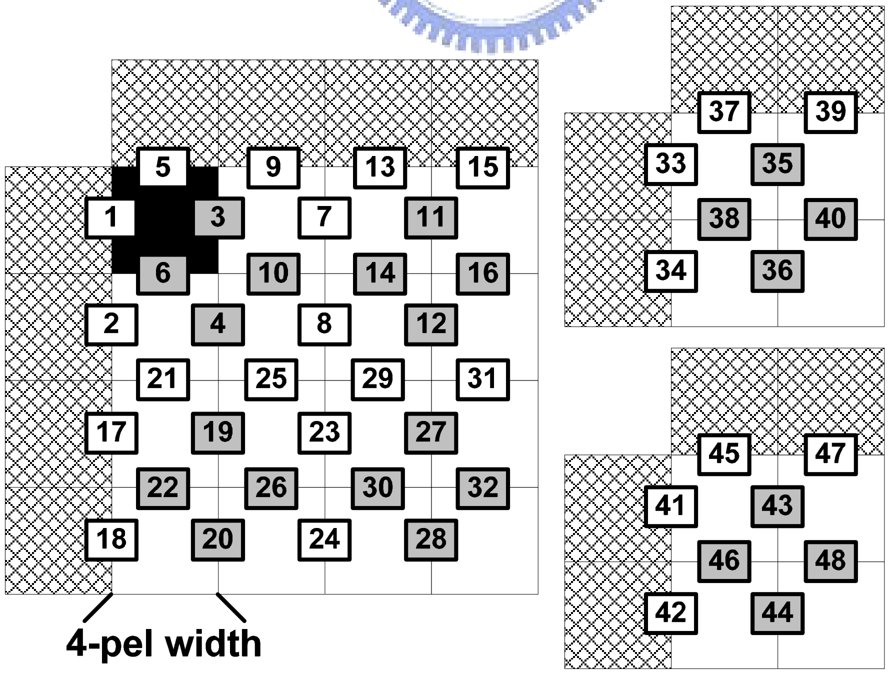
3.4.3.1 Hybrid Filtering Schedule

We propose a hybrid filtering schedule to reuse the intermediate data and thereby eliminate the additional memory accesses when deblocking filters change the filtered edges from vertical to horizontal direction. It reduces one-half of processing cycles with slight increment of buffer cost. Figure 3.18(a) describes the filtering order where vertical edges are filtered first, followed by horizontal edges. Each square is sized to 4×4 pixel data. The number within rectangles represents the processing order in one luma macroblock. A direct approach may induce a drawback that intermediate data have to be stored and loaded again when altering the filtered directions. For example, considering the gray region, the edge #1 will be filtered first followed by the edge #5. After that, the processing data in gray region cannot be reused since the distance between vertical and horizontal edges (i.e. #5 vs. #17) becomes longer. Therefore, the memory accesses are required in both vertical and horizontal directions. To cope with this problem, we propose a hybrid scheduler without affecting the standard-defined data dependency in Figure 3.18(b) where un-shaded numbers are carried out in 8×8 post-loop filters. Considering the orders in the black region to perceive a contrast, the black region can be reused because the orders between different directions become close. Therefore, the proposal prevents the data re-access for different directions and reuses the intermediate pixels to reduce the processing cycles.

Though B. Sheng *et al.* [60] also proposed a novel schedule to reduce the processing cycles, this schedule requires eight 4×4 sub-blocks as the kept buffers. To reduce this buffer size, we partition each MB into two main parts (i.e. Deblocking Filter-MB-Upper or Lower) in Figure 3.19(a), and each part is composed of eight time indices to realize the filtering procedure in Figure 3.19(b). Each bold line represents the edge to be filtered in the corresponding time index. As a result, our kept buffer size is reduced to four 4×4 sub-blocks where is located on shaded regions. By the same way, the proposed schedule is performed on the chroma MB as well.



(a)



(b)

Figure 3.18: Filtering orders in (a) standard-defined and (b) proposed filtering schedule.

