ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Design Space Exploration of an Advanced Direct Memory Access Unit for a Generic VLIW processor

AUTOR: Fátima Reino Gómez
DIRECTOR(ES): José Javier Martínez Álvarez,
Francisco Javier Garrigós Guerrero
CODIRECTOR: G.Payá Vayá
P.Pirsch

Septiembre/ 2008

Autor	Fátima Reino Gómez
E-mail del Autor	fatimareino@hotmail.es
Director(es)	José Javier Martínez Álvarez, Francisco Javier Garrigós Guerrero
E-mail del Director	JJavier.Martinez@upct.es
Codirector(es)	G.Payá Vayá, P.Pirsch

Título del PFC	Design Space Exploration of an Advanced Direct Memory Access Unit for a Generic VLIW Processor
Descriptores	DMA avanzado, codificación de video, procesamiento de video, motion estimation, procesadores VLIW.

Resumen

Analizando los actuales requerimientos para la transferencia de datos de los estándares de codificación de video, los diferentes tipos de procesadores, y estudiando el funcionamiento y las ventajas de los accesos directos a memoria (Direct Memory Access), he desarrollado un modelo parametrizable de un acceso directo a memoria. Considerando un método de codificación de video concreto (motion estimation), he implementado una versión mejorada, añadiendo funcionalidad extra en el DMA para conseguir una reducción en el tiempo de ejecución y por lo tanto mejoras en las aplicaciones de video. A partir del estudio de los resultados obtenidos he realizado un análisis exhaustivo indicando las posibles futuras modificaciones.

Titulación	Ingeniero Superior de Telecomunicaciones
Intensificación	Planificación y gestión de Telecomunicaciones
Departamento	Electrónica, tecnología de computadoras y proyectos
Fecha de Presentación	Septiembre-2008

El objetivo de este documento es realizar una breve presentación en castellano sobre el proyecto realizado.

El procesamiento multimedia tiene un gran número de aplicaciones como la televisión interactiva de alta definición o teléfonos móviles multimedia. Debido a esto, está tomando cada vez más importancia en la actualidad.

El procesamiento multimedia abarca la captura, almacenamiento, manipulación y transmisión de objetos multimedia como por ejemplo objetos de audio, de texto, imágenes, video y gráficos de 2-D/3-D.

Hoy en día, debido a la amplia variedad de aplicaciones como por ejemplo las actuales aplicaciones de video, teléfonos móviles multimedia, la televisión interactiva de alta definición… son necesarias continuas mejoras requiriendo para esto sofisticadas operaciones multimedia, las cuales implican una alta demanda de potencia. Por ello y para conseguir mejores resultados, han aparecido una gran variedad de métodos que consiguen que los procesadores se comporten menos linealmente y más en paralelo.

El paralelismo puede existir a varios niveles, por ejemplo de datos, de instrucciones o de hilos (programas).

a) **Paralelismo de instrucciones (ILP)** busca aumentar la velocidad a la que las instrucciones son ejecutadas en la CPU (es decir, aumenta la utilización de los recursos en la ejecución). Se pueden distinguir dos tipos principales:

    - <u>Super-Scalar:</u> Superscalar es el término empleado para describir un tipo de microarquitectura de procesador capaz de ejecutar más de una instrucción por ciclo de reloj. En un procesador superescalar es el hardware en tiempo de ejecución el que se encarga de planificar las instrucciones. La microarquitectura superescalar usa el paralelismo de instrucciones además de el paralelismo de flujo, este último gracias a la estructura en pipeline.

    - <u>VLIW:</u> Esta arquitectura implementa una forma de paralelismo a nivel de instrucción. Es similar a las arquitecuras superescalares, pues ambas usan varias unidades funcionales (por ejemplo: varias ALUs, varios multiplicadores, etc) para lograr este paralelismo.
      Los procesadores con arquitecturas VLIW están caracterizados, como su nombre sugiere, por tener juegos de instrucciones muy simples en cuanto al número de instrucciones diferentes, pero muy grandes en cuanto al tamaño de cada instrucción. Esto es así porque en cada instrucción se especifica el estado de todas y cada una de las unidades funcionales del sistema. El objetivo de este tipo de arquitectura de computadoras es simplificar el diseño hardware dejando la tarea de la planificación del código en manos del programador/compilador. Por el contrario, un procesador superescalar requiere hardware extra para realizar esta programación en tiempo de ejecución.

b) **Paralelismo a nivel de hilo de ejecución (TLP)** busca aumentar el número de hilos (programas individuales) que un CPU puede ejecutar simultáneamente. Ejecuta simultáneamente instrucciones de múltiples hilos en lugar de ejecutar concurrentemente múltiples instrucciones del mismo hilo.

c) **Paralelismo de datos (DLP)** trata con múltiples piezas de datos en el contexto de una instrucción. Por ejemplo:

- **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD) es una técnica empleada para lograr el paralelismo de datos. Las instrucciones aplican la misma operación en un conjunto de datos. Una única unidad de control envía las instrucciones a diferentes unidades de procesamiento. Todos los procesadores reciben la misma instrucción de la unidad de control, pero operan en diferentes conjuntos de datos. En otras palabras, la misma instrucción es ejecutada de manera síncrona por todas las unidades de procesamiento.

Debido a la posibilidad de explotar el paralelismo de datos en las aplicaciones multimedia, la arquitectura VLIW está siendo muy empleada en los últimos años. Usando subword parallelism (es una forma a pequeña escala de SIMD), una arquitectura VLIW puede conseguir unas mejoras significativas en los resultados.

Para explotar el paralelismo de datos, es necesario subword parallelism, que realiza la misma operación simultáneamente en diferentes subword empaquetadas en una word. Subword es una pequeña unidad de datos que está contenida en una word. En subword parallelism, se empaquetan varias subword en una word para posteriormente procesar la word completa. El paralelismo de datos también necesita:

1) Un modo de expandir los datos en largos contenedores.

2) Reorganización de las subword en un registro. Muchos son los algoritmos que necesitan esta reorganización de las subword. Dos ejemplos de instrucciones que se encargan de realizar esta reorganización de las subword son:

   - **MIX:** Estas instrucciones cogen subwords de dos registros e intercalan alternativas subwords de cada uno de los registros en el registro final. Mix left empieza cogiendo la subword situada más a la izquierda de cada uno de los dos registros fuente, mientras que mix right termina con la subword situada más a la derecha de cada uno de los dos registros fuente. La figura 1.1 del proyecto ilustra esto para 16-bit subwords [1].

- **Permute:** La instrucción permute coge un registro fuente y realiza una permutación de las subwords de este registro. Con subwords de 16-bits, esta instrucción puede generar todas las permutaciones posibles, con o sin repeticiones, de las 4 subwords en el registro fuente. La figura 1.2 del proyecto muestra una posible permutación. Para especificar una permutación particular, se usa un índice de permutación. El número de subword en el registro fuente empieza con cero para el subword situado más a la izquierda. El índice de permutación identifica el subword del registro fuente se situará en un determinado subword del registro destino [1].

3) Alineamiento de datos realizado durante una transferencia entre una memoria externa y una memoria local.

Una referencia a memoria es llamada alineada cuando accede a posiciones de memoria que coinciden con el acceso a memoria alineado del procesador, cuando esto no ocurre es llamada no alineada. Para acceder a una posición no alineada, se debe realizar un proceso de alineamiento. El proceso de alineamiento se muestra en la figura 1.3 del proyecto.

Se debe leer la word alineada en memoria que se encuentra antes de la posición no alineada y descartar los bytes que no son necesarios, después se lee la word alineada en memoria que está localizada después de la posición no alineada y descartar los bytes que no son necesarios y finalmente se juntan los bytes necesarios de las dos anteriores words.

La mayoría de los procesadores SIMD proporcionan acceso solo a datos contiguos en memoria, con fuertes restricciones de alineamiento porque tienen una arquitectura de memoria limitada. Estas arquitecturas, a menudo no proporcionan ningún soporte hardware para accesos no alineados o lo proporcionan pero degradando los resultados. Debido a esto, normalmente el programador acaba realizando el alineamiento en software, lo cual implica una carga extra y degrada los rendimientos significativamente. Por ejemplo, en aplicaciones de codificación y decodificación de video que usan los algoritmos motion estimation (ME) y motion compensation (MC), se requieren alineamientos de modo impredecible.

Hoy en día, hay numerosos consensos sobre la importancia de los accesos no alineados para las aplicaciones de video (no teniendo un soporte eficiente para accesos no alineados se degradan los rendimientos de manera significativa), pero la mayoría de las actuales arquitecturas SIMD que soportan accesos no alineados presentan restricciones y limitaciones.

Debido a que las aplicaciones requieren cada vez más recursos, una unidad de acceso directo a memoria (DMA) es una característica esencial en todos los ordenadores modernos, ya que permite a diferentes dispositivos comunicarse sin someter a la CPU a una carga masiva de interrupciones, incrementando de este modo los rendimientos de la aplicación. Una transferencia DMA esencialmente copia un bloque de memoria de un dispositivo a otro. Con un DMA, la CPU podría iniciar la transferencia, realizar otras operaciones mientras la transferencia está en proceso, y recibir una interrupción del controlador del DMA una vez que la operación ha sido realizada. Un uso típico en un DMA es transferir un bloque de memoria desde/hacía una memoria externa o desde/hacía un buffer del dispositivo.

Con las instrucciones explicadas anteriormente, el proceso de alineamiento somete a la CPU a una gran sobrecarga. Hay muchas aplicaciones, como las de procesamiento de video, que requieren un soporte eficiente para accesos no alineados, el no tenerlo degrada los rendimientos significativamente. Debido a que el DMA permite a los dispositivos transferir datos sin someter a la CPU a una carga masiva de interrupciones, una posible solución sería diseñar un **'Enhanced DMA Unit'** (DMA inteligente) capaz de realizar este alineamiento de los datos. Este nuevo DMA ahorrará muchos ciclos porque ahora este alineamiento es realizado en el DMA y la CPU puede ejecutar otras partes del código de la aplicación en paralelo.

La idea de este proyecto es implementar un DMA que además de realizar este alineamiento pueda también realizar otras permutaciones de datos, por ejemplo padding (extensiones de datos). Las instrucciones explicadas anteriormente seguirán siendo empleadas porque una vez que los datos estén en la memoria interna, estas instrucciones serán las responsables de realizar cualquier alineamiento requerido. Pero muchos serán los casos en los que podremos usar el **'Enhanced DMA Unit'** y colocar los datos ya alineados del modo requerido en la memoria interna. De este modo se eliminarán muchas instrucciones de código y por tanto estaremos eliminando ciclos.

Un ejemplo donde el **'Enhanced DMA Unit'** puede claramente reducir la sobrecarga para la CPU sería en aplicaciones de codificación y decodificación de video que usan los algoritmos ME y MC. Estos algoritmos requieren mover bloques de datos de una memoria externa a una local más rápida, y alinear estos bloques para colocarlos correctamente en la memoria local. Además en los casos en los que estemos leyendo fuera de la imagen de referencia será necesario realizar una extensión de los datos (padding process).

Los objetivos de este proyecto son los siguientes:

- Analizar los requerimientos de transferencias de los actuales estándares de codificación de video y desarrollar un modelo parametrizable de un acceso directo a memoria (Direct Memory Access).

- Considerando un método particular de codificación, por ejemplo motion estimation, una versión mejorada será implementada, añadiendo funcionalidad extra en el DMA para disminuir el tiempo de ejecución.

- Un análisis exhaustivo de los rendimientos de los resultados será realizado, indicando posibles futuras modificaciones.

Este proyecto está organizado del siguiente modo. El capítulo 2 presenta los procesadores VLIW y una introducción a los DMA. El capítulo 3 introduce H.264/AVC que es un estándar de compresión de video y además introduce el algoritmo motion estimation (ME). El capítulo 4 expone las propuestas para el 'Enhanced DMA Unit'. Después de esto, en el capítulo 5, se muestra la evaluación del DMA propuesto usando el algoritmo motion estimation. Finalmente, las conclusiones son presentadas en el capítulo 6.

H.264/ AVC y Motion Estimation

H.264 es un estándar para compresión de video. El objetivo del proyecto H.264/AVC era crear un estándar capaz de proporcionar buena calidad de video, sin aumentar mucho la complejidad del diseño para que este no fuera impráctico o excesivamente caro de implementar.

La figura 3.2 del proyecto muestra un diagrama genérico de un codificador de video.

A continuación se va a realizar un estudio del bloque motion estimation para analizar las mejoras que serían útiles en un DMA.

Motion Estimation

Es el proceso de determinar los vectores de movimiento que describen la transformación de una imagen a otra.

La idea clave de motion estimation es que imágenes de video consecutivas serán similares excepto por pequeños cambios introducidos por objetos que se mueven en las imágenes.

Motion estimation extrae información del movimiento de las secuencias de video. El movimiento está representado con un vector de movimiento(x,y). Este vector indica el desplazamiento de un pixel o un bloque de pixels de la imagen actual debido al movimiento. El proceso de aplicar este vector de movimiento a una imagen para obtener la siguiente se llama motion compensation. La combinación de motion compensation y estimation es una pieza clave de la compresión de video.

Hay muchas técnicas para realizar motion estimation, pero la más empleada es el algoritmo de correspondencia de bloque (Block Matching Algorithm).

Block Matching Algorithm

Este algoritmo calcula un vector de movimiento para un bloque de pixels. La figura 3.11 del proyecto puede ayudar a comprender el funcionamiento del algoritmo.

La imagen actual es dividida en bloques de pixels y motion estimation se realiza para cada uno de estos bloques.

Motion estimation se realiza identificando el bloque de pixels de la imagen de referencia que más se le parece al bloque a codificar. El bloque de referencia es generado por el desplazamiento (este desplazamiento se realiza con los vectores de movimiento) de la localización del bloque a codificar en la imagen de referencia.

Para encontrar el bloque de pixels que más se parece al bloque actual la búsqueda en la imagen de referencia se realiza solo en una región de búsqueda.

Para decidir cual es el bloque de pixels que más se parece al actual hay dos posibles criterios:

- Sum of Square Error (SSE) = $\sum_{x=1}^{x=N}\sum_{y=1}^{y=N}(C(x,y)-R(x,y))^2$

SSE proporciona una muy buena correspondencia de bloque pero requiere una carga computacional alta.

- Sum of Absolute Difference (SAD) = $\sum_{x=1}^{x=N}\sum_{x=1}^{y=N}|C(x,y)-R(x,y)|$

SAD proporciona una buena correspondencia de bloque y requiere menor carga computacional. Por esto el criterio más usado es SAD. El bloque que más se parece al bloque actual es el bloque con el menor SAD, y su vector de movimiento será el vector de movimiento que describa el desplazamiento del bloque actual.

Se ha asumido que el bloque se va a encontrar situado en un número entero de píxeles. En la práctica, el desplazamiento de un objeto entre dos imágenes consecutivas de video no coincidirá con un número entero de píxeles. Por lo tanto, los actuales estándares de codificación de video emplean subredes de píxeles, en las cuales los vectores de movimiento pueden apuntar a bloques candidatos localizados en medio-pixel o incluso en cuarto-pixel.

En la aplicación empleada en el proyecto es posible apuntar a bloques situados solo en medio-pixel (además de pixeles enteros) y existen los siguientes casos:

- FH: Media vertical (círculo verde).
- HF: Media horizontal (círculo azul).
- HH: Media vertical y media horizontal (círculo amarillo).
- FF: copia (círculo rojo).

Estos casos están representados en la figura 3.9 del proyecto.

Buscar el bloque en toda la región de búsqueda supone un gasto computacional elevado. Para reducir esta búsqueda en la imagen de referencia existen diversos algoritmos, en este proyecto se va a emplear búsqueda recursiva de tres dimensiones (Three Dimensional Recursive Search (3DRS)).

Three Dimensional Recursive Search

Three dimensional recursive search (3DRS) es un algoritmo de búsqueda. Este algoritmo se basa en dos premisas:

- Los objetos son más largos del tamaño de un bloque
- Los objetos tienen inercia

Teniendo en cuenta la primera premisa los vectores de movimiento de los vecinos pueden ser usados como candidatos para el bloque actual. En este punto puede aparecer un problema para bloques vecinos cuyo vector de movimiento aun no haya sido calculado. Aquí es donde se aplica la segunda premisa y se emplean para estos bloques los vectores de movimiento de la imagen anterior. En la aplicación empleada en este proyecto para la búsqueda de los vectores de movimiento candidatos mediante 3DRS, se cogen los vectores de movimiento de cuatro bloques vecinos y sumando un random a estos cuatro vectores de movimiento obtenemos otros cuatro. Es decir, obtenemos en total ocho vectores de movimiento candidatos.

Cuando hay dos candidatos iguales, dos vectores de movimiento iguales, solo se empleará una vez este vector de movimiento, por tanto tendremos en este caso un candidato menos.

Scheme of operation of motion estimation

La figura 3.15 del proyecto muestra el esquema de operación de motion estimation.

Los pasos son los siguientes:

1. El bloque (a codificar) de la imagen actual es introducido.
2. Se estima el primer vector de movimiento en la imagen actual por medio de 3DRS.
3. Con el vector de movimiento y la imagen de referencia, por medio de motion compensation, se localiza el candidato en la imagen de referencia.
4. Por medio de la resta del bloque de la imagen actual y del candidato localizado en la imagen de referencia, se obtiene el SAD.
5. Se repiten los pasos del 2 al 4 para los 8 candidatos, para quedarse finalmente con el vector de movimiento del candidato con el menor SAD.

DMA propuesto

DMA con alineamiento

Las direcciones apuntan a datos de 64 bits. Debido a que ahora existe subword parallelism, la información está dividida en pequeños datos de 8 bits. Es decir, las direcciones apuntan ahora a words que están formadas por 8 subwords. Gracias al alineamiento de datos es posible apuntar a cualquier subword y a partir de esta coger una word completa.

La idea es usar los primeros tres bits de la dirección para realizar el alineamiento de datos, es decir, con esos tres bits indicaremos si la word está alineada o si es necesario coger una word de 64-bits a partir de una determinada subword. En la figura 4.7 del proyecto se ilustra esto.

Como se puede observar, los números en color negro permanecen iguales para los 64 bits de la word alineada, y los tres primeros bits (marcados en rojo) indican la subword a la que se está apuntando.

En el caso en que sea necesario realizar el alineamiento de datos, se deberá copiar la word alineada en memoria que se encuentra antes de la posición no alineada en un registro interno del DMA, después copiar la word alineada en memoria que se encuentra después de la posición no alineada en otro registro interno del DMA, y finalmente en otro registro almacenar los datos necesarios (determinaremos los datos necesarios con los tres bits nombrados anteriormente) de las dos words anteriores. Las figuras 4.9 y 4.10 del proyecto ilustran un ejemplo.

Realizando el alineamiento en el DMA, se conseguirá una mejora en los resultados obtenidos porque se disminuye considerablemente la carga a la que se sometía a la CPU realizando el alineamiento con otros métodos.

Con las instrucciones descritas anteriormente, era necesario cargar los datos en memoria, posteriormente realizar el alineamiento en software y finalmente cargar los datos alineados de nuevo en memoria. Por lo tanto se están empleando tres registros y además se está haciendo trabajar a la CPU.

Realizando el alineamiento en el DMA, los datos son cargados en memoria ya alineados. De este modo el alineamiento se lleva a cabo sin interrumpir a la CPU que mientras se realiza este proceso puede realizar otras tareas, además solo se está empleando un registro de memoria (en el que se almacenan los datos ya alineados del modo deseado).

En aplicaciones de codificación y decodificación de video que usan los algoritmos ME y MC, este modo de realizar el alineamiento de datos puede mejorar significativamente los resultados.

DMA con extensiones

En aplicaciones de codificación y decodificación de video que usan los algoritmos ME y MC, cuando se quiere buscar el bloque de la imagen de referencia que más se parece al bloque que se va a codificar, se busca solo en una región de búsqueda. Cuando un candidato (a ser el bloque de la imagen de referencia que más se parece al bloque a codificar) se sale fuera de esta región, es necesario hacer una extensión de los datos (padding). La figura 4.11 del proyecto muestra la necesidad de realizar padding en las aplicaciones de video.

En la figura, se puede observar como parte de un candidato se encuentra fuera de la región de búsqueda (el bloque amarillo), lo cual significa que para representar este candidato, es necesario realizar una extensión de los datos. En este caso, la extensión de los datos se realiza por la parte de arriba, es decir, los contenidos de la última línea de datos que está dentro de la región de búsqueda, deben ser replicados hacía arriba.