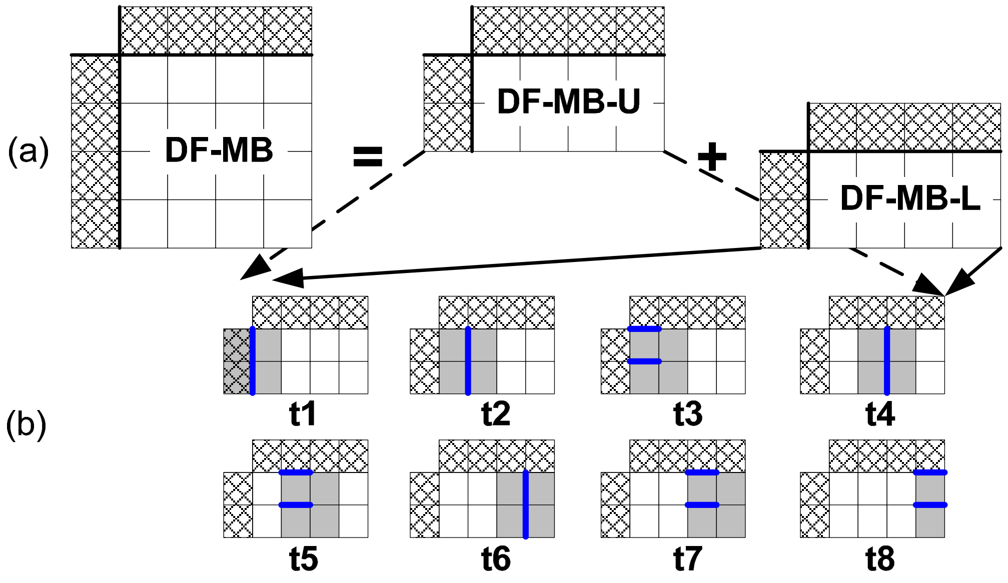


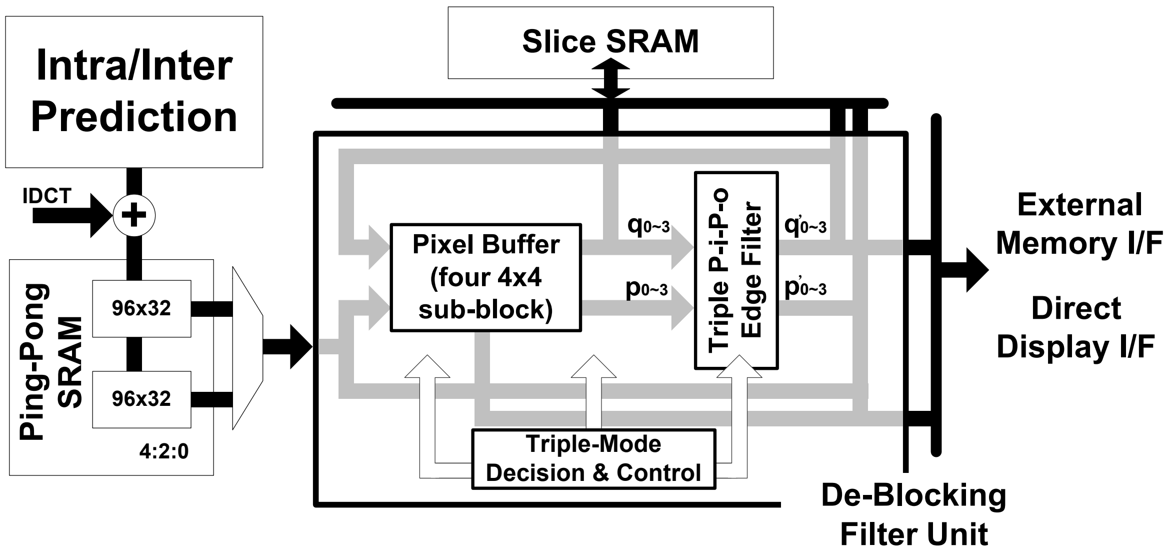
Figure 3.19: The (a) partitioned MB and (b) each time index for hybrid filtering schedule.

To apply the hybrid filtering schedule to a dedicated hardware, a new architecture of deblocking filter is presented in this sub-section. Figure 3.20(a) shows the proposed design with block diagram and data flow representations. Specifically, deblocking filter receives the data from ping-pong SRAM (see Figure 3.6), extracts the neighboring pixel from slice SRAM and writes the filtering results into external frame buffer. A detailed illustration of ping-pong SRAM, slice SRAM and external memory have been made in Section 3.3. The shaded-arrows denote the data flow inside the de-blocking filter unit, and the black-arrows denote the data flow outside. The pixel buffer contains four 4×4 pixel values and is used to store the intermediate pixel value when applying the proposed hybrid filtering schedule.

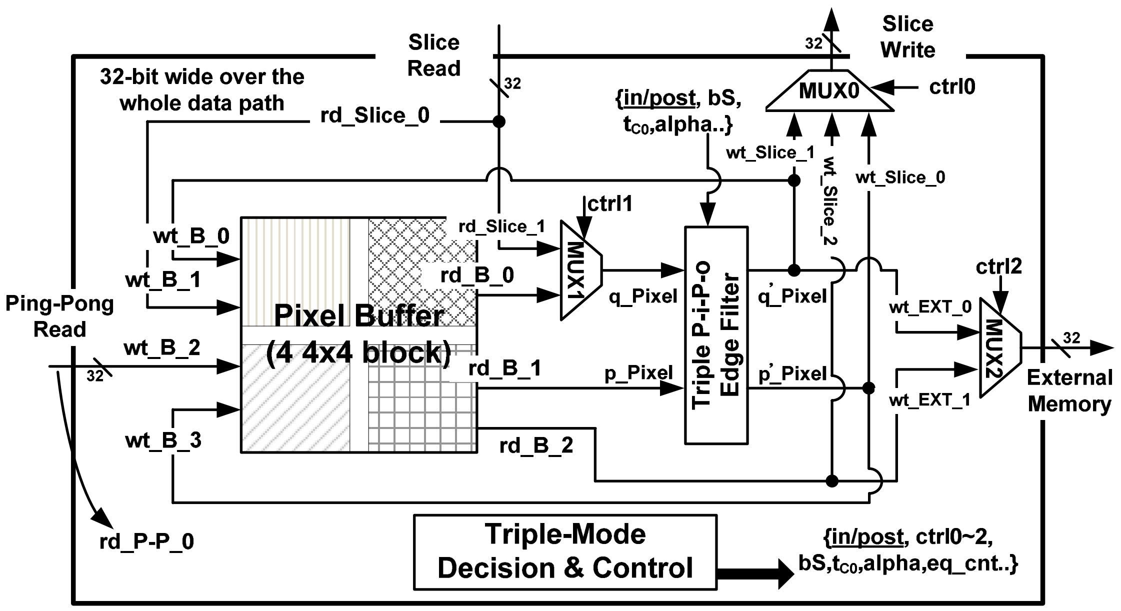
The detailed architecture of deblocking filters has been redrawn in Figure 3.20(b). All signals are 32-bit wide and possess the 1×4 row-by-row data organization. There are four input signals {*wt_B_0*, *wt_B_1*, *wt_B_2*, *wt_B_3*} to write the buffers with four 4×4 sub-blocks. Further, there are three output signals {*rd_B_0*, *rd_B_1*, *rd_B_2*} to perform the edge filter and then write to the frame memory, pixel buffer or slice SRAM. By the same naming rule, each data flow represents the writing/reading to/from the storage module including ping-pong (P-P) memory, slice SRAM, or external frame memory. Specifically, we use one MB with 48 edges of in-loop filter as an example to demonstrate the filtering behavior in Figure 3.20(b). It can be divided into two main parts:

- Write Process is a writing mechanism through the signal {*wt_Slice_0~2*, *wt_I/F_0~1*, *wt_B_0~3*}.
- Read Process is a reading mechanism through the signal {*rd_Slice_0~1*, *rd_P-P_0*, *rd_B_0~2*}.