Hay ocho casos diferentes donde existe una necesidad de extensión como se puede observar en la figura 4.12 del proyecto:

- Extensión horizontal por la derecha (bloque marrón).
- Extensión horizontal por la izquierda (bloque gris).
- Extensión vertical superior (bloque amarillo).
- Extensión vertical inferior (bloque naranja).
- Extensión vertical superior y extensión horizontal por la derecha (bloque azul).
- Extensión vertical superior y extensión horizontal por la izquierda (bloque verde).
- Extensión vertical inferior y extensión horizontal por la derecha (bloque lila).
- Extensión vertical inferior y extensión horizontal por la izquierda (bloque rosa).

La idea es que el **'Enhanced DMA Unit'** además de realizar el alineamiento de los datos, realice también la extensión de estos.

Para realizar la extensión de datos en el DMA es necesario crear dos nuevos registros que aparecen ilustrados en la figura 4.13 del proyecto:

- **X_EDGE**: Indica el número de words que se encuentran fuera de la imagen. Si es positivo significa que las words están fuera de la imagen por el lado izquierdo, y si es negativo por el lado derecho.
- **Y_EDGE**: Indica el número de líneas que se encuentran fuera de la imagen. Si es positivo significa que las líneas están fuera de la imagen por arriba, y si es negativo por abajo.

En aplicaciones de codificación y decodificación de video que usan los algoritmos ME y MC, realizar las extensiones en el DMA reduciría considerablemente la sobrecarga de la CPU. Cada vez que un candidato se sale fuera de la región de búsqueda, la CPU debe hacer un trabajo extra, un trabajo que se puede ahorrar si es realizado por el DMA. Además si la CPU no tiene que realizar este trabajo, podrá realizar otras tareas mientras el DMA realiza las extensiones.

DMA con cola

En aplicaciones de codificación y decodificación de video que usan los algoritmos de ME y MC, para encontrar el bloque candidato que más se parece al bloque a codificar, es necesario transmitir varios candidatos, y cada vez que se transmite un candidato se interrumpe a la CPU y se le hace trabajar para cargar los datos de la transmisión.

Por lo tanto, sería interesante que el DMA fuera capaz de almacenar varias transmisiones en la CPU, de modo que cuando una transferencia termine, no sea necesario programar la siguiente sino que esta ya estará en la cola y simplemente cargará directamente los datos necesarios para realizar la siguiente transferencia.

La idea es que el **'Enhanced DMA Unit'** además de realizar el alineamiento y las extensiones de datos, pueda almacenar varias transmisiones en una cola, porque de este modo estaremos reduciendo la carga de la CPU.

Para implementar esto en el DMA un se ha sido necesario crear un nuevo registro:

- **DMA_READ_OUT**: Es un registro de un bit. El usuario pone a 1 esta variable cuando quiere saber si el DMA ha terminado con una determinada transferencia o está aun transmitiendo. Este registro incrementa un puntero de lectura interno.

Además para implementar el DMA son necesarios tres punteros internos:

- **Write**: Indica el número de transferencias cargadas. Es incrementado cada vez que los datos de una nueva transmisión son cargados en la cola.
- **Read**: Indica el número de la transmisión del que el usuario quiere saber el estado. Es incrementado cada vez que el registro DMA_READ_OUT está a alta.
- **Transfer**: Indica el número de la transmisión actual. Es incrementado cada vez que la transmisión de un bloque finaliza.

Cuando una de estas variables alcanza el límite de la cola, se pone a cero, y los registros internos que usan esta variable como índice son sobrescritos.

Para entender mejor el modo de funcionamiento de las colas y el uso del registro DMA_READ_OUT, ver las figuras 4.29 y 4.30 del proyecto respectivamente.

En aplicaciones como la codificación y decodificación de video que usan los algoritmos ME y MC, un DMA capaz de almacenar varias transmisiones en una cola, reduce considerablemente la sobrecarga de la CPU mejorando de este modo los resultados.

Evaluación del DMA propuesto

Descripción del sistema

El sistema con el que trabaja este proyecto se muestra en la figura 5.1 del proyecto.

El proyecto trabaja con el **'Enhanced DMA Unit'**, que está en el interior del procesador Moai4k.

Hay tres modelos diferentes de DMA que serán analizados:

- **Basic Profile (BP)** es un DMA básico.

- **Advanced Profile 2 (AP2)** es el Basic Profile añadiéndole los procesos de alineamiento y padding.

- **Advanced Profile 3 (AP3)** es el Advanced Profile 2 añadiéndole una cola para almacenar varias configuraciones de transferencias.

El **'Enhanced DMA Unit'**, se comunica con la memoria externa (SDRAM) por medio del bus AMBA.

Con el objetivo de obtener resultados más realistas, una latencia ha sido introducida en el controlador de memoria externa y los resultados han sido obtenidos con diferentes valores para esta latencia (entre 15 y 55 ciclos).

Algoritmo de Motion Estimation

Usando el **'Enhanced DMA Unit'** propuesto y el algoritmo de motion estimation como aplicación, las ventajas de este DMA serán demostradas.

El algoritmo motion estimation ha sido elegido como aplicación por las siguientes razones:

- Con el algoritmo motion estimation, es inevitable el tener que acceder a posiciones de memoria no alineadas cuando se hace referencia a un bloque candidato.

- Con el algoritmo motion estimation, para buscar el bloque de la imagen de referencia que más se asemeja al bloque a codificar, la búsqueda se realiza solo en una región de búsqueda, por lo tanto cuando un candidato se sale fuera de esta región es necesario realizar una extensión de los datos.

- Con el algoritmo motion estimation es necesario transmitir una gran cantidad de bloques, y cada vez que se transmite uno la CPU debe interrumpir lo que esté realizando y cargar los datos para la siguiente transferencia.

El algoritmo de motion estimation ha sido programado en lenguaje ensamblador. La estructura del código se puede ver en la Figura 5.2 del proyecto.

El algoritmo ha sido implementado de este modo porque se ha comprobado que es la forma óptima para programar las transferencias en este algoritmo teniendo en cuenta que se debe terminar una transferencia para programar la siguiente.

Resultados

Empleando el **'Enhanced DMA Unit'** propuesto y el algoritmo motion estimation como aplicación, se han realizado análisis con latencias entre 15 y 55 ciclos para cada uno de los diferentes modelos de DMA implementados.

En el proyecto se pueden ver los resultados detallados para cada uno de los modelos con diferentes latencias, pero en este resumen se van a comentar los resultados para el modelo básico con diferentes latencias y se va a realizar una comparación entre los diferentes modelos implementados.

En las figuras 5.3, 5.4 y 5.5 del proyecto aparecen los resultados del modelo básico con latencias de 15, 30 y 45 ciclos respectivamente.

En las gráficas se puede observar el número de ciclos que cada bloque emplea en realizar cada tarea. Además se puede observar el número total de ciclos empleado por cada bloque para encontrar el bloque de la imagen de referencia que más se parece al bloque a codificar.

Los picos aparecen en las gráficas porque cuando hay dos vectores de movimiento iguales, se usa solo una vez este vector de movimiento, y por lo tanto tendremos un candidato menos. Debido a esto habrá un número de candidatos diferente dependiendo del bloque a codificar, por lo que se realizarán un número diferente de operaciones, y por este motivo aparecen los picos en las gráficas.

Comparando los resultados del modelo básico con diferentes latencias se puede apreciar como el número de ciclos que cada bloque emplea para encontrar el bloque que más se le parece, aumenta al aumentar la latencia. Esto es porque cuando la latencia aumenta, aumenta el tiempo que se debe esperar a que una transferencia finalice para poder empezar la siguiente. Con los otros modelos de DMA implementados ocurre lo mismo.

En la figura 5.12 del proyecto se muestra una comparación de los resultados obtenidos con los diferentes modelos de DMA implementados.

En la gráfica se puede apreciar el número total de ciclos que cada bloque emplea en media en ejecutar el algoritmo para cada modelo con cada latencia.

A partir de la gráfica se puede concluir que:

- Cuando la latencia aumenta, aumenta el número total de ciclos ejecutados para identificar el bloque de la imagen de referencia que más se asemeja al bloque que se quiere codificar.

- Comparando el BP con el AP2 con latencias bajas, el número total de ciclos ejecutados decrece porque el proceso de alineamiento y padding es realizado por el DMA. Sin embargo con latencias altas, el número de ciclos tiende a ser igual. Esto se debe a que con bajas latencias no es necesario esperar a que llegue el bloque y cuando el alineamiento y las extensiones se realizan en el DMA, el número de ciclos se reduce. Por el contrario, con latencias altas es necesario esperar a que llegue el bloque y no importa el hecho de ahorrar ciclos realizando el alineamiento y las extensiones en el DMA porque de todos modos tendré que seguir esperando una vez estos se hayan realizado.

- Comparando AP2 con AP3, es fácil apreciar que AP3 emplea menos ciclos que AP2 y esto es más notable conforme aumenta la latencia. Esto se debe a que con latencias bajas en la mayoría de los casos no es necesario esperar, pero con largas latencias será necesario esperar en casi todos los casos, y con AP2 se debe esperar más porque espera el fin de una transmisión para programar la siguiente.

- El código para AP2 ha sido implementado para obtener los mejores resultados teniendo en cuenta que para realizar una transferencia debe haberse completado la previa, y por este motivo la diferencia entre el AP2 y AP3 es menor. Con AP3 además de disminuir el número de ciclos que se emplean al ejecutar el algoritmo se ahorra trabajo al programador, al no tener este que decidir el mejor modo de implementar el código teniendo en cuenta que debe haberse terminado una transferencia para empezar la siguiente.

Conclusiones

En este proyecto han sido analizados los procesadores VLIW y SIMD, las unidades de acceso directo a memoria (DMA) y los estándares de codificación de video. Motion estimation (un método particular de codificación de video) ha sido estudiado, analizando el algoritmo y las transferencias necesarias.

En aplicaciones de codificación y decodificación de video que usan motion estimation (ME), no es posible evitar las referencias a posiciones no alineadas de memoria. El algoritmo ME busca el bloque que más se parece al bloque a codificar. Este bloque, llamado de referencia, es generado por el desplazamiento de la localización del bloque a codificar en la imagen de referencia. Este desplazamiento puede ser arbitrario y por tanto, esto hace que sean necesarios múltiples e impredecibles accesos a posiciones no alineadas de memoria.

En este tipo de aplicaciones, las actuales optimizaciones de software no son del todo satisfactorias porque las instrucciones necesarias para realizar el alineamiento de datos en software someten a la CPU a una gran sobrecarga.

Conociendo que un DMA es una característica esencial en todos los ordenadores modernos debido a que el DMA permite a los dispositivos transferir datos sin someter a la CPU a una carga masiva de interrupciones, la solución adoptada en este proyecto ha sido diseñar un **'Enhanced DMA Unit'** (DMA inteligente) capaz de realizar el alineamiento de los datos.

Este **'Enhanced DMA Unit'** además de realizar el proceso de alineamiento puede realizar el proceso de extensión de los datos (padding). Por ejemplo en aplicaciones de codificación y decodificación de video que emplean el algoritmo de motion estimation para encontrar el bloque de la imagen de referencia que más se le parece al bloque que se quiere codificar, la búsqueda se realiza solo en una región conocida como área de búsqueda. Cuando un candidato (a ser el bloque de pixel de la imagen de referencia que mejor se asemeja al bloque a ser codificado) se sale fuera del área de búsqueda, es necesario realizar una extensión de los datos (padding).

Además este **'Enhanced DMA Unit'** es capaz de almacenar varias configuraciones de transferencias en una cola, de este modo cuando una transferencia termine, se iniciará inmediatamente la siguiente. Ahora no es necesario esperar a que termine la transferencia anterior para programar otra.

Con el **'Enhanced DMA Unit'** propuesto, que se encuentra en el interior del procesador Moai4k y se comunica con la memoria externa (SDRAM) por medio del bus AMBA, y usando el algoritmo motion estimation como aplicación, han sido analizados los resultados obtenidos empleando tres modelos diferentes:

- **Basic Profile (BP)**
- **Advanced Profile 2 (AP2)**

- **Advanced Profile 3 (AP3)**

Con el objetivo de obtener resultados más realistas, una latencia ha sido introducida en el controlador de memoria externa y los resultados han sido obtenidos con diferentes valores para esta latencia (entre 15 y 55 ciclos).

Se puede concluir que:

- Cuando la latencia aumenta, aumenta el número total de ciclos ejecutados para identificar el bloque de la imagen de referencia que más se asemeja al bloque que se quiere codificar.

- Comparando el BP con el AP2, el número total de ciclos ejecutados decrece porque el proceso de alineamiento y padding es realizado por el DMA.

- Comparando AP2 con AP3, es fácil apreciar que AP3 emplea menos ciclos que AP2. Con AP2 DMA es necesario esperar el final de una transferencia para programar la siguiente. Esta "espera para programar transferencias", que es programada en el código de instrucciones, es evitada empleando el AP3 DMA.

En conclusión, con AP3 se ha disminuido el tiempo de ejecución, porque ha sido eliminado código de instrucciones (proceso de alineamiento y padding) y no es necesario esperar el final de una transferencia para programar una nueva. Además, con AP3, el tiempo requerido para considerar el mejor lugar en el código de la aplicación para esperar el fin de una transferencia para empezar una nueva, es eliminado.

Para finalizar el proyecto, se proponen algunas ideas indicando posibles futuras modificaciones.

Cuando el padding está siendo realizado, es necesario copiar los contenidos de una línea en todas las líneas que están fuera de la imagen, esto ha sido implementado leyendo varias veces la misma línea. Leer todo el tiempo el contenido de la misma línea no es necesario y se producen transferencias innecesarias que pueden ser evitadas:

- Ajustando el tamaño del bloque a los contenidos de este que permanecen dentro de la imagen.

- Almacenando este contenido en la memoria local, y copiando estas líneas que son iguales al mismo tiempo.

La implementación de estas futuras modificaciones podría ahorrar bastantes recursos pero haría el DMA mucho más complejo.

Como conclusión a este proyecto se puede decir que un nuevo modelo de DMA inteligente **'Enhanced DMA Unit'** ha sido integrado en un procesador VLIW genérico para futuros estudios con otras aplicaciones multimedia.

Leibniz Universität Hannover

Institute of Microelectronic Systems

Prof. Dr.–Ing. P. Pirsch

Design Space Exploration of an Advanced Direct Memory Access Unit for a Generic VLIW Processor

Master Thesis

Fátima Reino Gómez

July 2008

Advisor : Ing. G. Payá Vayá

Examiner : Prof. Dr.-Ing. P. Pirsch

MASTER'S THESIS

Ms. Fátima Reino Gómez

Design Space Exploration of an Advanced Direct Memory Access Unit for a Generic VLIW Processor

In the last decade embedded consumer device requirements have grown, requiring more efficient computer architectures. On the one hand, current image and video coding standards, e.g. MPEG-4, H.264 or JPEG2000, are pushing the limits of the existing media processors. Actual media processors cannot comply the tremendous demand on the increasingly sophisticated multimedia operations, requiring special architectural improvements. On the other hand, low power consumption is mandatory in any multimedia portable device.

A Direct Memory Access (DMA) unit bus is mandatory in this kind of media processors. A processor uses the DMA to transfer data between memory spaces or between a memory space and a peripheral in background. An Advanced DMA can help to reach the performance demanded by a multimedia application by performing some data transformation during the background transfers. Moreover, this data transformation will decrease the amount of executed operations, saving power consumption.

Ms Fátima Reino Gómez's task is firstly to analyze the transfer's requirements of actual video coding standards and to develop a parameterized Direct Memory Access unit model. Then, considering a particular video coding task, i.e. motion estimation, an improved version has to be implemented, by adding extra functionality in the DMA unit in order to decrease the required execution time. Finally, an exhaustive analysis of the performance results should be done, indicating possible future modifications.

The submitted copies as well as the results of the work remain the property of the institute.

Prof. Dr.-Ing. Peter Pirsch

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Hannover, July 2008

Fátima Reino Gómez

Contents

1. Introduction **1**

2. VLIW processors and DMA Basics **6**

2.1. - Basics of VLIW/SIMD processors **6**

2.1.1. - VLIW vs. Superscalar processors 6

2.1.2. - VLIW benefits 8

2.1.3. - SIMD processors 9

2.2. - Moai4k **9**

2.2.1. - Introduction 10

2.2.2. - A generic VLIW architecture. 10

2.3. - Basics of Direct Memory Access Units **15**

2.3.1. - Introduction 15

2.3.2. - Operation of a DMA 15

2.3.3. - DMA Transfer Strategies. 19

2.3.4. - How to program a DMA Controller 19

2.3.5. - DMA Classification 25

3. H.264/AVC and Motion Estimation **26**

3.1. - H.264/AVC Coding Standard **26**

3.1.1. - Introduction 26

3.1.2. - Video Encoder 28

3.1.3. - Video Decoder 35

3.2. - Motion Estimation **35**

3.2.1. – Introduction 36

3.2.2. - Block Matching Algorithm 36

3.2.3. - Three Dimensional Recursive Search 38

3.2.4. - Scheme of operation of motion estimation 40

3.2.5. - Example of motion estimation 41

4. Proposed enhanced DMA unit 44

4.1. - Introduction 44

4.2. - DMA processor interface 45

4.2.1. - Processor interface 46

4.2.2. - Example 46

4.3. - Alignment process 48

4.3.1. - Introduction 48

4.3.2. - DMA with alignment 50

4.4. - Padding process 56

4.4.1. - Introduction 56

4.4.2. - DMA with padding 58

4.5. - Queue implementation 71

4.5.1. - Introduction 71

4.5.2. - DMA with queue 71

5. Evaluation of the proposed enhanced DMA Unit 74

5.1. - Design Space Exploration 74

5.1.1. - System description 74

5.1.2. - Motion Estimation task 75

5.2. - Characteristics of motion estimation algorithm . . . 78

5.3. - Results 78

5.3.1. - Results of Basic Profile (BP) 79

5.3.2. - Results of Advanced Profile 2 (AP2) 82

5.3.3. - Results of Advanced Profile 3 (AP3) 85

5.3.4. - Comparison of the different profiles 88

6. Conclusions 91

7. References 94

Chapter 1

Introduction

Multimedia processing has increasingly applications such as high-definition interactive television or multimedia cell phones. Because of this, multimedia processing is taking an increasingly importance.

Media processing involves the capture, storage, manipulation and transmission of multimedia objects such as audio objects, handwritten data, text, images, full-motion video and 2-D/3-D graphics.

Nowadays, because of the wide variety of applications such as current video applications, multimedia cell phones, high-definition interactive television… are necessary continuous improvements requiring sophisticated multimedia operations, which imply a high demand of power. Because of this and to achieve better performance, it has resulted in a variety of design methodologies that cause the processors to behave less linearly and more in parallel.

Parallelism can exist at various levels i.e. data, instruction and thread.

a) **Instruction level parallelism (ILP)** seeks to increase the rate at which instructions are executed within a CPU (that is, to increase the utilization of on-die execution resources). We can distinguish two principal types:

   - Super-Scalar: Superscalar is the term used to describe a type of microarchitecture processor capable of running more than one instruction per clock cycle. In a superscalar processor is the hardware at runtime which is responsible for planning the instructions. The microarchitecture superscalar uses the instruction parallelism in addition to the parallelism flow, the latter through the pipeline structure.

   - VLIW: This architecture CPU implements a form of parallelism at the level of instruction. It is similar to architectures Superscalar, both used several functional units (e.g., several ALUs, several multipliers, etc.) to achieve this parallelism.
     Processors with VLIW architectures are characterized, as its name suggests, in executing very long instructions that contain several independent operations, which each one uses a different functional unit. The aim of this computer architecture is to simplify the hardware design leaving the instruction scheduling task in the hands of the programmer / compiler. As opposed, a

superscalar processor requires extra hardware to perform this scheduling at runtime.

b) Thread level parallelism (TLP) seeks to increase the number of threads (effectively individual programs) that a CPU can execute simultaneously.

c) Data level parallelism (DLP) deal with multiple pieces of data in the context of one instruction. An example:

- **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD) is a technique employed to achieve data level parallelism, as in a vector processor. The instructions apply the same operation on a set of data. A single control unit sends instructions to different processing units. All processors receive the same instruction from the control unit, but operating on different data sets. In other words the same instruction is executed in a synchronous manner by all processing units.

Due to the capability of exploiting parallelism in multimedia applications, VLIW architecture is being very used in the last years. Using subword parallelism (this is a form of small-scale SIMD), a VLIW architecture can achieve a significant improvement in performs.

To exploit data parallelism, we need subword parallel compute instructions, which perform the same operation simultaneously on subwords packed into a word. A subword is a lower precision unit of data contained within a word. In **subword parallelism,** we pack multiple subwords into a word and then process the whole word. Data parallelism also need:

1) A way to expand data into larger containers.

2) Subword rearrangement within a register. Many algorithms require this subword rearrangement. Two data rearrangement instruction examples are:

   - **MIX:** These instructions take subwords from two registers and interleave alternate subwords from each register in the result register. Mix left starts from the left most subword in each of the two source registers, while mix right ends with the rightmost subwords from each source register. Figure 1.1 illustrates this for 16-bit subwords [1].

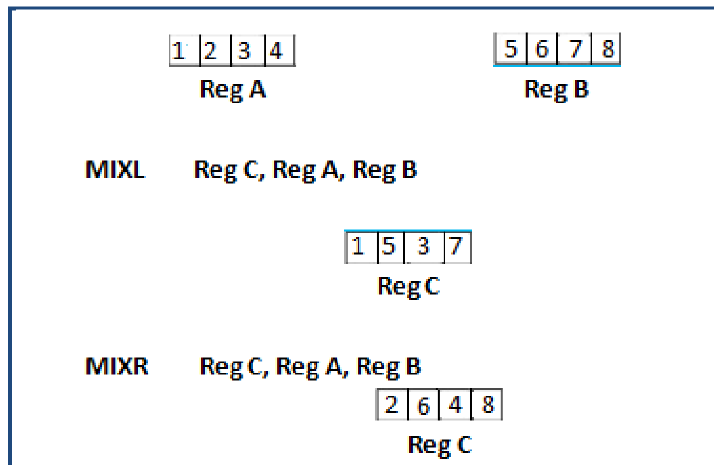


Figure 1.1.*MIX instruction*

- **Permute:** The permute instruction takes one source register and produces a permutation of that register's subwords. With 16-bit subwords, this instruction can generate all possible permutations, with and without repetitions, of the four subwords in the source register. Figure 1.2 shows a possible permutation. To specify a particular permutation, we use a permute index. The instruction numbers subwords in the source register starting with zero for the leftmost subword. A permute index identifies which subword in the source registers places in each subword of the destination register [1].

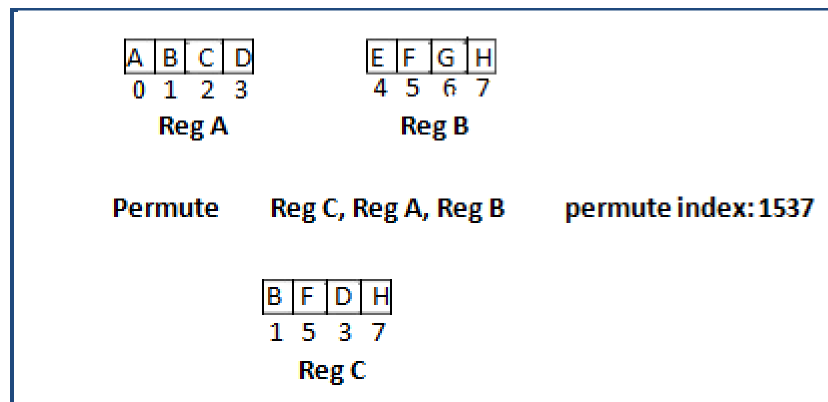


Figure 1.2.*Permute instruction*

3) Data alignment performed during a transfer between an external memory and a local memory.

A memory reference is called *aligned* when it accesses positions that match with the memory access granularity of the processor, when that does not happen is called *misaligned* (or *unaligned*). To access an unaligned position, must be done a realignment process. The realignment process is shown in Figure 1.3:

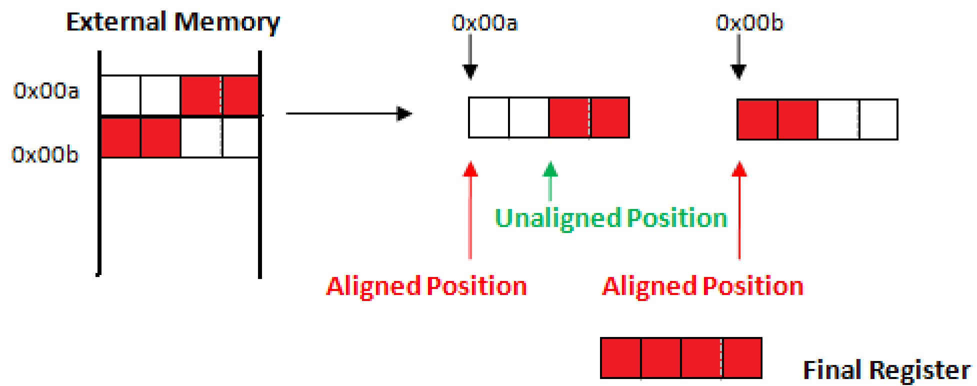


Figure 1.3.*Alignment process*

We must read the aligned memory word that is located before the unaligned position and discards the unnecessary bytes, then read the aligned memory word that is located after the unaligned position and discards the unnecessary bytes and finally merge the necessary bytes of the two previous words.

Most SIMD processors provide access to only contiguous data in memory, with strong alignment restrictions because they have limited memory architecture. These architectures, either do not provide any hardware support for unaligned accessed or provide it but with degrade results. Therefore, the programmer usually ends doing the alignment in software which implies an extra-overhead and degrades the performance significantly. For example, applications like video coding and decoding that use motion estimation (ME) and motion compensation (MC) algorithms, require performing unpredictable alignments.

Nowadays, there are wide consensus about the importance of non-aligned accesses for video applications (not having an efficient support for unaligned accesses degrade the performance significantly), but most of the current SIMD architectures that support unaligned accesses have restrictions and limitations.

Because the applications require increased resources, **DMA Unit** is an essential feature of all modern computers, as it allows devices to transfer data without subjecting the CPU to a heavy overhead, thus increasing application performance. A DMA transfer essentially copies a block of memory from one device to another. With DMA, the CPU would initiate the transfer, do other operations while the transfer is in progress, and receive an interrupt from the DMA controller once the operation has been done. A typical usage of a DMA is to transfer a block of memory from/to an external memory or from/to a buffer on the device. A DMA is usually used in high performance embedded systems.

Because the instructions explained above, the alignment process subjects the CPU to a heavy overhead. There are some applications, like video processing that require an efficient support for unaligned accesses, degrading the performance significantly. Due to the DMA allows devices to transfer data without subjecting the CPU to this heavy overhead, a possible solution would be to design an **‘Enhanced DMA Unit’** capable of performing this alignment of

data. This new DMA will save many cycles because now the alignment is performed in the DMA and the CPU can execute other part of the application code in parallel.

The idea of this project is to implement a DMA that in addition to perform this alignment can also perform some data permutations, for example padding. The instructions explained before can continue be using because once we have the data in internal memory, these instructions will be responsible for performing any required alignment. But in many case, we will use an **'Enhanced DMA Unit'** because then we will be able to remove instructions code and therefore reduce instructions cycles.

One example where the **'Enhanced DMA Unit'** can reduce fairly the overload for the CPU would be in applications like video coding and decoding that use ME and MC algorithm. This algorithm requires to move blocks of data from an external memory to a local memory faster, and to align theses blocks to place them correctly in the local memory. Moreover in case of reading out from a reference frame some padding process should be executed.

The objectives of this project are the follows:

- Analyze the transfer's requirements of actual video coding standards and to develop a parameterized Direct Memory Access unit model.

- Considering a particular video coding task, i.e. motion estimation, an improved version has to be implemented, by adding extra functionality in the DMA unit in order to decrease the required execution time.

- An exhaustive analysis of the performance results should be done, indicating possible future modifications.

This project is organized as follows. Chapter 2 presents the VLIW processors and the DMA basics. Chapter 3 introduces H.264/AVC which is a standard for video compression and also introduces motion estimation (ME) algorithm. Chapter 4 explains the DMA proposal for the 'Enhanced DMA Unit'. After that, in Chapter 5, the evaluation of the proposal DMA Unit using Motion Estimation algorithm is shown. Finally, conclusions are presented in Chapter 6.

Chapter 2

VLIW processors and DMA Basics

This section presents the VLIW processors compared with other types of processors. After that, we are going to study and analyze Moai4k processor, which is a processor that implements a VLIW architecture. Finally, the basics of Direct Memory Access Units (DMA) are presented in this section.

2.1. - Basics of VLIW/SIMD processors

This section, presents the VLIW processors compared with Superscalar processors. After that, VLIW benefits are explained. Finally, SIMD processors are presented.

2.1.1. - VLIW vs. Superscalar processors

Superscalar is the term used to describe a type of microarchitecture processor capable of running more than one instruction per clock cycle. The term is used as opposed to scalar microarchitecture that only is able to execute an instruction per clock cycle. A superscalar processor presents the following characteristics:

- In a superscalar processor, it is the hardware at runtime which is responsible for planning the instructions. The microarchitecture superscalar uses the instruction parallelism in addition to the parallelism flow, the latter through the pipeline structure.

- In a superscalar processor, the processor handles more than one instruction at every stage. The maximum number of instructions in a particular stage of the pipeline is called degree and a processor superscalar grade 2 in reading (fetch) can read up to two instructions per cycle.

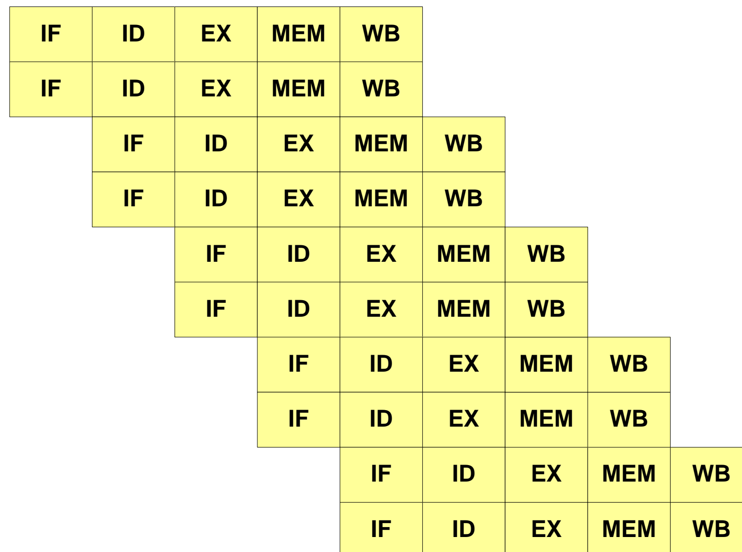


Figure 2.1.*Simple superscalar pipeline. By fetching and dispatching two instructions at a time, a maximum of two instructions per cycle can be completed.*

- A superscalar processor is able to execute more than one instruction simultaneously only if the instructions do not present any kind of dependence (hazard).

A statically scheduled superscalar must verify in the issue packet for any dependence between instructions, as well as between any issue candidate and any instruction already in the pipeline. A type of superscalar processor requires so much compiler assistance to achieve good performance. In contrast, a dynamically-scheduled superscalar requires significant hardware costs but less compiler assistance.

An alternative to the superscalar processors is to rely on compiler technology not only to minimize the potential data hazard stalls, but to actually format the instructions in a potential issue packet so that the hardware need not check explicitly for dependences.

With this new approach the compiler may be required to avoid dependences within the issue packet, or if there are dependences to indicate when may be presented. This new approach offers good performance with the potential advantage of simpler hardware.

The first multiple-issue processors that required the instruction stream to be explicitly organized to avoid dependences used wide instructions with multiple operations per instruction. For this reason, this architectural approach was named **VLIW**, standing for **V**ery **L**ong **I**nstruction **W**ord, and denoting that the instructions, since they contained several instructions, were very wide. A VLIW processor presents the following characteristics:

- VLIWs use multiple functional units that must be independent. A VLIW packages the multiple operations into one very long instruction.

- VLIW leaves all the work of planning the code in the hands of the programmer / compiler, as opposed a superscalar processor, which is the hardware at runtime which plans instructions.

Example: Assume the following program:

1. Multiply Reg1 and Reg2 save in Reg3
2. Add Reg3 and Reg4 save in Reg5
3. Subtract Reg1 and Reg4 save in Reg6

a) Execution on a Superscalar machine

In this program, the planner code that would see that the second instruction depends on the first (until it is not calculated Reg3 cannot run the sum), and that change in the third instruction is independent of the other two. Therefore, the processor will begin simultaneously multiplication and subtraction in different units and, once finished the multiplication, will execute the sum. All this scheduling work is conducted by an internal circuitry in the microprocessor.

b) Execution on a VLIW

In this case all the work will be made by the compiler and the code would be:

1.	MULT(Reg1, Reg2, Reg3)	-	REST(Reg1, Reg4, Reg6)
2.	-	SUM(Reg3, Reg4, Reg5)	-

Figure 2.2.*Code executed for a VLIW processor*

In each instruction specifies the status of each and every one of the functional units of the system. In the first instruction, it would be carried out multiplication and subtraction and in the second the sum.

2.1.2. - VLIW benefits

VLIW offers the followings benefits:

a) Offers the potential advantage of simpler hardware while still exhibiting good performance through extensive compiler optimization.

b) Reducing the number of instructions in the programs.

c) Has the ability to use a more conventional, and typically less expensive, cache-based memory system.

d) Has the potential to extract some amount of parallelism from less regularly structured code.

2.1.3. - SIMD processors

Vector processors deal with multiple pieces of data in the context of one instruction. This contrasts with scalar processors, which deal with one piece of data for every instruction. These two schemes of dealing with data are generally referred to as SISD (single instruction, single data) and SIMD (single instruction, multiple data), respectively. The great utility in creating CPUs that deal with vectors of data lies in optimizing tasks that tend to require the same operation (for example, a sum) to be performed on a large set of data. Some classic examples of these types of tasks are multimedia applications (images, video, and sound), as well as many types of scientific and engineering tasks. Whereas a scalar CPU must complete the entire process of fetching, decoding, and executing each instruction and value in a set of data, a vector CPU can perform a single operation on a comparatively large set of data with one instruction. Of course, this is only possible when the application tends to require many steps which apply one operation to a large set of data.

The repertoires of SIMD consist of instructions that apply the same operation on a set more or less big of data. A single control unit sends instructions to different processing units. All processors receive the same instruction from the control unit, but operating on different data sets. In other words the same instruction is executed in a synchronous manner by all processing units.

For example: *Typical operations encountered such as SAD require the same operation to be performed on multiple pixels or data units with no dependencies involved. Therefore, the same instruction can be used to perform this task simultaneously on all the data units.*

In most embedded media processors and in microprocessors for desktop computers, the common approach for dealing with the requirements of digital video processing, and the other multimedia applications, has been the extension of the Instruction Set Architecture (ISA) with SIMD instructions.

2.2. - Moai4k

Moai4k is a processor, created in Rapanui project, that implements a VLIW architecture. This section study and analyze this processor, and presents the generic VLIW architecture [2].

2.2.1. - Introduction

Multimedia processing has increasingly applications and because of the wide variety of applications such as current video applications, multimedia cell phones, high-definition interactive television… are necessary continuous improvements requiring sophisticated multimedia operations, which imply a high demand of processing power.

Because of this and to achieve better performance, have resulted in a variety of design methodologies that cause the processors to behave less linearly and more in parallel. VLIW architecture is being very used in the last years due to the capability of exploiting parallelism in multimedia applications. VLIW architectures execute multiple operations within a single long instruction word. Therefore, to allow this concurrency, multiple parallel functional units have to be implemented.

Using subword parallelism in a VLIW architecture can be achieved a significant improvement of results. VLIW can be also improved with instruction set extensions, specialized functional units and instruction level parallelism. VLIW architectures require instruction scheduling to be performed at compile time to assemble the long instruction words.

For designing a media processor there are multiple architectural features that should be taken into account. For example, increasing the amount of instruction level parallelism (ILP) requires more registers to hold the operands and results of those instructions. This lead to a complex register file with a high number of read/write ports to allow the concurrence access. On the other hand large multiport memories introduce performance degradation. Therefore, partitioned register should be considered for this kind of processor architectures.

2.2.2. - A Generic VLIW architecture

This generic VLIW architecture covers the basics requirements for media processing. Two important characteristics have been considered:

1) A fully parameterized architecture: easily extendable by adding or modifying functional units.
2) A novel register file structure avoids a high number of required ports when there are a big number of parallel operations.

For this generic VLIW architecture, a parameterized simulator was written in Open Vera and standard C. Figure 2.3 shows an overview of the design space exploration environment.

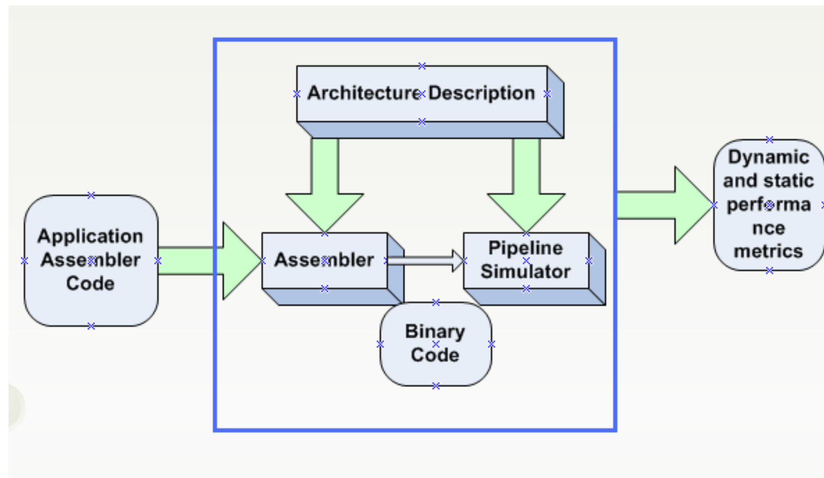


Figure 2.3.*Rapanui Design Space Exploration Environment*

a) Vector Unit Structure

The basic structure of this VLIW architecture, has been specially designed for efficient processing of blocks of data or macroblocks normally used in video coding algorithms. This basic structure will be hereafter called Vector Unit (VU).

Figure 2.4 shows a Vector Unit architecture:

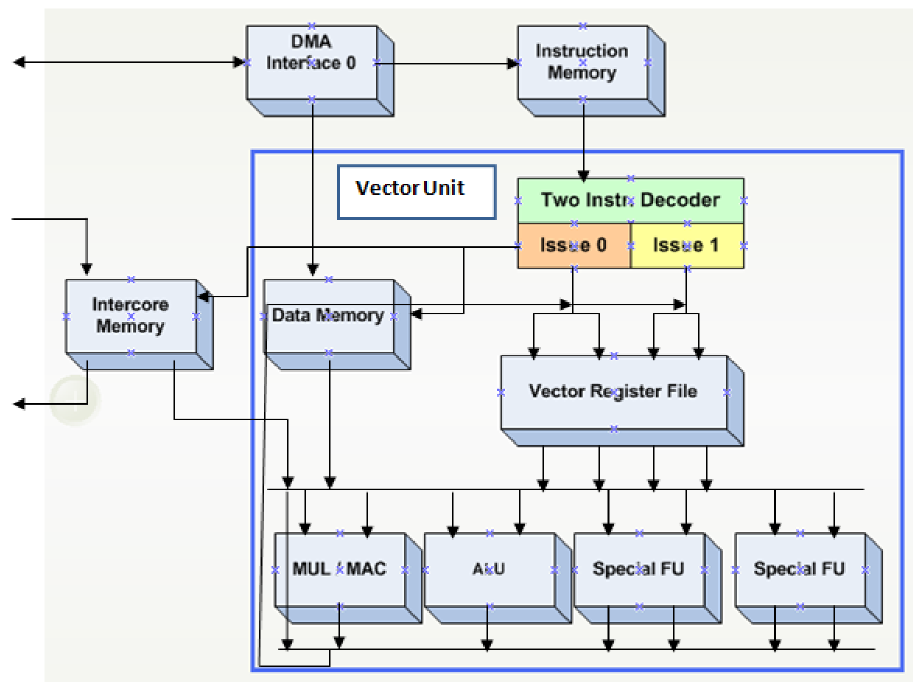


Figure 2.4.*Vector unit architecture*

The VU comprises a flexible datapath controlled by a 64-bit dual-issue VLIW (two 32-bit operations). Moreover, the datapath implements subword parallelism (e.g. for a 64 bit datapath, operands splitable into one 64-bit subword, two 32-bit subwords, four 16-bit subwords or eight 8-bit subwords) in almost every functional unit.

The control path is divided initially into 5 different basics stages:

- Instruction Fetch(IF)
- Instruction Decode(DE)
- Register Access(RA)
- Execution(EX)
- Write Back(WB)

The EX stage, as shown in Figure 2.5, can be subdivided into more stages depending on the characteristics, e.g. latency, of the functional units.