The key idea of the high-throughput architecture is that the ping-pong memory is exploited only for the reading processes. In Figure 3.20(b), the writing signal, wt_Slice_0, is activated on the edges 6, 10, 14 and 16 because the lower sub-blocks become the upper neighboring sub-blocks of DF-MB_L (see Figure 3.19(a)). Therefore, the *wt_Slice* writes the filtering results into slice memory for follow-up filtering processes. For the *wt_EXT*, it writes filtered data into the external memory or display interface. For instance, *wt_EXT_0* will be activated on each filtering process on horizontal edges except for the edge of activated signal *wt_Slice_1* and *wt_B_0*, since *wt_EXT_0*, *wt_Slice_1* and *wt_B_0* have the same root-signal of *q'_Pixel*. On the other hand, the reading signal, *rd_Slice_0*, is activated on vertical edges while the *rd_Slice_1* is valid on horizontal edges. In addition, the *rd_P-P_0* directly feeds through the pixel buffers. In other words, P-P memory is employed for the reading processes, and there is no need to write the filtered results into the P-P memory in one direction and thereby read them in another direction. Therefore, the proposal exploits four 4×4 buffers to reuse the intermediate pixel and eliminate the writing accesses to the P-P memory.



(a)



(b)

Figure 3.20: The (a) block diagram and (b) architecture for the proposed deblocking filter.

3.4.3.2 Cycle Analysis

To clarify the cycle reduction of proposed hybrid filtering schedule, we formulate processing cycles in Eqs. (3.5) and (3.6) where *C.C.* means cycle counts. The overall cycles of deblocking filters can be considered as a combination of the pre-process, filter-process and post-process. The pre-process is an initial stage which loads neighboring pixel from external into slice memory while the post-process is a write-back stage which writes filtered results from slice memory to external memory. In the filter-process, the processing cycles include slice or ping-pong Memory to pixel Buffers (i.e. Mslice/pp-to-Bpixel), generic filtering, and pixel Buffers to slice Memory (i.e. Bpixel-to-Mslice). The processing cycles of the generic filtering are 4×(32+16) which become a lower bound to fulfill filtering processes if the rest of processing cycles are zero in an ideal case.

$$\begin{aligned}
&\text{Total Cycle Counts} = C.C._{\text{Pre-process}} + C.C._{\text{Filter-Process}} + C.C._{\text{Post-Process}} + C.C._{\text{misc.}} \\
&C.C._{\text{Pre-process}} = C.C._{\text{initial}} = C.C._{\text{Mexternal-to-Mslice}} \\
&C.C_{\text{Post-Process}} = C.C._{\text{write back}} = C.C._{\text{Mslice-to-Mexternal}}
\end{aligned} \tag{3.5}$$

$$\begin{aligned}
&C.C._{\text{Filter-Process}} = C.C._{\text{Luma Filter}} + C.C._{\text{Chroma Filter}} = C.C._{\text{Mslice/pp-to-Bpixel}} \\
&+C.C._{\text{generic}} + C.C._{\text{Bpixel-to-Mslice}},\ where\ C.C._{\text{generic}} = 32\times4+16\times4=192
\end{aligned} \tag{3.6}$$

Based on the proposed hybrid filtering schedule, the overall cycles are 243 and close to a lower bound of processing cycles. Based on the aforementioned three-level memory hierarchy, the neighboring pixel can be fetched from the slice memory instead of external DRAM, and the filtered results are written into the external memory without going through the slice memory. As a result, the cycle counts of the pre-process and post-process can be eliminated. In the filter-process stage, the evaluated cycle counts are 148 cycles for luma MB and 88 cycles for chroma MB. Specifically, we take 8 cycles (DF-MB-U + DF-MB-L) in the Mslice/pp-to-Bpixel stage. There are 4×32 cycles required to filter horizontal and vertical edges in a luma MB. Moreover, we need 12 cycles (i.e. Bpixel-to-Mslice) to write the filtered results for the edges {16,30,32}. Overall, we need 148 (i.e. 8+4×32+12) cycles to accomplish filtering processes in a luma MB. By the same analysis, we need 88 (i.e. 4+4×8+8=44 for each chroma) cycles in a chroma MB. Therefore, there are total 243 (ie. 148+88+7) cycles with extra 7 cycles for the data hazard. The cycle overheads in the control logic can be neglected since it acts as a pipelined fashion. In addition, the processing cycles of the post-loop filter are identical to that of the in-loop filter because they share the same architecture and control flows in Section 2.8.3. In conclusion, 243 cycles are close to a lower bound (192 cycles) by the proposed schedule. For clarity, we make a detailed comparison in Table 3.4. We choose single-port architecture [61] and worst case processing cycle [62] for a fair comparison. We don't consider an extreme case (e.g. group of picture: 1I+149P, skip mode occurred frequently) because all designs can eliminate the processing cycles on the skip mode through an extra add-on module. In sum, compared with the existing approaches [60]-[62], the proposed architecture saves about one-half of processing cycles per MB.