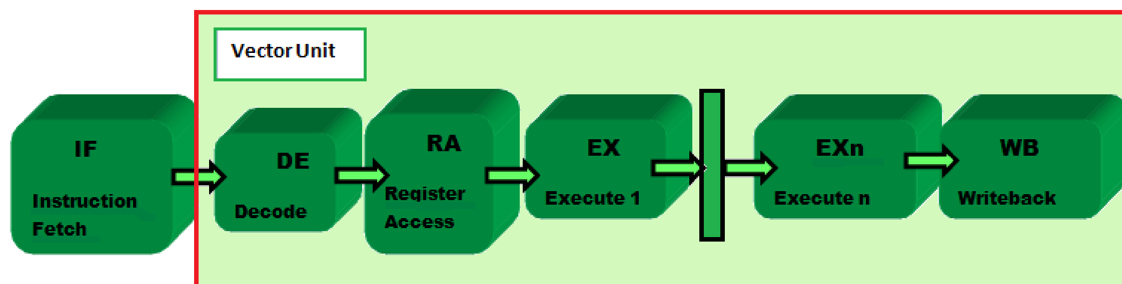


Figure 2.5.*Vector Unit pipeline*

The VU implements a configurable global address map which allocates a 64-bit dual port instruction memory with configurable size. There is also a dual port data memory with configurable size. Each memory is accessed by the processor itself and DMA unit, which performs data transfers in background between memories and external slaves through a configurable bus system.

b) Specialized instructions and functional units

The VU implements an orthogonal instruction set that contains several extensions for typical video processing computations, e.g. data formatting, different kinds of rounding, multiply-and-accumulate operations. To facilitate adding new instructions and functional units, most instructions use a common bit-pattern, as shown in Figure 2.6:

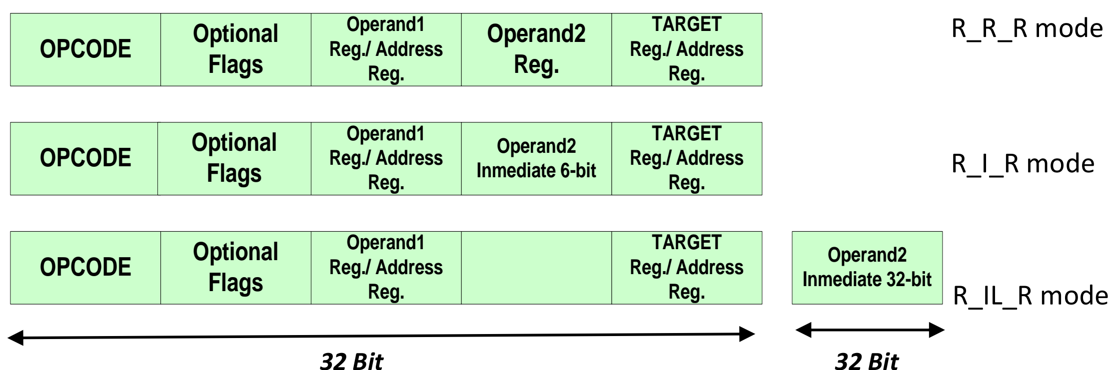


Figure 2.6.*Instructions types*

Move and flow operations use different patterns. Access to memory or special registers, e.g. address registers for indirect addressing mode or configuration register located in the functional units, are performed by move operations.

Functional units are represented by a set of characteristics, e.g. latency, which is expressed in number of execution stages, and type and number of operands, which are related with the number of read and write ports required in the register file.

Table shows details about the functional units implemented: operations performed, latency required and number of register and number of register written in the write-back (WB) stage. MV, LS and FLOW are not considered functional units.

Functional Units	Operations performed	Latency (typical)	Write Registers
AU	Arithmetic	1	1
MAC	Multiplication and Multiply and Accumulation	2	1 or 2
LU	Logic	1	1
SR	Shift and Round	1	1
CMM	Clip, Max and Min	1	1
FOR	Data formatting	1	1
MV	Move	1 or 3	0 or 1
LS	Load and store	1	0 or 1
FLOW	Control flow	2 or 3	0 or1

Table 2.1.*Details about the functional units implemented*

Another interesting feature is that a VU can execute all kinds operations either of its two issues. Therefore, two operations that execute the same function can only be scheduled in the same instruction, if the needed functional unit has been replicated for the desired architecture configuration.

c) *Multiple Vector Unit and Partitioned Register File*

Using a higher degree of parallelism in some applications, results in considerably more performance. This can be achieved by increasing the number of VUs to two or even three within the architecture as shown in Figure 2.7:

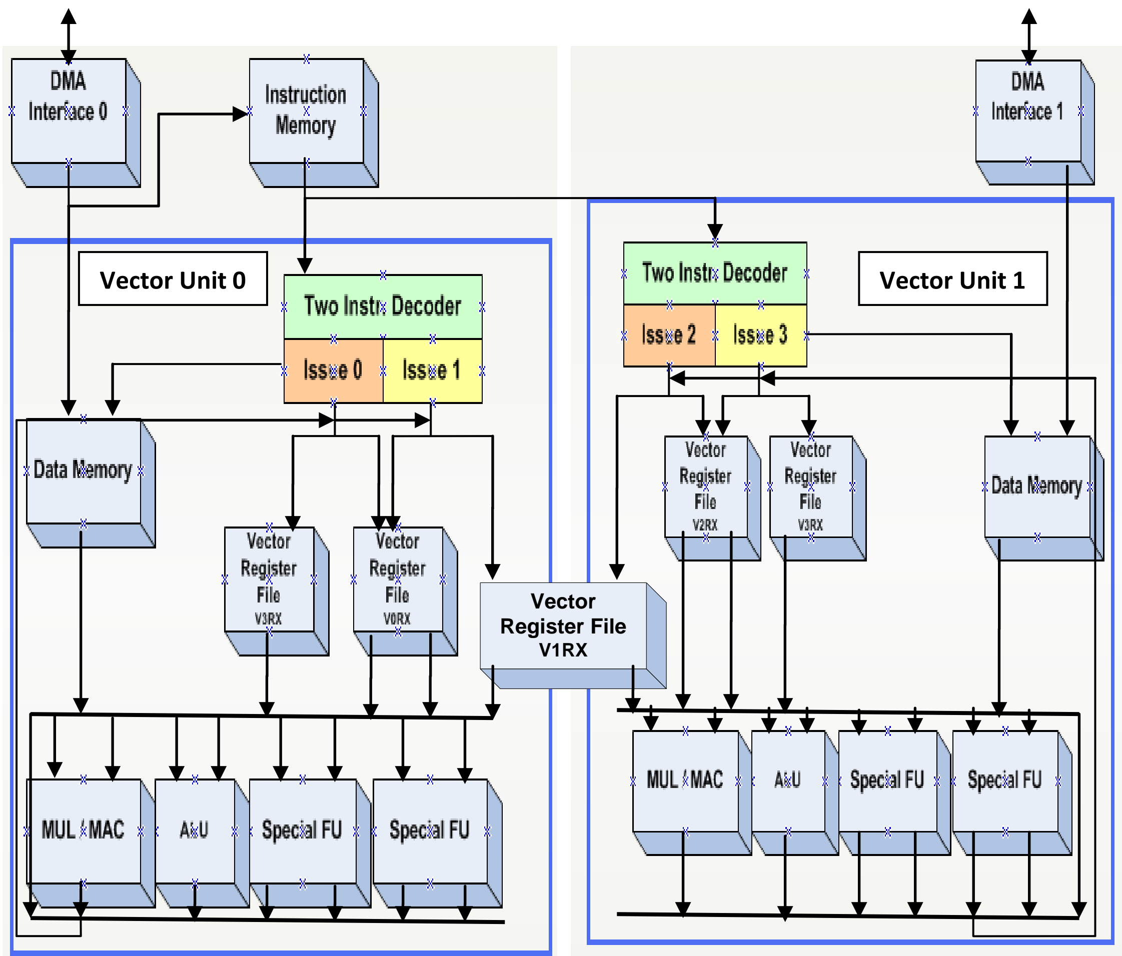


Figure 2.7.*Dual Vector Unit architecture*

The pipeline configuration is mostly the same, with the exception of the IF stage, which is now common for all VUs. The other stages are independent for each VU.

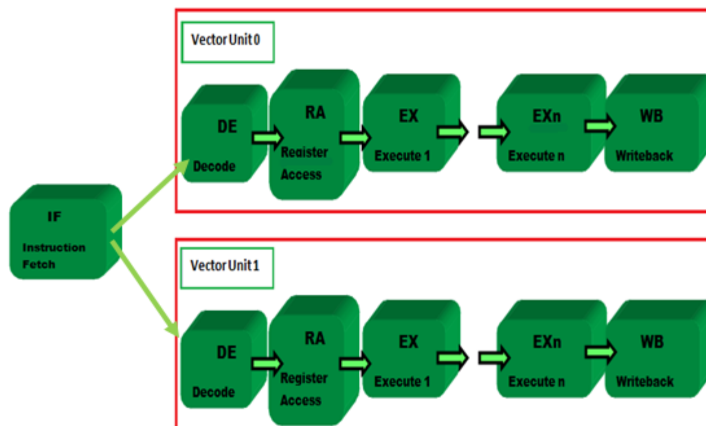


Figure 2.8.*Dual Vector Unit pipeline architecture*

In this new architecture, every VU has its own data memory and DMA, allowing multiple transfers in parallel.

These changes implicate a redesign of the register file structure:

- Having a unique register file for two VUs (with two issues each) results in implementing many ports memory, which is not practicable.

- Having separated register files for each VU results in inefficient data transfers.

Therefore, the register file structure for multiple VUs can be implemented with a ring structure. This new register file structure increases the operation parallelism allowing to parallelize operations that require same functional units and same input data. Input data is not duplicated, instead it is stored in the register file located between VUs. This structure occupies less area and improves timing performance of the hardware implementation.

2.3. - Basics of Direct Memory Access Units

This section, presents the Directs Memory Access Units, the operation of a DMA Unit, the DMA Unit strategies transfer, how to program a DMA controller, and finally a classification of the DMA Units.

2.3.1. - Introduction

The processor core can make multiple operations in a cycle such as data fetches, data stores and pointer increments/decrements. Moreover, the core, moving data into and out of the register file, can lead data transfer between internal and external memory spaces.

The issue is that for achieving high performance in a specific application it is necessary to interrupt the core to perform the transfers. To solve this, it is possible to use a DMA. Processors use DMA to release the core from these transfers between internal/external memory and peripherals, or between memory spaces.

2.3.2. - Operation of a DMA

Direct memory access (**DMA**) is a feature of modern computers that allows certain hardware subsystems within the computer to access system memory for reading and/or writing independently of the central processing unit. Many hardware systems use DMA including graphics cards, disk drive controllers, sound cards, and network cards. DMA is also used for intern-chip data transfer in multi-core processors, especially in multiprocessor system-on-chips where each core is equipped with its local scratchpad memory and DMA is used for transferring data to and from these local memories. Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without a DMA channel.

Using a DMA Unit, in a programmed input/output mode, the CPU would initiate the transfer, makes other operations while the transfer is in progress, and receive an interrupt from the DMA controller once the operation has been done. This is especially useful in real-time computing applications where not stalling behind concurrent operations is critical. If we don`t use a DMA, for the entire duration of the read or write operation the CPU is fully occupied and is thus unavailable to perform other work.

DMA allows devices to transfer data without subjecting the CPU to a heavy overhead. Otherwise, the CPU would have to copy each piece of data from the source to the destination. During this time the CPU would be unavailable for any other tasks involving CPU bus access, although it could continue doing any work which did not require bus access.

A typical usage of DMA is copying a block of memory from system RAM to or from a buffer on the device. Moreover DMA is used in high performance embedded systems. DMA is also used in functionalities such as network audio playback, packet routing and streaming video.

Figure 2.9 shows a common interaction between the processor and the DMA controller.

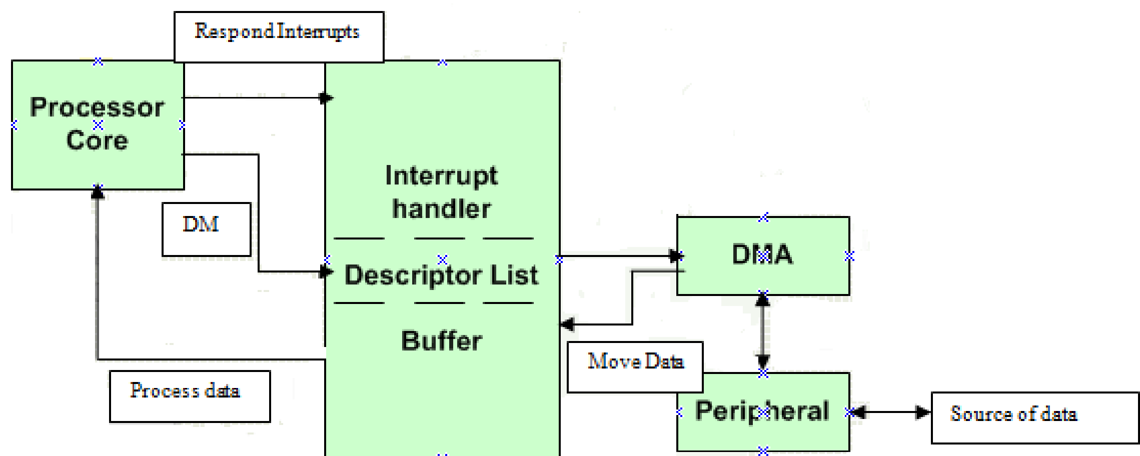


Figure 2.9.*Interaction between the processor and DMA controller*

The steps allocated to the processor involve enabling interrupts, setting up the transfer, and running code when an interrupt is generated. The lines between peripheral and the memory indicate operations the DMA controller makes to move data independent. Finally, the interrupt input back to the processor can be used to signal that data is ready for processing.

The DMA controller can perform several types of data transfers:

1) Moving to peripherals

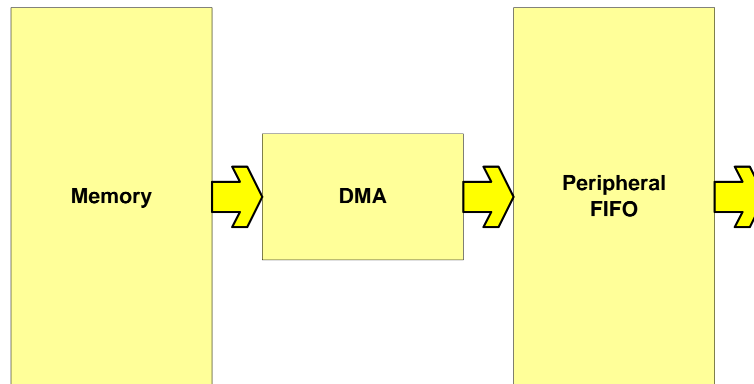


Figure 2.10.*Data transfer to a peripheral*

2) Moving from peripherals

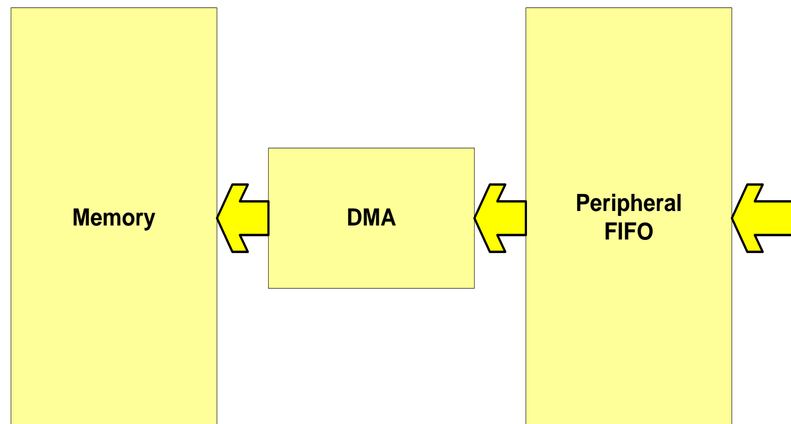


Figure 2.11.*Data transfer from a peripheral*

3) Moving from one memory space to another

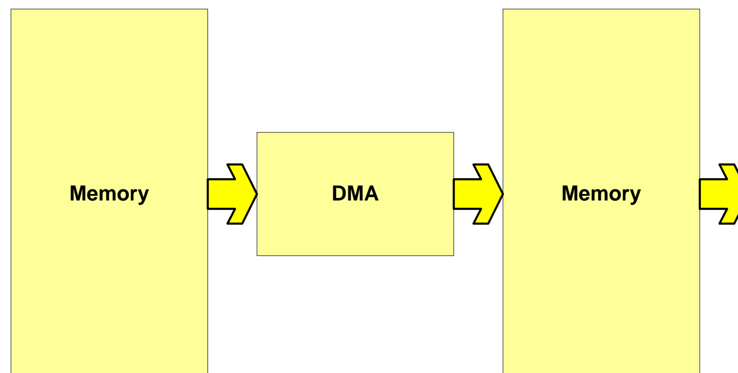


Figure 2.12.*Data transfer between memories*

For example, source video might flow from a video to external memory, because the working buffer size is too large to fit into internal memory. We don´t want to make the processor fetch pixels from external memory every time we need to perform a calculation, so a memory-to-memory DMA can be used for more efficient access time.

A DMA transfer can involve data and code. We can use code overlays to improve performance, configuring the DMA controller to move code into instruction memory before execution. The code is usually staged in larger external memory.

Figure 2.13 shows one DMA bus structure, where the DMA Core Bus connects the controller to internal memory, the DMA External Bus connects the DMA controller to external memory and DMA Access Bus connects to the peripherals. Moreover an additional DMA bus set is also available when on-chip memory is presented, in order to move data within the processor's internal memory spaces.

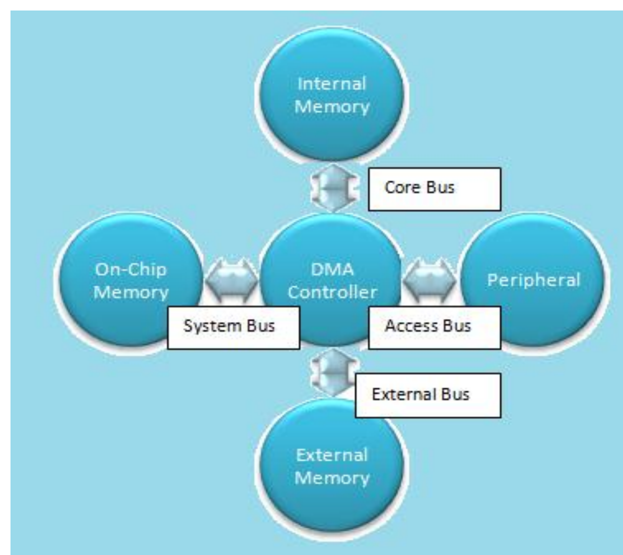


Figure 2.13.*DMA Unit bus structure*

Each DMA channel on a DMA controller (if the DMA controller has channels, in our case the DMA has a queue) has a programmable priority associated with it. If more than one channel requests the DMA bus in a single cycle, the highest priority channel wins access.

When more than one DMA controller is presented on a processor, the channels from one controller can run at the same time as channels on the other controller. This is possible when both don´t try to access the same resource, if both DMA controllers try to access the same resource, arbitration must take place (one of the controllers can be programmed to a higher priority than the other).

2.3.3. - DMA transfer strategies

The following are different techniques to make data transfer:

1) **'Cycle stealing'**: is based in use one or more CPU cycles per instruction that runs (hence the name). So it is achieved high availability of the system bus to the CPU, although, consequently, the transfer of data is considerably slower. This method is being used routinely because the interference with the CPU is very low.

2) **'Burst DMA'**: consists of sending the block of data requested by a burst, occupying the system bus until the end of transmission. So gets the maximum speed, but the CPU can not use the bus during that time, so it would remain inactive.

3) **'Transparent DMA'**: uses the system bus when there is confidence that the CPU does not require it, such as in the stages of the execution process of the instructions where never uses because the CPU performs internal tasks (eg phase decoding of the instruction). Thus, as its name suggests, the DMA will remain transparent to the CPU and the transfer will be done without jeopardizing the relation CPU-system bus. As a disadvantage, the transfer rate is the lowest possible.

4) **'Scatter-gather DMA'**: allows data transfer to several areas of memory in a single transaction DMA. It is equivalent to chaining multiple requests DMA simple. Again, the aim is to liberate the CPU of copy tasks of data and multiple interruptions of data input/output.