Table 3.4: The cycle analysis of the de-blocking filter.

Cycle Counts			**[61]**	**[62][10]**	**[60][11]**	**Proposed**
Vertical / Horizontal			Separate	Separate	Hybrid	Hybrid
Pre-process (Initial) stage			160	N/A	64	0
Filter-process Stage	Luma	Horizontal	128	106	128	148
		Vertical	200	106		
	Chroma	Horizontal	64	74	64	88
		Vertical	112	74		
Post-process (write-back) stage			160	N/A	160	0
Misc.			54	N/A	30	7
Total			878	360+N/A	446	243

[10] We only consider the worst case and exclude the effect of mode decision for a fair comparison.

[11] Authors didn't report the processing overheads between the internal memory and kept buffers.

To enhance the system performance, this VLSI solution for deblocking filter is designed to achieve reduced processing cycles. The proposal is implemented using a 0.18μm CMOS process. Excluding the internal memory, the synthesized gate counts are 21.1K which is reduced to 70% of the original design that realize in-loop or post-loop filtering process separately. Moreover, it achieves 4×10^5 MB/sec of throughput rates when operating at 100MHz. Finally, Table 3.5 reveals that the throughput rates of the proposed design are about 1.5~2.5 times larger than that of existing approaches [60]-[62].

Table 3.5: A hardware summary for the deblocking filter.

Cycle Counts	**[61]**	**[60]**	**[62]**	**Proposed**
Function	in-loop	in-loop	in-loop	in/post-loop
Kept Data Size	$2\times4\times4$ sub-blocks	$8\times4\times4$ sub-blocks	$2\times4\times4$ sub-blocks	$4\times4\times4$ sub-blocks
Memory Organization	1) $96\times32 + 64\times32$ 2) External MEM	1) $96\times32\times2 + 64\times32$ 2) External MEM	1) 96×32 2) External MEM	1) $96\times32\times2$ 2) $2(N+12)\times32$[12] 3) External MEM
Processing Cycles	878	446	360+N/A	243
Process	0.25μm	0.25μm	0.18μm	0.18μm
Gate Counts	20.66K	24K	11.8K	21.1K (in/post)
Clock Rate	100MHz	100MHz	100MHz	100MHz
Filtering Throughput[13]	1.6×10^5 MB/sec	2.2×10^5 MB/sec	2.6×10^5 MB/sec	4×10^5 MB/sec

[12] N: frame width

[13] Throughput: Macro-Block/sec = $\text{Clock Rate}/\text{Processing Cycles}$

3.4.4 Performance Evaluation

Let's make a brief summary in terms of processing cycles in Figure 3.21. In the beginning, we propose several techniques to reduce the processing cycles in the MC. They are the 1×4 decoding ordering, horizontal-switch method, and efficient memory interface. Therefore, the processing cycles are reduced by 37% as compared to conventional designs. Next, the processing cycles on the MC module decrease, whereas the deblocking filter becomes the cycle bottleneck from the system point of view. To further reduce the processing cycles, the hybrid filtering schedule and LPL prediction methods are proposed to improve the memory access efficiency and a 34% of cycle reduction can be further obtained compared to the previous design stage.

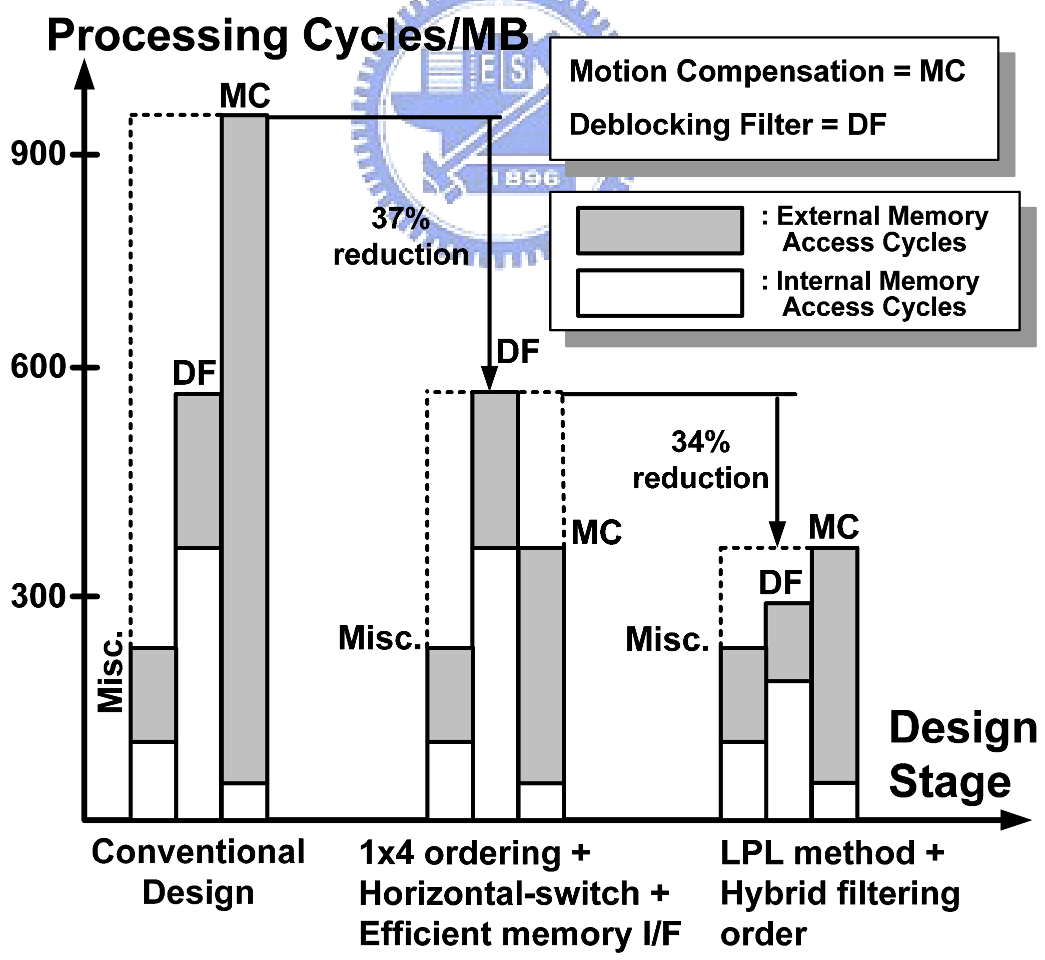


Figure 3.21: Processing cycle breakdown in each architectural design phase.

3.5 Summary

To summarize the low-power techniques discussed in this chapter, Figure 3.22 shows the composition of the power consumption when the dual-video decoder runs at H.264/AVC mode and meets the real-time decoding requirements of QCIF@15fps. By applying domain-pipelined scalability (DPS), three-level memory hierarchy and LPL methods, the CLK and SRAM power are reduced due to optimized register and memory allocations. The DPS method greatly reduces the clock tree power while LPL method is exploited to keep the useful pixels, leading to the elimination of additional accesses to SDRAM. It greatly reduces the on-chip memory power due to the reduction of memory size and access frequency. Moreover, further reduction is obtained through the low-power architectures including the motion compensation and deblocking filter modules. a content-switched MC and hybrid-scheduled DF lower the required working frequency at a cost of slight buffer increment. A dedicated SDRAM interface is presented to improve the access efficiency and reduce the external bandwidth. Therefore, the power dissipation is dramatically reduced. As a result, the overall design approximately reduces power dissipation by 68% as compared to the conventional design approach.