2.3.4. - How to program a DMA Controller

To perform a DMA transfer, starting source and destination address is required. In the simplest case of a memory DMA, we need to specify the DMA controller the source address, the destination address and the number of words to transfer. This type of transaction is called simple one-dimensional (1D) transfer with a unity 'stride'. With a unity stride, the address increments by 1 byte for 8-bit transfer, 2bytes for 16-bit transfers, and 4 bytes for 32 bits transfers. Figure 2.14 shows a 1D DMA Unit with unity stride:

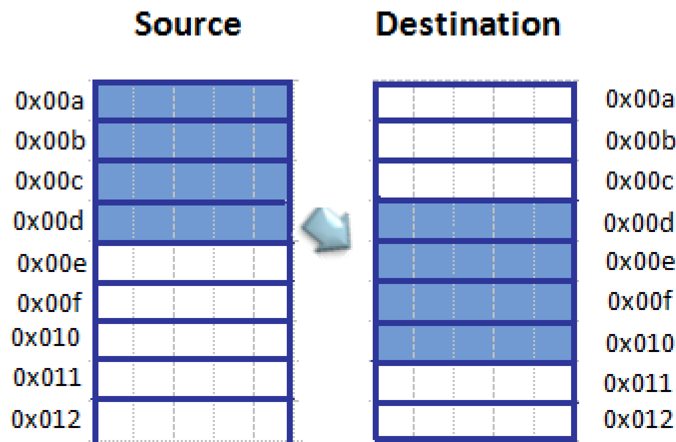


Figure 2.14.*1D DMA Unit with unity stride*

If we want to add more flexibility to a one-dimensional DMA Unit, we only have to change the stride. For example, with non unity strides, we can skip addresses in multiples of the transfer sizes. That is, specifying a 8-bit transfer and striding by 4 samples results in an address increment of 4 bytes (four 8-bit words) after each transfer. Figure 2.15 shows this case:

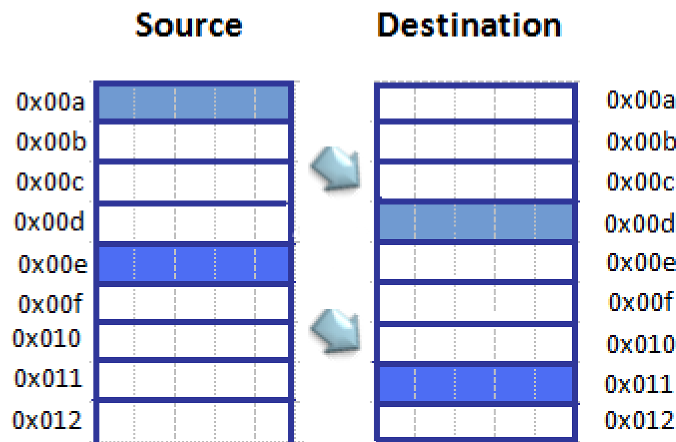


Figure 2.15.*1D DMA Unit with non-unity stride*

In the case of a peripheral DMA, the peripheral's FIFO serves as either the source or the destination. If the peripheral serves as the destination, a memory location serves as the source address, in otherwise (the peripheral serves as the source), a memory location (internal or external) serves as the destination address.

We can describe two registers, XCOUNT and XMODIFY.

- **XCOUNT**: is the number of transfers that need to be made

- **XMODIFY**: is the number of bytes to increment the address pointer after the DMA controller moves the first data element. Is always expressed in bytes.

The two-dimensional (2D) capability is even more useful than 1D DMA. 2D DMA is very used especially in video applications, 2D DMA is an extension to 1D DMA. In this configuration we program two registers more, YCOUNT and YMODIFY:

- **YCOUNT**: is the number of transfers

- **YMODIFY**: is specified as a number of bytes to increment the address pointer

2D DMA can be seen as a nested loop, where the inner loop is specified by XCOUNT and XMODIFY, and the outer loop specified by YCOUNT and YMODIFY. A 1D DMA can then be viewed simply as an 'inner loop' of the 2D transfer of the form:

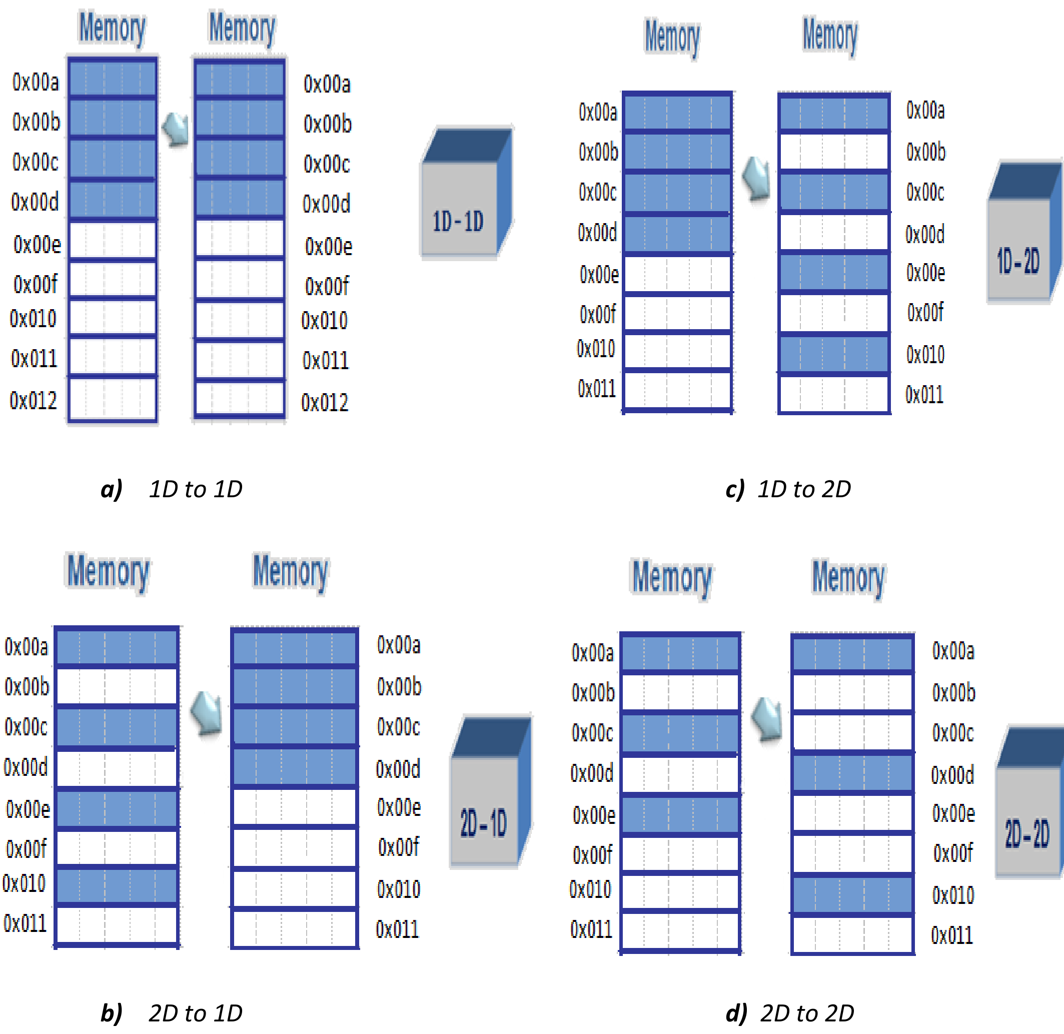
```
for y =1 to YCOUNT STEP YMODIFY /*2D with outer loop*/
    for x =1 to XCOUNT STEP XMODIFY /*1D inner loop*/
        {/*Loop goes here*/
        }
```

Figure 2.16.*1D DMA Unit as an 'inner loop' of the 2D DMA transfer*

While YMODIFY determines the stride value of DMA controller takes every time XCOUNT decrements, YMODIFY determines the stride taken whenever YCOUNT decrements. YMODIFY can be negative, which allows the DMA controller to wrap back around to the beginning of the buffer.

For a memory DMA, the 'memory side' of the transfer can be 1D-to-1D transfer, a 1D-to-2D transfer, a 2D-to-1D transfer, and a 2D-to-2D transfer. The only constraint is that the total number of bytes being transferred on each end of the DMA transfer block has to be the same.

Figures 2.17 show different Memory DMA configurations:



a) 1D to 1D *c) 1D to 2D*

b) 2D to 1D *d) 2D to 2D*

Figure 2.17.*Posible Memory DMA configurations*

Peripheral DMA offers less flexibility, the 'memory side' of the transfer can be either 1D or 2D. On the peripheral side it is always a 1D transfer. The only constraint is that the total number of bytes transferred on each size (source and destination) of the DMA must be the same.

Example

Consider a 3-pixel (per line) x 4-line array, with byte-sized pixel values, ordered as shown in Figure 2.18:

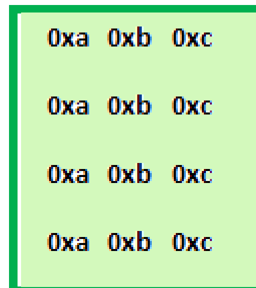
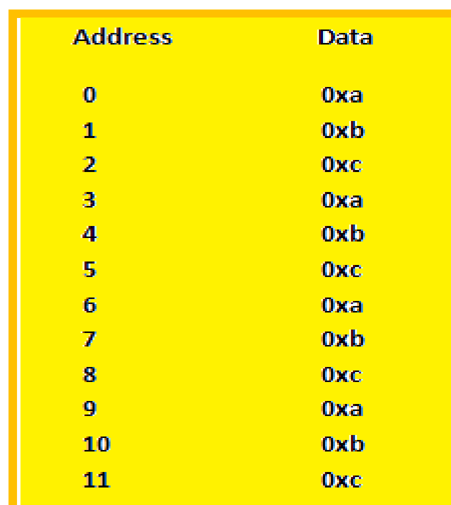


Figure 2.18.*Source array*

While this data is shown as a matrix, it appears consecutively in memory as shown in the Figure 2.19:



Address	Data
0	0xa
1	0xb
2	0xc
3	0xa
4	0xb
5	0xc
6	0xa
7	0xb
8	0xc
9	0xa
10	0xb
11	0xc

Figure 2.19.*Posible Memory, it appears consecutively en memory*

We want now to create using the DMA controller, the array shown in the Figure 2.20:

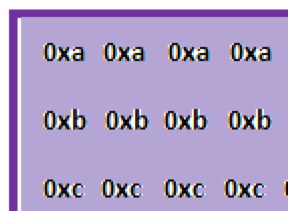


Figure 2.20.*Destination array*

Source and destination word transfer size = 1 byte per transfer.

We can use a memory DMA, with a 2D-to-1D transfer configuration. First I will explain the values of the registers of the source.

The source is 2D and the array size is 4 lines x 3 pixels/line, therefore XCOUNT is 4 and YCOUNT is 3.

We want to take the first value (0xa) and skip 3 bytes to the next value of 0xa. We will repeat this four times (Source XCOUNT = 4). The value of the source XMODIFY is 3, because that is the number of bytes the controller skips over to get to the next pixel (including the first pixel). XCOUNT decrements by 1 every time a pixel is collected. When the DMA controller reaches the end of the first row, XCOUNT decrements to 0, and YCOUNT decrements by 1.The value of YMODIFY on the source side then needs to bring the address pointer back to the second element in the array (0xb). At the instant this happens, the address pointer is still pointing to the last element in the first row (0xa). Counting back from that point in the array to the second pixel in the first row, we traverse back by 11 elements. Therefore, the source YMODIFY = -11.

Therefore the source DMA register setting for this transfer are:

```
SOURCE

XCOUNT = 4

XMODIFY = 3

YCOUNT = 3

YMODIFY = -11
```

Figure 2.21.*Source DMA registers*

Now the values of the registers of the destination will be explained.

Due to we will use a 1D transfer to fill the destination buffer, we only need to program XCOUNT and XMODIFY on the destination side. In this case, the value of XCOUNT is set to 12, because that is the number of bytes that will be transferred. The YCOUNT value for the destination side is simply 0, and YMODIFY is also 0.

Therefore the destination DMA register setting for this transfer are:

```
DESTINATION

XCOUNT = 12

XMODIFY = 1

YCOUNT = 0

YMODIFY = 0
```

Figure 2.22.*Destination DMA registers*

2.3.5. - DMA Classification

There are two main classes of DMA transfer configuration: **Register mode** and **Descriptor mode**. The Table 2.2 shows the type of information into the DMA controller, which is the same regardless of the class of DMA.

DMA Registers	
Next Descriptor Pointer (lower 16 bits)	Address of next descriptor
Next Descriptor Pointer (upper 16 bits)	Address of next descriptor
Start Adress (lower 16 bits)	Start Address (source or destination)
Start Adress (upper 16 bits)	Start Address (source or destination)
DMA Configuration	Control information (enable, interrupt selection,1D vs. 2D)
X_Count	Number of transfer in inner loop
X_modify	Number of bytes between each transfer in inner loop
Y_Count	Number of transfer in outer loop
Y_Modify	Number of bytes between end of inner loop and start of outer loop

Table 2.2._DMA registers_

a) Register-based DMA

In a register-based DMA, the processor directly programs DMA control register to initiate a transfer. Registers don’t need to keep reloading from descriptors in memory, and the core does not have to maintain descriptors. This DMA provides the best performance.

b) Descriptor-based DMA

Require a set of parameters stored within memory to initiate a DMA sequence. The descriptor contains all of the same parameters normally programmed into the DMA control register set. In descriptor-based DMA operations, we can program the DMA to automatically set up and start another DMA transfer after the current sequence completes. The descriptor-based model provides the most flexibility in managing system’s DMA transfers.

Chapter 3

H.264/AVC and Motion Estimation

This section introduces H.264/AVC which is a standard for video compression and also introduces motion estimation (ME) algorithm.

3.1. - H.264/AVC Coding Standard

H.264 is a standard for video compression. It is also known as MPEG-4 Part 10, or MPEG-4 AVC (for Advanced Video Coding). H.264 is the result of the collaboration between the ISO/IEC Moving Picture Expert Group and the ITU-T Video Coding Expert Group.

The intent of the H.264/AVC project was to create a standard capable of providing good video quality at substantially lower bit rates than previous standards (e.g. half or less the bit rate of MPEG-2, H.263, or MPEG-4 Part 2), without increasing the complexity of design so much that it would be impractical or excessively expensive to implement. An additional goal was to provide enough flexibility to allow the standard to be applied to a wide variety of applications on a wide variety of networks and systems, including low and high bit rates, low and high resolution video, broadcast, DVD storage, RTP/IP packet networks, and ITU-T multimedia telephony systems.

In this section H.264/AVC Coding Standard is studied, and the encoder and decoder will be explained.

3.1.1. - Introduction

H.264/AVC is the latest standard in a sequence of the video coding standards H.261, MPEG-1 Video, MPEG-2 Video, H.263, MPEG-4 Visual or part 2. These previous standards reflect the adaptation of video coding to different applications and networks and the technological progress in video compression. Applications range from video telephony to consumer video on CD and broadcast of standard definition or high definition TV.

Nowadays, the importance of new network access technology (cable modem, Xdsl, and UMTS) is increasing and therefore creating demand for the new video coding standard H.264/AVC, providing enhanced video compression performance in view of non-interactive applications like storage, broadcast and streaming of standard definition TV where the focus is on high coding efficiency and interactive applications like video telephony requiring a low latency system.

Comparing the H.264/AVC video coding tools to the tools of previous video coding standards, H.264/AVC brought in the most algorithmic discontinuities in the evolution of standardized video coding.

The standardization of the first version of H.264/AVC was completed in May of 2003. The JVT (Joint Video Team) then developed extensions to the original standard that are known as the Fidelity Range Extensions (FRExt). H264/AVC achieved a leap in coding performance that was not foreseen just five years ago.

Figure 2.23 shows the scope of video coding standardization, where only the syntax and semantics of the bit stream and the processing that the decoder needs to perform when decoding the bit stream into video are defined:

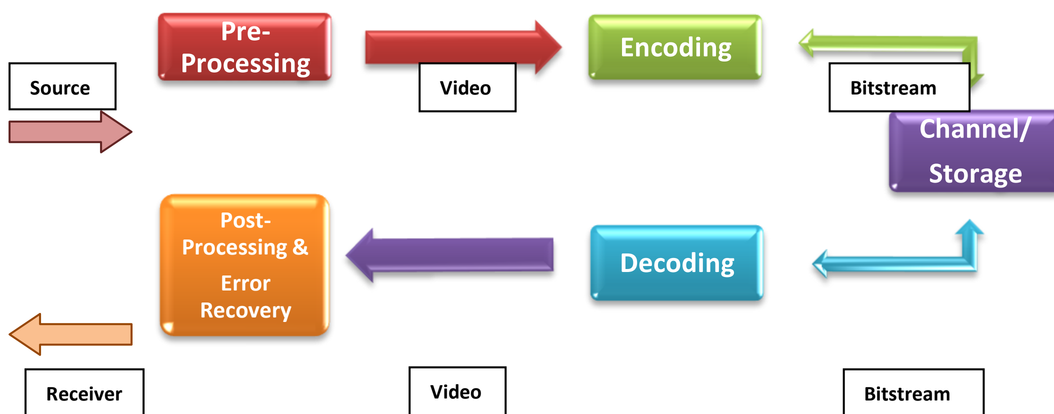


Figure 3.1.*Scope of video coding standardization*

The steps of the previous scheme would be as follows:

- A digital video is captured from a camera or synthesized using appropriate tools.
- Optional pre-processing: the sender might choose to preprocess the video using format conversion or enhancement techniques.
- Encoder: encodes the video and represents the video as a bit stream.
- Transmission of the bit stream over a communications network.
- Decoder: decodes the video.
- Optional post-processing: step which might include format conversion, filtering to suppress coding artifacts, error concealment, or video enhancement.

The standardization of the bit stream and the decoder by the H.264/AVC preserves the fundamental requirement for any communications standard—interoperability.

Manufactures of video decoders can only compete in areas like cost and hardware requirements. Optional post-processing of the decoded video is another area where different manufactures can compete creating competing tools to create a decoded video stream optimized for the targeted application. The standard does not define how encoding or other video pre-processing is performed thus enabling manufactures to compete with their encoders in areas like coding efficiency, cost, error resilience and error recovery or hardware requirements.

Not only coding efficiency is relevant for efficient transmission in different environment, but also the seamless and easy integration of the coded video into all current future protocol and network architectures. This includes Internet and wireless network expected to be a major application for the new video coding standard.

3.1.2. - Video Encoder

An encoder is a device used to change a signal or data into a code. In H264/AVC standard the encoder encodes the video and represents the video as a bit stream for the transmission over a communications network. Figure 2.24 shows a generalized diagram of a video encoder with motion compensation:

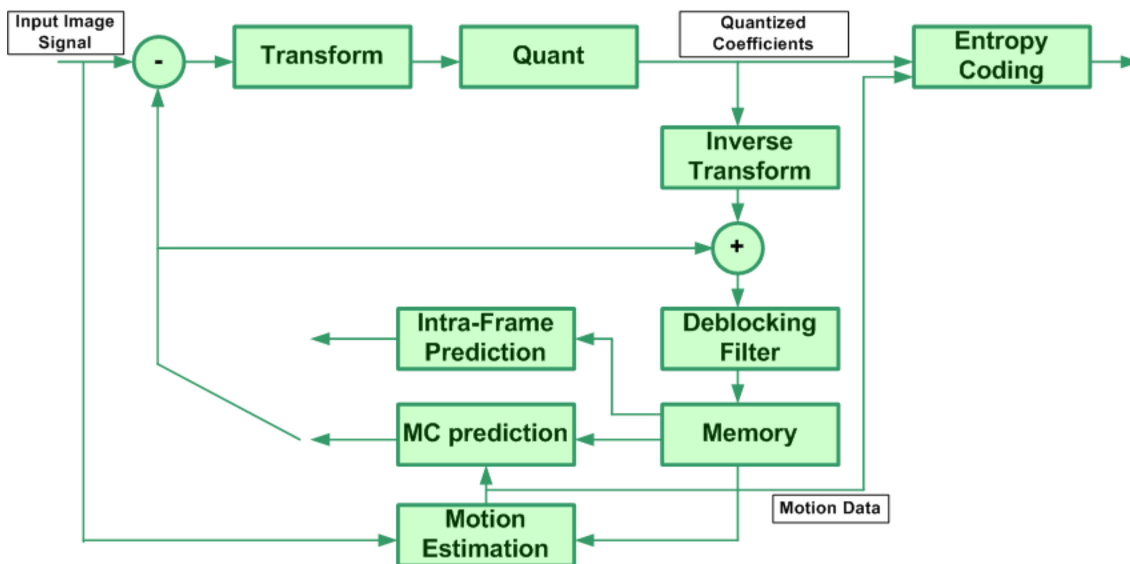


Figure 3.2.*Block diagram of a video encoder with motion compensation*

The input image is divided into macroblocks and each of these macroblocks is what we introduce into our encoder. Each machroblocks consists of the three components:

- **Y**: Luminance component which represents the brightness information.
- **Cr** and **Cb**: Chrominance components which represent the color information.

As the human eye system is more sensitive to the Luminance than to the chrominance, the chrominance signals are both divided by a factor 2 in horizontal and vertical direction.