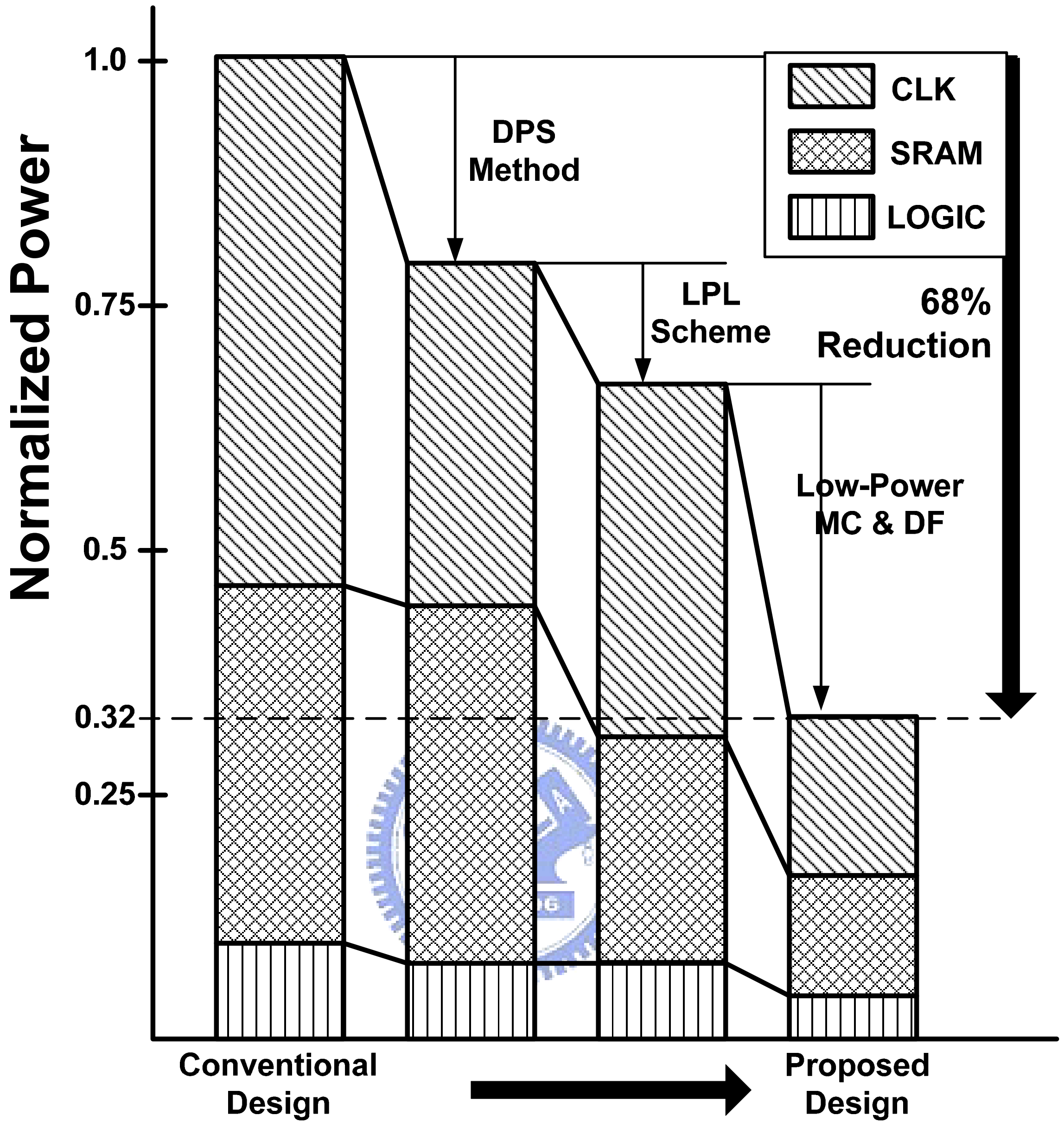


Figure 3.22: Power reduction of proposed MPEG-2/H.264/AVC video decoder.

Note1: Power reduction of the "Low-Power MC and DF" technique is based on the assumption of neglecting power overhead due to cost increment.
Note2: Memory power reduction of "LPL" technique only addressed the memory usage in deblocking filter and intra prediction instead of overall memory usage of H.264/AVC.