A macroblock consists of one block of 16 by 16 picture elements (luminance component) and two blocks of 8 by 8 picture elements (chrominance components).

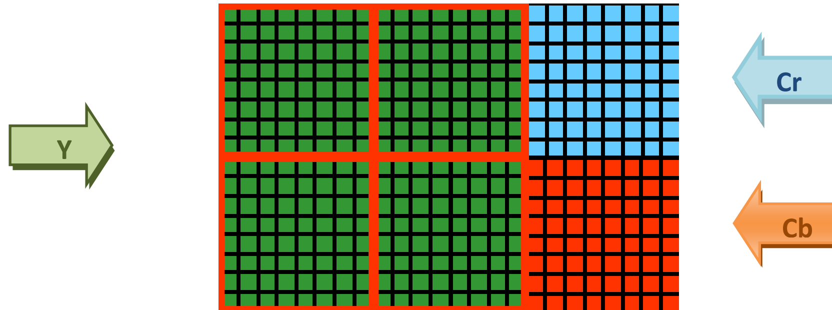


Figure 3.3.*Macroblock consisting of a Luminance component and Chrominance components.*

Each macroblock are coded in Inter or Intra mode:

- Inter mode: A macroblock is predicted using motion compensation (will be explained later).

- Intra mode: The image can be coded without reference to previously sent information (will be explained later).

The macroblocks are processed in so called slices. A slice is usually a group of macroblocks processed in raster scan order. There are five different slice-types:

- **I**: All macroblocks are encoded in Intra Mode.

- **P**: All macrobloks are predicted using a motion compensated prediction with one reference frame.

- **B**: All macrobloks are encoded using a motion compensated prediction with past and future pictures as reference.

- **SI** and **SP**: are used for an efficient switching between two different bit streams.

The following is the explanation of the operation of each of the blocks that appear in the diagram, to finally explain the operation step by step of the encoder.

Intra prediction: Predict a current block from previously coded blocks in the same frame.

For the prediction of the luminance component Y are in H.264/AVC two different types of intra prediction:

- INTRA_16X16: Only one prediction mode is applied for the whole macroblock. Four different prediction modes are supported for the type INTRA_16X16: Plane prediction, vertical prediction, horizontal prediction and DC prediction.

- INTRA_4X4: The macroblock, which is of 16x16, is divided into sixteen 4x4 subblocks and a prediction for each subblock of the luminance signal is applied individually. Nine different prediction modes are supported for the type INTRA_4X4. One is DC prediction mode and in addition eight prediction modes each for a specific prediction direction.

For the prediction of the chrominance signals Cb and Cr of a macroblock is similar to the INTRA_16x16 type for the luminance signals. It is performed always on 8x8 blocks using plane prediction, vertical prediction, horizontal prediction and DC prediction.

Motion compensation prediction: One method used by various video formats to reduce file size is motion compensation, that is a technique for describing a picture in terms of the transformation of a reference picture to the current picture. For many frames of a movie, the only difference between one frame and another is the result of either the camera moving or an object in the frame moving. In reference to a video file, this means much of the information that represents one frame will be the same as the information used in the next frame. Motion compensation takes advantage of this to provide a way to create frames of a movie from a reference frame.

For this purpose, each macroblock can be divided into smaller partitions. The use of larger blocks can reduce the number of bits needed to represent the motion vectors, while the use of smaller blocks can result in a smaller amount of prediction residual information to encode. Partitions with luminance block sizes of 16x16, 16x8, 8x16, 8x8 samples are supported.

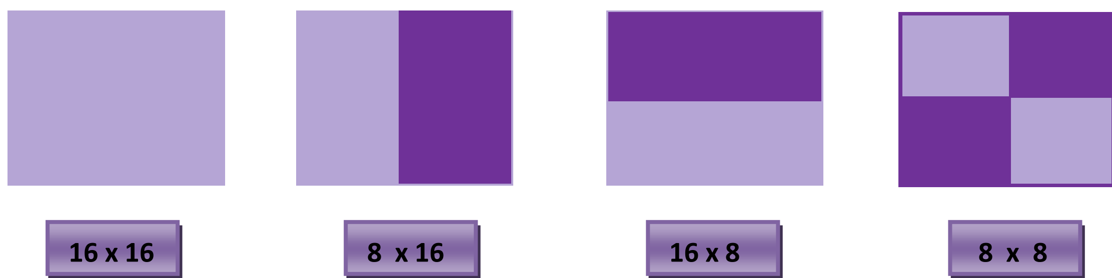


Figure 3.4.*Partitions with luminance block*

In case of an 8x8 sub-macroblock, one additional syntax element specifies if the corresponding 8x8 sub-macroblock is further divided into partitions with block sizes of 8x4, 4x8, 4x4.

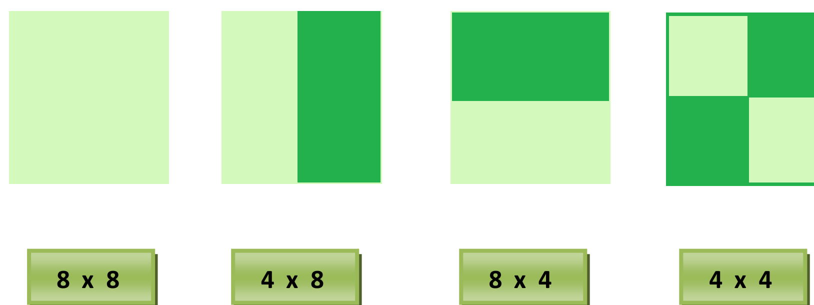


Figure 3.5.*Partitions of an 8x8 sub-macroblock*

A displacement vector (motion data) is estimated and transmitted for each block (motion estimation), refers to the corresponding position of its image signal in an already transmitted reference image. With this (motion data) and the reference image stored in memory, motion compensation calculates the prediction of the current image.

In former MPEG standards this reference image is the most recent preceding image. In H.264/AVC it is possible to refer to several preceding images. For this purpose, an additional picture reference parameter has to be transmitted together with the motion vector. This technique is called motion-compensated prediction with multiple reference frames.

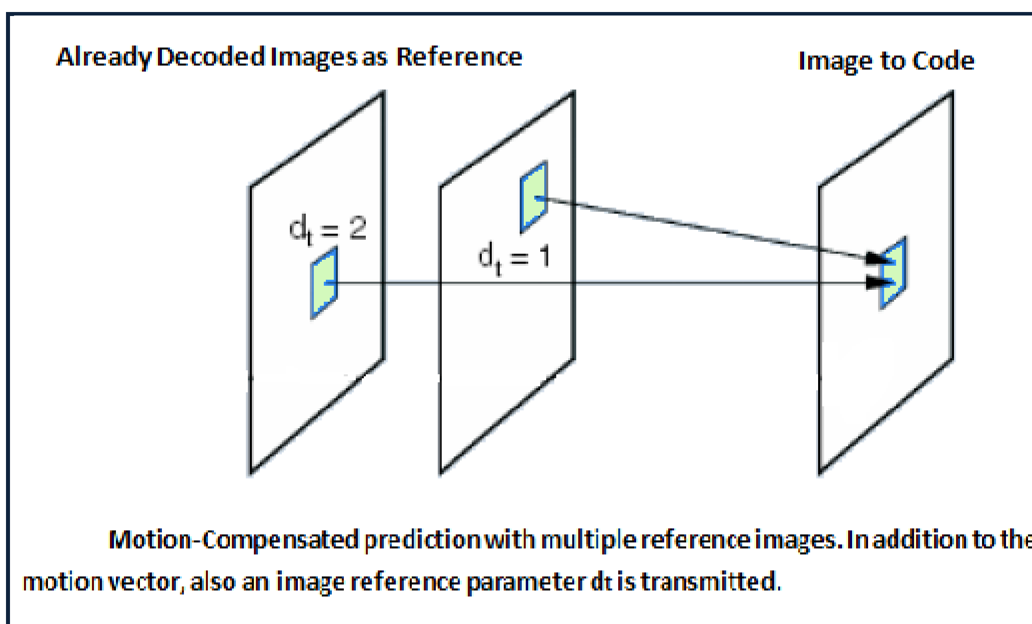


Figure 3.6.*Motion-compensated prediction with multiple reference frames*

H.264 encoding supports sub-pixel resolution for motion vectors, meaning that the reference block is actually calculated by interpolating inside a block of real pixels.

Hence, we employ sub-pixel grid, in which motion vectors may point to candidate blocks placed at half-pixel or sometimes quarter pixel locations. The reference block at sub-pixel grid can be generated using either bi-linear interpolation but in H.264 standard is used a more sophisticated six-tap filter.

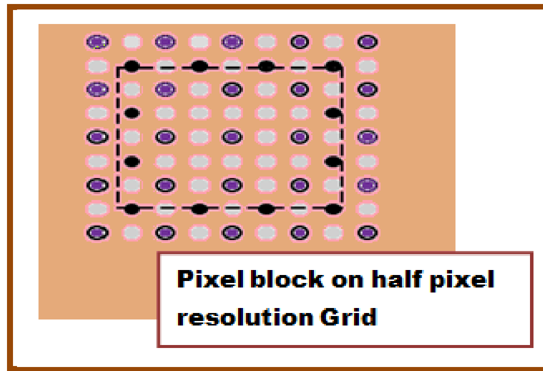


Figure 3.7.*Pixel block on half pixel resolution Grid*

Normally, the integer motion vector is further refined to first half pixel by testing the eight neighbouring half pixel locations and further to quarter pixel by testing the neighbouring eight quarter pixel locations.

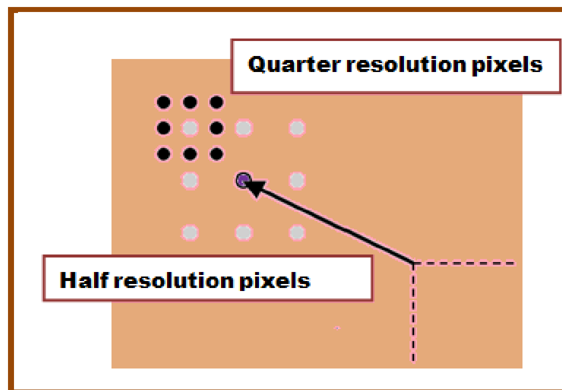


Figure 3.8.*Quarter resolution pixels*

In the application used in the project it is possible to point to candidate blocks placed only at half-pixel (of course also at full-pixel) and we will have the following cases:

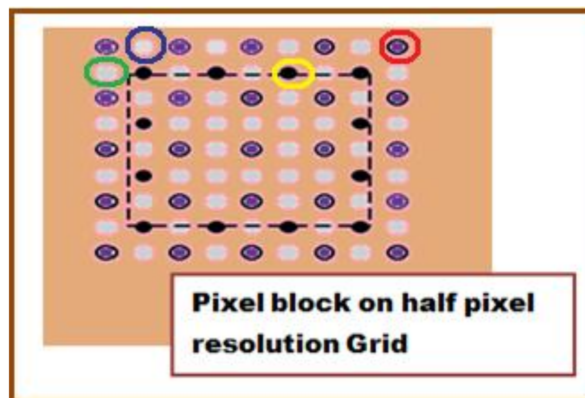


Figure 3.9.*Cases in the application of the project.*

- FH: Average vertical (green circle).
- HF: Average horizontal (blue circle).
- HH: Average vertical and average horizontal (yellow circle).
- FF: copy (red circle).

Transform Coding: Is applied in order to code the prediction error signal. The task of the transform is to reduce the spatial redundancy of the prediction error signal. For the purpose of transform coding, all former standards such as MPEG-1 and MPEG-2 applied a two dimensional Discrete Cosine Transform. In H.264/AVC, different integer transforms are applied instead of the DCT.

Quantification: The human eye is very good detecting small changes of brightness in relatively large areas, but not when the rapidly changing brightness in small areas (high-frequency variations), this allows elimination of high frequencies, without losing too much visual quality. All coefficients are quantized by a scalar quantizer. The quantization step size is chosen by a so called quantization parameter QP which supports 52 different quantization parameters. This is the process that loses much of the information (and quality) when an image is processed by this algorithm.

Entropy Coding Schemes: The entropic coding is a special form of compression without data loss. H.264/AVC specifies two alternative methods of entropy. These methods are used to codify coefficients and motion vectors. The encoding used for this task are based on VLC (Variable Length Coding) adaptative, of this concept is born the CAVLC (Context Adaptive Variable Length Coding) and CABAC (Context Adaptive Binary Arithmetic Coding).

Adaptative Deblocking Filter: In block-based video compression technology, blocking artifacts are obvious because of the luminance and chrominance discontinuities which are caused by block-based motion compensation. Filtering the block edges has been shown to be a powerful tool to reduce the visibility of these artifacts.

The filter described in the H.264/AVC standard is highly adaptive. Several parameters and thresholds and also the local characteristics of the picture itself control the strength of the filtering process. All involved thresholds are quantizer dependent, because blocking artifacts will always become more severe when quantization gets coarse.

The filter is adaptive on three levels:

- On Sample level: It is important to distinguish between true edges in the image and those created by the quantization of the transform-coefficients. True edges should be left unfiltered as much as possible.
- On Slice level: The global filtering strength can be adjusted to the individual characteristics of the video sequence.
- On block edge level: The global filtering strength is made dependent on intra/inter prediction decision, motion differences, and the presence of coded residuals in the two participating blocks. From these variables a filtering-strength parameter is calculated, which can take values from 0 to 4 causing modes from no filtering to very strong filtering of the involved block edge.

Memory: Where stored the macroblocks and can be used to predict future macroblocks.

Motion Estimation: Is the process of determining motion vectors that describe the transformation from one 2D image to another; usually from adjacent frames in a video sequence. (This process will be widely explained in the next point).

Once explained the operation of each of the blocks that appear in the diagram, the operation of the encoder is the following:

The input image is divided into macroblocks and each of these macroblocks is what we introduce into our encoder.

Each macroblock is introduced in the block of **motion estimation** where a displacement vector is estimated and transmitted for each macroblock (motion data) that refers to the corresponding position of its image signal in already transmitted reference image stored in memory. Vectors are introduced in the block of **motion compensation** that with this and the reference image stored in memory calculates the prediction of the current macroblock. The motion data calculated in motion estimation is also **entropy coded** and sent to the decoder.

The difference between the original and the predicted macroblock is called prediction error and is **transformed**, **quantized**, **entropy coded** and sent to the decoder. The prediction error has smaller energy than the original pixel values and can be coded with fewer bits, because of this is sent the prediction error instead of the original macroblock.

In order to reconstruct the same image on the decoder side, the quantized coefficients are **inverse transformed** and added to the prediction signal. Once this is done, the macroblock is introduced in the **deblocking filter** to reduce the visibility of blocking artifacts. The result is the reconstructed macroblock that is also available at the decoder side. This macroblock is stored in the **memory** where can be used to predict future macroblocks.

With respect to the block diagram H.264/AVC introduce the following changes:

- The Discrete Cosine Transform (DTC) used in former standards is replaced by an integer transform.
- An adaptive filter deblocking filter is used in order to reduce the block-artifacts.
- In H.264/AVC a prediction scheme is used also in Intra mode.
- H.264/AVC allows storing multiple video frames in the memory, whereas in previous standards the memory contains only one video frame.

3.1.3. - Video Decoder

A **decoder** is a device which does the reverse of an encoder, undoing the encoding so that the original information can be retrieved. The same method used to encode is usually just reversed in order to decode.

The following figure shows a generalized diagram of a video decoder with motion compensation:

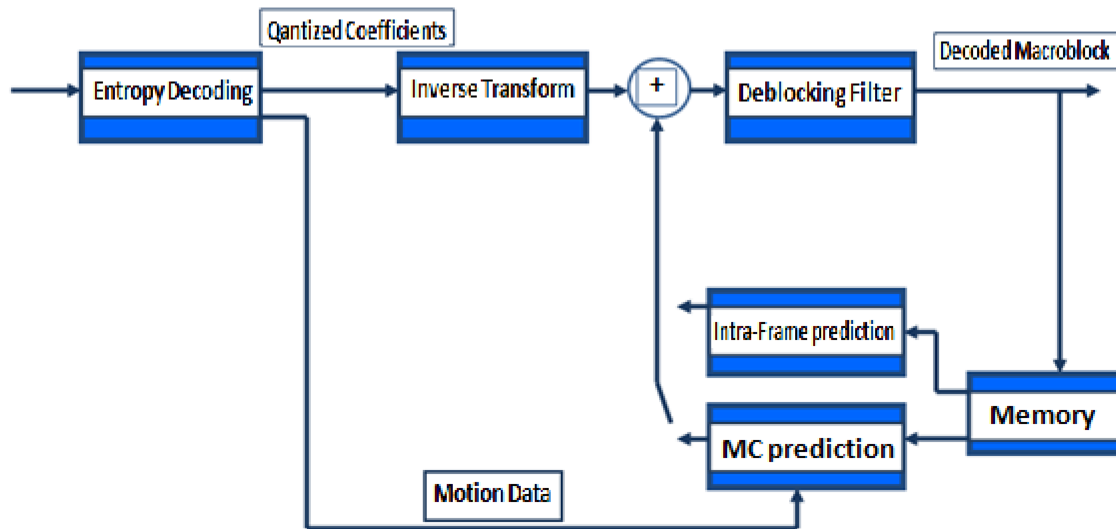


Figure 3.10. *Video decoder with motion compensation*

The decoder receives the motion data and the quantized coefficients, the **entropy decoder** decodes the quantized coefficients and the motion data, which is used for the **motion compensated prediction**.

As in the encoder, a prediction signal is obtained by **intra-frame** or motion compensation prediction.

The prediction signal obtained is added to the **inverse transformed coefficients**. Once this is done, the macroblock is introduced in the **deblocking filter** to reduce the visibility of blocking artifacts. After deblocking filtering, the macroblock is completely decoded and stored in the **memory** where can be used to predict future macroblocks.

3.2. - Motion Estimation

This point, presents the Motion Estimation algorithm with Three Dimensional Recursive Search (3DRS) as such algorithm. After that, an scheme of operation of motion estimation is presented, and finally, and example of motion estimation is explained.

3.2.1. - Introduction

Motion estimation is the process of determining motion vectors that describe the transformation from one image to another; usually from adjacent frames in a video sequence.

The basic assumption of motion estimation is that, consecutive frames of video will be similar except for changes induced by objects moving within the video frames.

Motion estimation extracts motion information from the sequence of video. The motion is represented by a motion vector (x,y). This vector indicates the displacement of a pixel or a pixel block from the current location due to motion. Applying the motion vector to an image to get the next image is called Motion compensation. The combination of motion compensation and estimation is a key part of video compression.

There are some techniques of motion estimation, as pel-recursive techniques, which generate motion vector for each pixel or phase plane correlation technique, which generate motion vectors via correlation between reference frame and current frame. But the most used is Block Matching Algorithm.

3.2.2. - Block Matching Algorithm

Block Matching Algorithm is the most used motion estimation algorithm. This algorithm (BMA) calculates a motion vector that is applicable to all the pixel in the block, i.e., calculates a motion vector for a block of pixels. Reducing thus the computational requirements and also results in a more accurate motion vector since the objects are typically a cluster of pixels.

Here are some figures to explain the functionality of the algorithm:

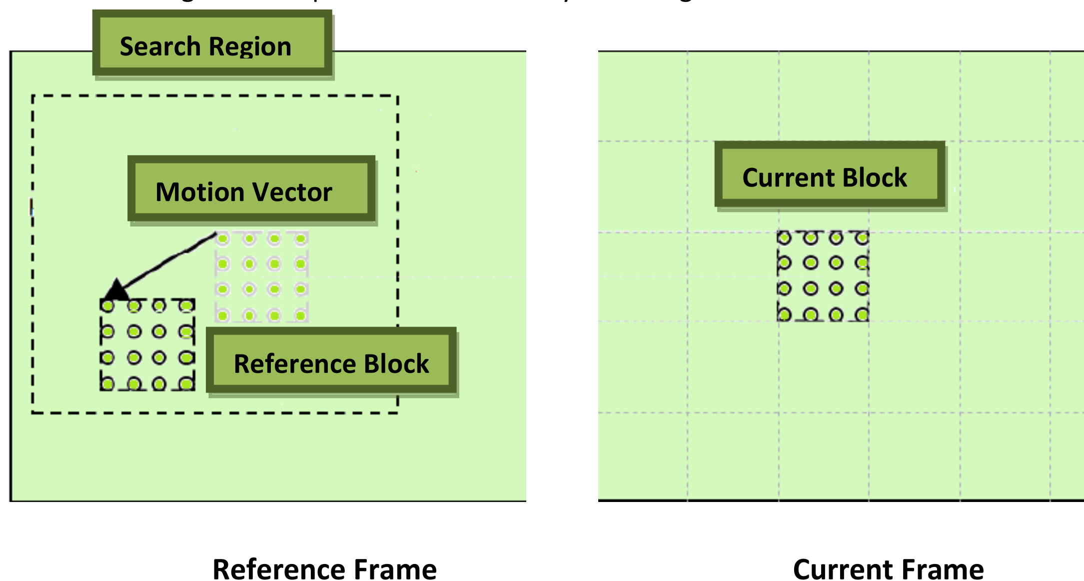


Figure 3.11.*Block matching algorithm*

The Current frame is divided into blocks of pixels and motion estimation is performed for each pixel block (as explained previously).

Motion estimation is performed by identifying a pixel block from the reference frame that best matches the block to be coded. The reference block is generated by displacement (this displacement is provided by the motion vector, which consists of a pair(x,y) of horizontal and vertical displacement values)from the location of the block to be coded in the frame of reference.

The reference pixel blocks are generated only from a region that defines the boundary for the motion vectors and limits the number of block to evaluate. This region is called search area. The height and width of search area is dependant on the motion in video sequence. Bigger search area require more computation due to increase in number of evaluate candidates. The available computer power determines the search area.

The choice of block size also determines the trade off between required computation and accuracy of motion vectors. Smaller block size can accurately describe motion of smaller objects but will need higher computation. A block size too small may not contain sufficient texture, therefore, it may not be possible to accurately predict block matching.

There are various criteria for calculating block matching. The most popular are listed below:

- Sum of Square Error (SSE) = $\sum_{x=1}^{x=N}\sum_{y=1}^{y=N}(C(x,y)-R(x,y))^2$

SSE provides a very accurate block matching, however requires many computations.

- Sum of Absolute Difference (SAD) = $\sum_{x=1}^{x=N}\sum_{x=1}^{y=N}|C(x,y)-R(x,y)|$

SAD provides fairly good match at lower computational requirement. Hence it is widely used for block matching.

We have assumed that the block can be found by an integer number of pixels. In practice, the displacement of an object between two subsequent frames in a video is not an integer number of pixels. Hence, modern coding standards employ also sub-pixel grid, in which motion vectors may point to candidate blocks placed at half-pixel or sometimes quarter pixel locations. The reference block at sub-pixel grid is generated using either bi-linear interpolation or a more sophisticated six-tap filter as used in H.264 standard. Figure 3.7 shows a pixel block on a half pixel resolution grid.

Normally, the integer motion vector is further refined to first half pixel by testing the eight neighbouring half pixel locations and further to quarter pixel by testing the neighbouring eight quarter pixel locations. Figure 3.8 presents a quarter resolution pixels.

In the application used in the project it is possible to point to candidate blocks placed only at half-pixel and we will have the following cases:

- FH: Average vertical (green circle).
- HF: Average horizontal (blue circle).
- HH: Average vertical and average horizontal (yellow circle).
- FF: copy (red circle).

These cases are presented in Figure 3.9.

To find the pixel block from the reference frame that best matches the current block, we must do a search of candidates. This search can be made with several algorithm, we are going to explain Three Dimensional Recursive Search (3DRS), since it will be used in the application.

3.2.3. - Three Dimensional Recursive Search

Three dimensional recursive search (3DRS) is one such algorithm. The algorithm has two assumptions:

- Objects are larger than a block size
- Objects have inertia

Taking into account the first assumption the neighbouring blocks motion vectors can be used as candidates for the current block. Here could appear a problem as for neighbouring blocks ahead in raster scan, there is no motion vectors calculated yet. Here is where we apply the second premise and motion vectors from previous frame are for this blocks. In my application, for the search of candidates (motion vectors candidates) through 3DRS, the algorithm that will be used is the explained below.

Bearing in mind that x is the coordinate x of the position of the macroblock to be coded and y is the coordinate y of the position of the macroblock to be coded, the candidates will be sought as follow:

Candidate1: MV(x-1, y-1)

Candidate2: MV(x+1, y+1)

Candidate3: MV(x+1, y-1)

Candidate4: MV(x+0, y-1)

Candidate5: MV(x-1, y-1) + (Random_x, Random_y)

Candidate6: MV(x+1, y+1) + (Random_x, Random_y)

Candidate7: MV(x+1, y-1) + (Random_x, Random_y)

Candidate8: MV(x+0, y-1) + (Random_x, Random_y)

NOTE: From the fifth candidate, the motion vector will be increased by a random to calculate the motion vector of that block.

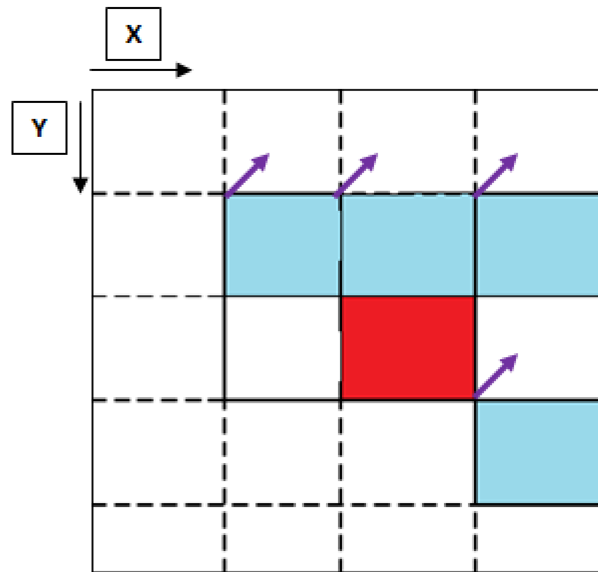


Figure 3.12.*Motion Vector candidates without random*

In Figure 3.12 is marked in red color the macroblock to be coded and in blue color the macroblocks whose motion vectors (without random) can be used as candidates for the current macroblock (found by 3DRS).

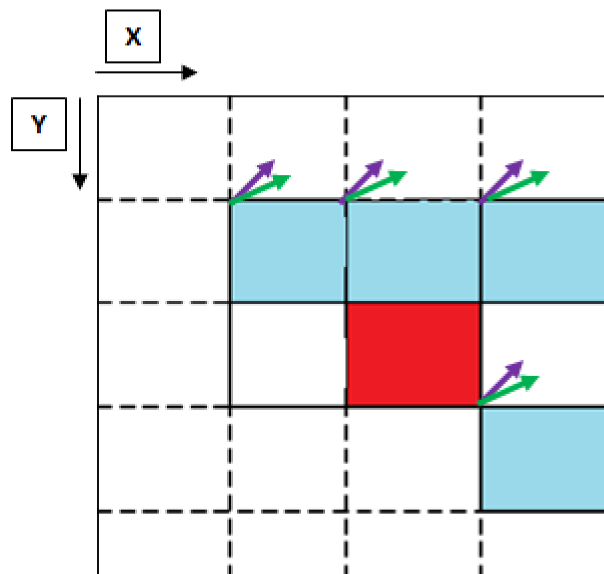


Figure 3.13.*Motion Vector candidates without and with random*

In Figure 3.13, the macroblock to be coded is marked in red color and in blue color the macroblocks, whose motion vectors without and with a later random, can be used as candidates for the current macroblock (found by 3DRS).

If any of the blocks chosen by 3DRS is outside of the image, the motion vector that takes will be:

- Exit at the top: the motion vector of the down block.
- Exit at the bottom: the motion vector of the top block.
- Exit on the side: the motion vector of the next block.

Example:

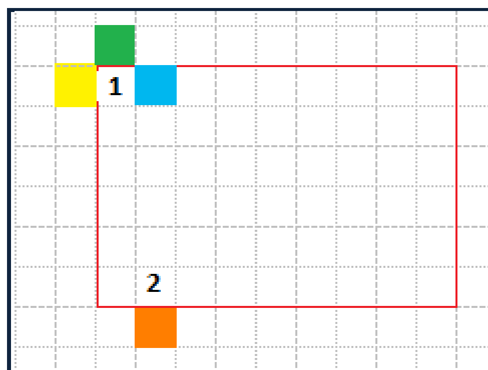


Figure 3.14.*Blocks chosen by 3DRS are outside of the image*

- The motion vector of the green block will be the motion vector of the block number 1.
- The motion vector of the yellow block will be the motion vector of the block number 1.
- The motion vector of the orange block will be the motion vector of the block number 2.

When there are two equals candidates, two motions vectors equal, we use only once this motion vector within the motion compensation, therefore we have a candidate less.

3.2.4. - Scheme of operation of motion estimation

Figure 3.15 shows the scheme of operation of motion estimation:

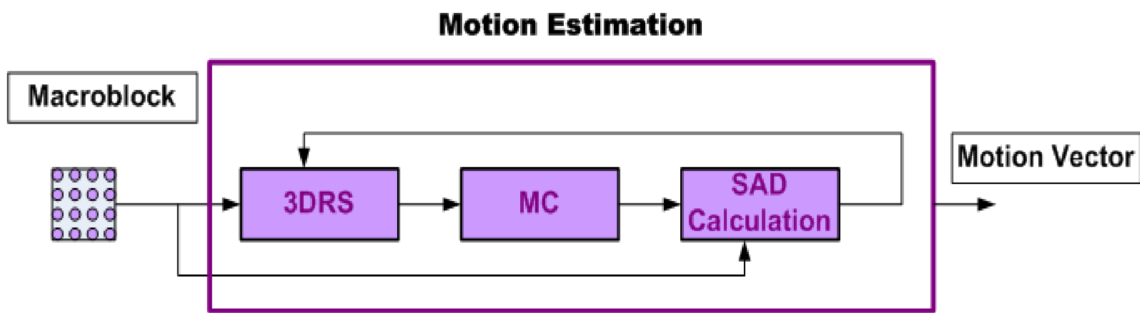


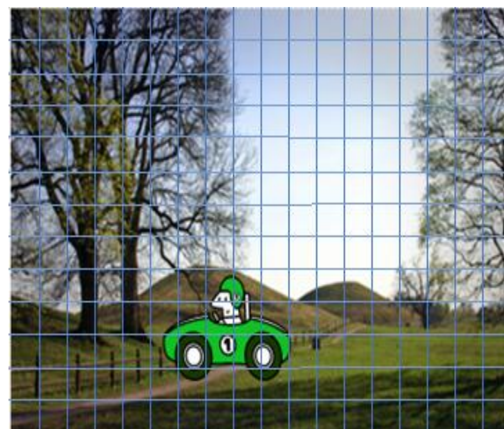
Figure 3.15.*Scheme of operation of motion estimation*

The steps will be as follow:

6. The macroblock (to be coded) of the current image is introduced.
7. It is estimated the first motion vector in the current image through 3DRS.
8. With the motion vector of the block calculated and the reference image, through motion compensation, is locate the candidate in the reference image.
9. Through the subtraction of the macroblock of the current image and the candidate located in the reference image, is obtained the SAD.
10. Repeat steps 2 through 4 for 8 candidates, to finally stay with the motion vector of the candidate with the lowest SAD.

3.2.5. - Example of motion estimation

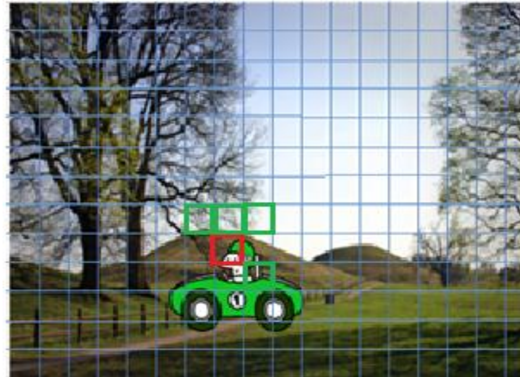
Figure 3.16 shows an example of a frame with a landscape and a car. The second half of this figure is an example of a possible next frame, where the car has moved towards the tree.



Frame 1 **Frame 2**

Figure 3.16.*Frame and next frame of an image*

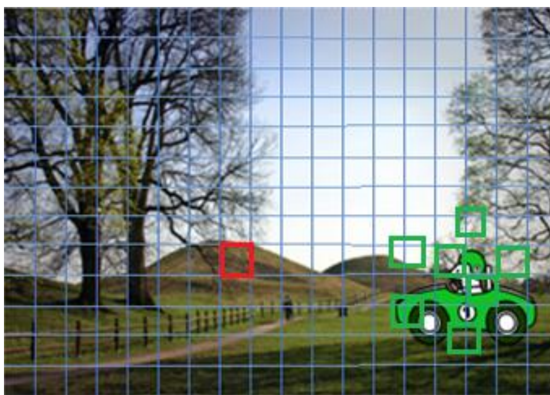
Then in the frame 2 is marked in red color the macroblock to be coded and in green color the macroblocks (found by 3DRS) whose motion vectors (4 without random and 4 with random) can be used as candidates for the current macroblock (for macroblocks ahead in raster scan, there is no motion vectors calculated yet, then assuming that the objects have inertia, the motion vector for previous frame are for these macroblocks).

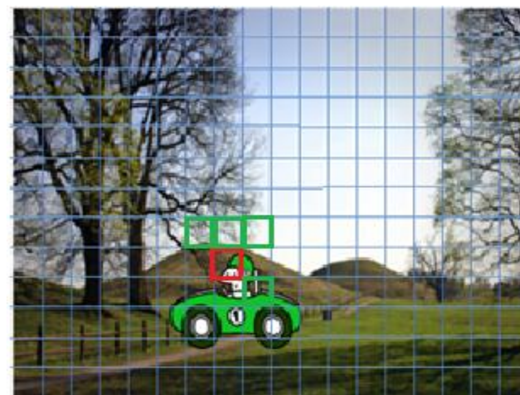


Frame 2

Figure 3.17.*In red color is marked the macroblock to be coded and in green color the macroblocks whose motion vectors (without and with a random) can be used as candidates for the current macroblock.*

Then place the same macroblock of the current picture in the image of reference and with the motion vectors calculated, the candidates (with motion compensation) are obtained.





Frame 1 **Frame 2**

Figure 3.18.*In frame 1 the candidates are obtained*

Once this is done, from SAD calculated for each candidate, we rule out candidates with the highest SAD, we stay therefore with the motion vector of the candidate with the lowest SAD.

The selected candidate will be the following:

Frame 1

Figure 3.19.*Selected candidate*

Chapter 4

Proposed enhanced DMA unit

This section introduces the DMA processor interface that will be used to introduce the improvements in the project. After that, the DMA proposals for the **'Enhanced DMA Unit'** will be presented.

4.1. - Introduction

Nowadays because of the wide variety of applications such as current video applications, multimedia cell phones, high-definition interactive television… are necessary continuous improvements requiring sophisticated multimedia operations, which imply a high demand of processing power.

Because of this and to achieve better performance, the processors tend to behave less linearly and more in parallel. Due to the capability of exploiting parallelism in multimedia applications, VLIW architecture is being very used in the last years.

Using subword parallelism a VLIW architecture can achieve a significant improvement in performs. In subword parallelism, we pack multiple subwords into a word and then process the whole words.

Therefore data-parallelism needs a way to expand data into larger containers for more precision in intermediate computations and data alignment before or after a certain operations for subwords.

Most SIMD processors provide access to only contiguous data in memory, with strong alignment restrictions because they have limited memory architecture. These architectures, either do not provide any hardware support for unaligned accessed or provide it but with degrade results. Therefore the programmer usually ends doing the alignment in software which, implies an extra-overhead not having an efficient support for unaligned accesses and that degrade the performance significantly.

For example there are instructions like MIX, permute, or Load+ Alignment but these instructions subject the CPU to a heavy overhead.

In applications like video coding and decoding that use ME and MC algorithms, it is not possible to avoid unpredictable unaligned memory references. This is because to identify a pixel block for the reference frame that best matches the block to be coded, the reference block is generated by displacement from the location of the block to be coded in the frame of

reference. This displacement is performed using the motion vector and can be arbitrary and therefore, that results in a lot of unpredictable unaligned accesses.

In this type of applications, current software optimizations become unsuccessful because the instructions that are necessary for doing the data realignment in software subject the CPU to a heavy overhead.

Knowing that **DMA** is an essential feature of all modern computers, as it allows devices to transfer data without subjecting the CPU to a heavy overhead, a solution would be to create a **'Enhanced DMA Unit'** capable of performing the alignment of data without subjecting the CPU to a heavy overhead.

The instructions mentioned above will remain used, because once the data is in internal memory, these instructions will be responsible for performing any alignment required. But in the other cases will use an **'Enhanced DMA Unit'** because thus will be removing code and therefore saving processing cycles.

The idea would be that this **'Enhanced DMA Unit'** in addition to the alignment of data can make padding of data.

For example in applications like video coding and decoding that use ME and MC algorithms, to search the block of the reference frame that best matches the block to be coded, the search is performed only in a region known as the search area, then when a candidate (to be a pixel block from the reference that best matches the block to be coded) goes beyond this search area, it is necessary to make padding of data.

Perform these extensions with DMA Unit also would reduce considerably the overhead of the CPU. Moreover it would be also interesting that this **'Enhanced DMA Unit'** was able to store several transmission in a queue, allowing to perform one transfer after the other without needing to wait in order to configure the transfers. Thus also would reduce considerably the overhead of the CPU.

4.2. - DMA processor interface

This section presents the characteristics of the DMA Unit that will be used to introduce the improvements.

4.2.1. - Processor interface

The DMA Unit used is a 2D-1D that conducted the exchange of data between an external memory (SDRAM) and an internal memory (mem).

DMA transfer between external memory and local data and instruction memory is performed through the 64bit DMA interface via the BUS AHB bus.

In a register-based DMA, the processor directly programs the DMA control registers to initiate a transfer. The DMA use the followings register:

DMA Registers	
DMA_TRANSFER_CTRL	With this register we will indicate if we want in SDRAM to read or write (8 bit). 1: Read to SDRAM; 0: Write to SDRAM
DMA_EXT_BASE_ADDR	External base address (32 bit)
DMA_SEGMENT_LENGTH	Segment length (64 bit words; 12 bit)
DMA_GAP_LENGHT	Offset between segments (64 bit words;12 bit)
DMA_SEGMENTS	Number of segments (6 bit)
DMA_MEM_ADDR	Internal base address (12 bit)
DMA_DATA_CTRL_START	DMA data transfer start (1bit)

Table 4.1.*DMA Registers*

DMA transfers are initiated by writing to DMA_DATA_CTRL_START a '1' for data transfers. The transfer direction is configured by writing '1' or '0' in the bit 0 of DMA_TRANSFER_CTRL ('1': read from extern; '0': write from extern). A busy transfer is indicated by flags. The flag is called dma_data_status, the flag is '1' while the transfer is being made and when the transfer ends the flag is set to '0'. DMA also will be working internally with two pointers one for the external memory (pointer_to_sdram (size configurable)) and the other one for the internal memory (pointer_to_mem (size configurable)).

4.2.2. - Example

To better understand the meaning of these registers, I will present an example. The configuration of the registers will be as follows:

- DMA_TRANSFER_CTRL: 0x001
- DMA_EXT_BASE_ADDR: 0x060 **NOTE*
- DMA_SEGMENT_LENGTH: 0x003
- DMA_GAP_LENGHT: 0x00a
- DMA_SEGMENTS: 0x002 (There are 3 segments because there are segment 0)
- DMA_MEM_ADDR: 0x000
- DMA_DATA_CTRL_START: 0x001

__NOTE:__ when our pointer takes the direction of SDRAM (DMA_EXT_BASE_ADDR) is not going to take every bit, it will leave the first three bits without take (to be explained in later), that is, it will take the bits from 3 onwards. Therefore the 0x060 (96 decimal) will be 1100000 in binary, if remove the first three bits we stayed with 1100 that is 0x00c in hex. So the base direction of our SDRAM will be 0x00c.

SDRAM contents will be as follows:

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09
0x0a	0x0b	0x0c	0x0d	0x0e	0x0f	0x10	0x11	0x12	0x13
0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d
0x1e	0x1f	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f	0x30	0x31
0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39	0x3a	0x3b

Figure 4.1._SDRAM contents_

The data are taking from the direction 0x0c and will have to take 3 rows of data and the length of the rows will be 3, the offset between segments will be 10.

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	GAP
0x0a	0x0b	**0x0c**	**0x0d**	**0x0e**	0x0f	0x10	0x11	0x12	0x13	
0x14	0x15	**0x16**	**0x17**	**0x18**	0x19	0x1a	0x1b	0x1c	0x1d	
0x1e	0x1f	**0x20**	**0x21**	**0x22**	0x23	0x24	0x25	0x26	0x27	
0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f	0x30	0x31	
0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39	0x3a	0x3b	

Figure 4.2._GAP_

MEM contents will be as follows:

```
0X000  ->  0X0c
0X001  ->  0X0d
0X002  ->  0X0e
0X003  ->  0X16
0X004  ->  0X17
0X005  ->  0X18
0X006  ->  0X20
0X007  ->  0X21
0X008  ->  0X22
```

Figure 4.3.*Mem contents*

4.3. - Alignment process

This section studies the alignment process and presents a DMA capable of performing the alignment process.

4.3.1. - Introduction

A memory reference is called aligned when it accesses positions that match with the memory access granularity of the processor, when that does not happen is called misaligned (or unaligned).

Most SIMD processors have a limited architecture and cannot access an unaligned memory position and if they can, is with a big performance penalty. To access an unaligned position, must be done a realignment process. The realignment process is shown in Figure 1.3.

That process consists in:

- To read the aligned memory word that is located before the unaligned position and discards the unnecessary bytes.
- To read the aligned memory word that is located after the unaligned position and discards the unnecessary bytes.
- To merge the necessary bytes of the two previous word.

In current SIMD extensions the level of support for unaligned accesses includes variations from hardware mechanisms, instructions to do the re-alignment in software and system exceptions. The only one that includes both hardware support and unaligned exceptions is the Intel's SSE extension. The initial design only provides support for aligned accesses with the instruction MOVDQA (Move Aligned Double Quadword). The SSE2 extension with the instruction MOVDQU (Move Unaligned Double Quadword) that was implemented using two 64-bit loads (or stores) and was based on microcode, allows to load and to store non-aligned 128 bit words. But the SSE2 extension results in big latencies and big performance penalties for unsigned accesses. Then to solve the mentioned problems the LDDQU (Load Unaligned Integer 128 bits) instruction was introduced in SSE3 extension. This instruction performs a 32 byte load and then performs a shift to extract the corresponding 16 bytes of unaligned data.

This kind of alignment of the data described above could be called **'Load + Alignment'** and subjects the CPU to a heavy overhead because it is necessary to make the load data in two different registers, then the operation of alignment and finally another load operation in the final register.

In addition to the manner described to perform the alignment, there are two more instructions that allow us to make the alignment of data:

- **MIX:** These instructions take subwords from two registers and interleave alternate subwords from each register in the result register. Mix left starts from the leftmost subword in each of the two source registers, while mix right ends with the rightmost subwords from each source register. Figure 1.1 illustrates this for 16-bit subwords [1].

- **Permute:** The permute instruction takes one source register and produces a permutation of that register's subwords. With 16-bit subwords, this instruction can generate all possible permutations, with and without repetitions, of the four subwords in the source register. Figure 1.2 shows a possible permutation. To specify a particular permutation, we use a permute index. The instruction numbers subwords in the source register starting with zero for the leftmost subword. A permute index identifies which subword in the source register the instruction places in each subword of the destination register [1].

The problem with all these instructions is that subject the CPU to a heavy overhead because they perform data realignment in software. That is a problem for applications like video coding and decoding that use ME and MC algorithms, and they need continuous access to unaligned memory references (to identify a pixel block for the reference frame that best matches the block to be coded, the reference block is generated by displacement from the location of the block to be coded in the frame of reference. This displacement is performed using the motion vector and can be arbitrary and that results in a lot of unpredictable unaligned accesses).

Therefore the proposal is to create a DMA capable of performing the alignment of data.

4.3.2. - DMA with alignment

To make the alignment of the data on the DMA Unit, the DMA Unit will be implemented as follows:

The directions point to a data of 64-bit. Due to there are subword parallelism, this information is in turn divided into several smaller data (8-bit), particularly in 8 data. That's, the directions are pointing to a words that are composed of 8 subwords. Figure 4.4 illustrates it:

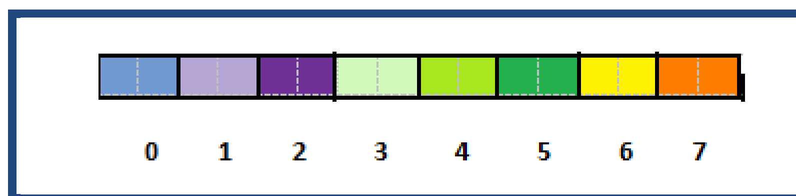


Figure 4.4.*Words composed of 8 subwords*

Through the alignment is possible to point to any subword, and from that subword take a whole word. Figure 4.5 shows an example of alignment:

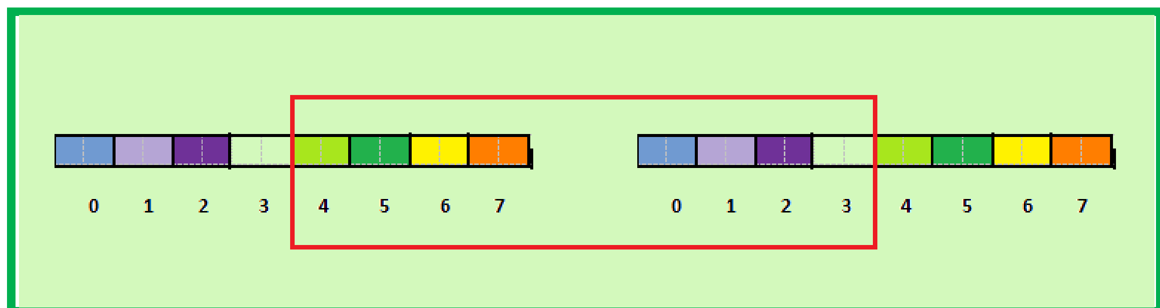
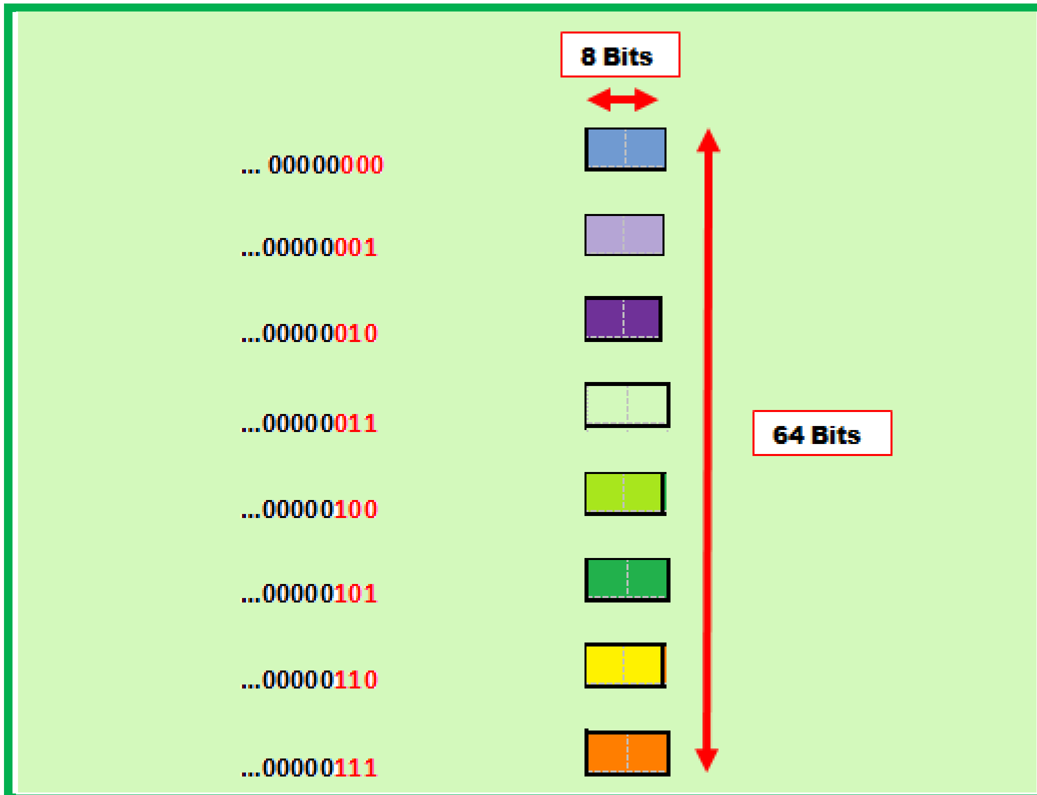


Figure 4.5.*Example of alignment*

The idea will be that of the direction that will point the data of 64 bits, the first three bits will be used to perform the alignment, that is, with these three bits will be indicated whether the word is aligned, or if is necessary to take the 64-bit word from a determined subword. Figure 4.6 illustrates that:



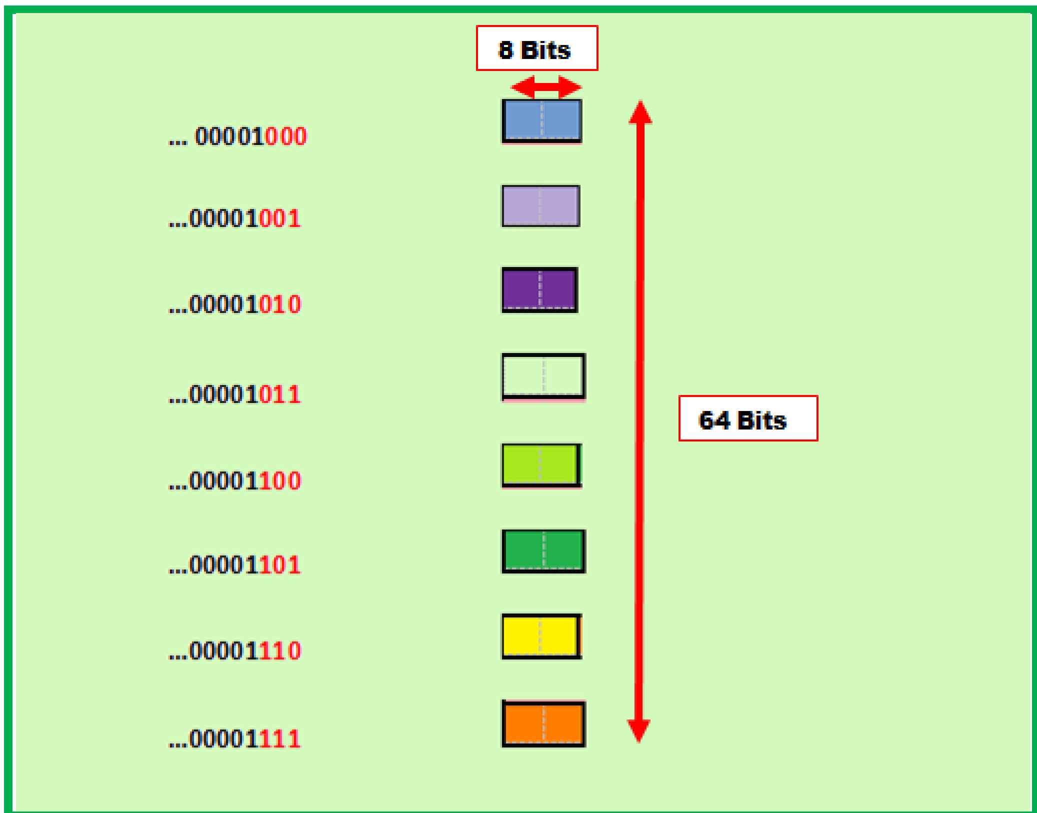


Figure 4.6.*The first three bits of the direction will be used to perform the alignment*

As can be seen, the numbers of black color remain the same for the 64 bits of the aligned word, and the first three bits (marked in red) indicates the subword of the word to be pointing. For example if the address is ...00000101, from that subword will have to take 64-bit:

Figure 4.7.*Example of alignment*

To implement this, is necessary to create an internal register of three bits in the DMA Unit in which will be stored the last three bits of the address at which wants to point.

```
bit[2:0]                    int_row_reg

int_row_reg = dma_ext_base_addr_reg[2:0]
```

With the pointer that pointing to the SDRAM I will always point to addresses of aligned data, therefore this pointer should not take the first three bits of the direction that I have:

```
pointer_to_sdram    = dma_ext_base_addr_reg[SDRAM_W-1+3:0+3]

SDRAM_W              = constant(configurable)
```

Figure 4.8 shows how the pointer always point to addresses of aligned data:

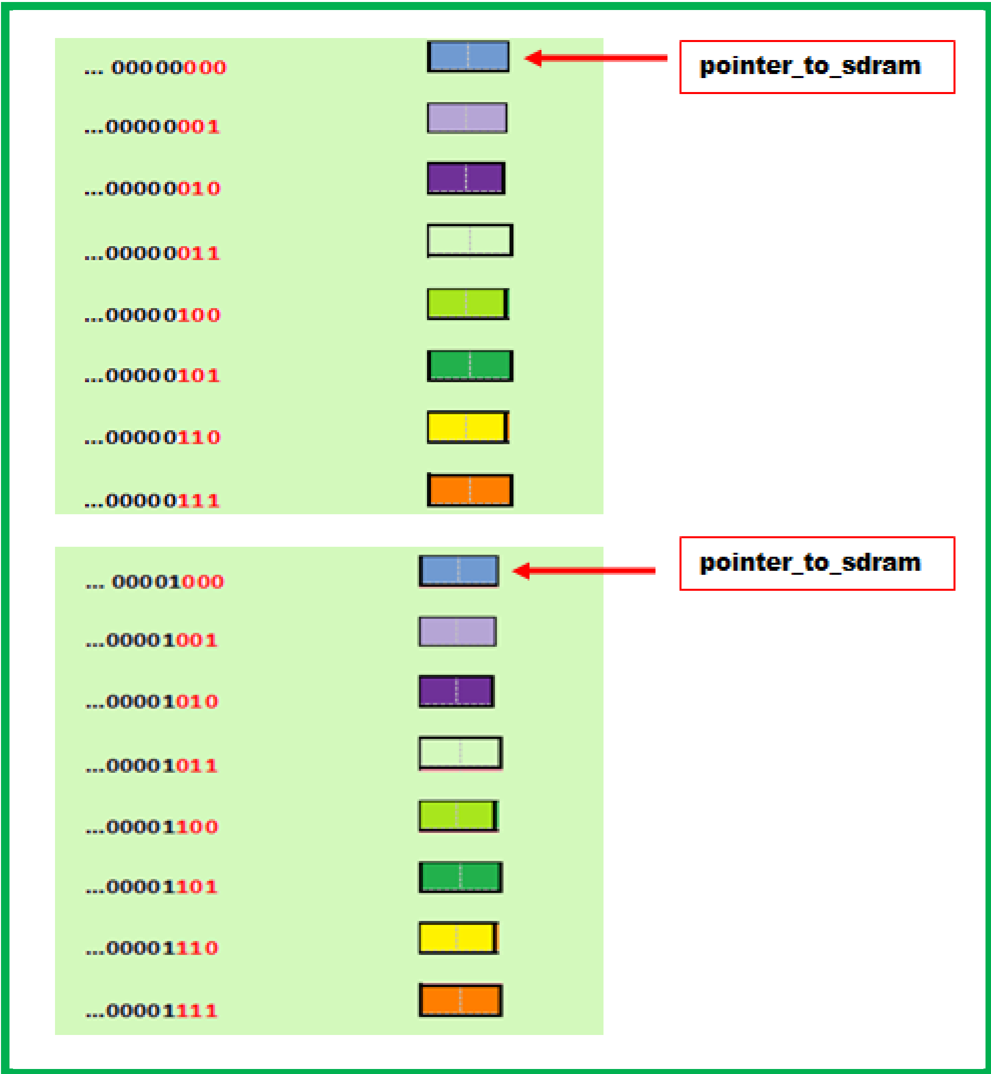


Figure 4.8.*Pointer always point to addresses of aligned data*

With the first three bits of data, as explained above, will be indicated the subword within the word that wants to point.

When these three bits are 0 mean that is not necessary to do alignment of data, because the word of data that wants to take is aligned. With any combination of these three bits that isn´t 000, will be necessary to perform alignment of data.

For that will be created three internal registers in the DMA:

```
bit[63:0]     int_aux1_reg

bit[63:0]     int_aux2_reg

bit[63:0]     int_final_reg
```

Then when int_row_reg has the three bits to zero, will not have to perform alignment of data and the implemented code will be the follow:

```
moai4k_mem.mem [pointer_to_mem]     = moai4k_sdram.sdram [pointer_to_sdram]
```

Otherwise, we will have to make alignment of data and the implemented code will be the follow:

```
int_aux1_reg              =     moai4k_sdram.sdram[pointer_to_sdram];

int_aux2_reg              =     moai4k_sdram.sdram[pointer_to_sdram +1];

int_final_reg[63:8*int_row_reg]     =     int_aux1_reg[((8-int_row_reg)*8)-1:0];

int_final_reg[(8* int_row_reg)-1:0]=     int_aux2_reg[63:(8- int_row_reg)*8];

moai4k_mem.mem [pointer_to_mem]     =     int_final_reg;
```

As can be seen, in the case of non alignment, the data indicated by the pointer of the SDRAM is copied directly into the address indicated by the pointer of the internal memory.

In the case of alignment, first copy the aligned memory Word that is located before the unaligned position in an internal register (int_aux1_reg), second copy the aligned Word that is located after the unaligned position in other internal register (int_aux2_reg). Then in other internal register (int_final_reg) will be stored the final results, that's, the necessary data of the two words that had been stored in the registers.

The content of this register is what will be stored into the address indicated by the pointer of the internal memory. Figure 4.9 shows an example:

In int_row_reg there is 2 in decimal (int_row_reg = 010) hence it is necessary to perform alignment. The operation done is the follow:

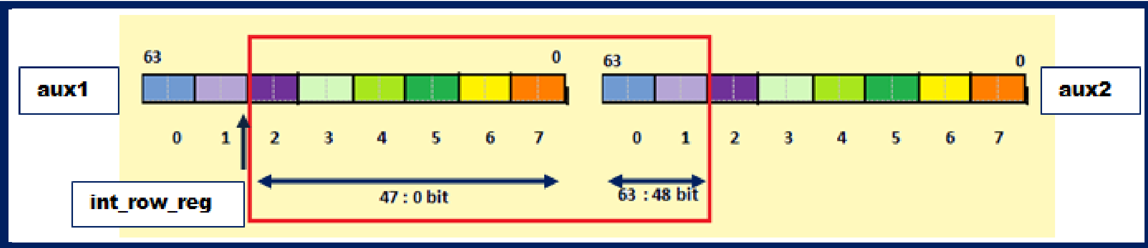


Figure 4.9.*Example of alignment using int_row_reg*

Of int_aux1_reg selects the bits from 0 to 47, since it would be:

```
int_aux1_reg[((8-2)*8)-1:0] = int_aux1_reg[47:0]
```

And of int_aux2_reg selects the bits from 48 to 63, since it would be:

```
int_aux2_reg[63:(8-2)*8] = int_aux2_reg[63:48]
```

The result is stored in the final register:

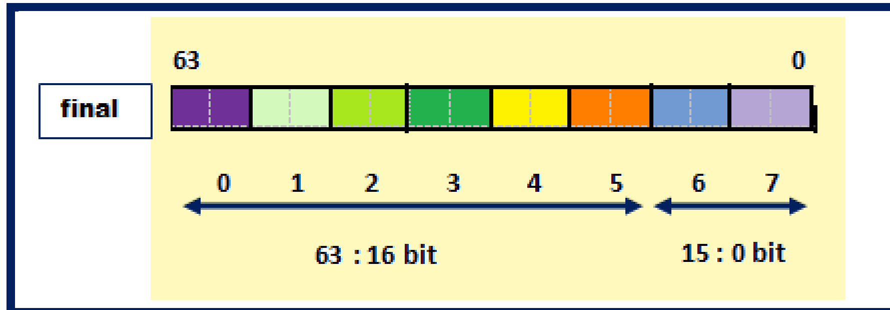


Figure 4.10.*Final register*

The final register stores the bits from 0 to 47 of int_aux1_reg, in the bits from 16 to 63 of int_final_reg, since it would be:

```
int_final_reg[63:8*2]   =   int_aux1_reg[((8-2)*8)-1:0]

int_final_reg[63:16]    =   int_aux1_reg[47:0]
```

The final register stores the bits from 48 to 63 of int_aux2_reg, in the bits from 0 to 15 of int_final_reg, since it would be:

```
int_final_reg[(8*2)-1:0]     =   int_aux2_reg[63:(8-2)*8]

int_final_reg[15:0]          =   int_aux2_reg[63: 48]
```

Performing the alignment in the DMA, much better results will be obtained because the CPU is not subject to a heavy overhead.

With the instructions described above, it was necessary to load the data in memory, later perform the alignment in software and finally load the aligned data into another register. Therefore three registers are being used and the CPU is performing work.

Performing the alignment in the DMA, the alignment is performed while loading the data, and the data are loaded into memory already aligned. In this way, all the work of alignment is being performed in the DMA without interrupting the CPU and also is being used only a register of memory (where the results will be stored in the desired order).

In applications like video coding and decoding that use ME and MC algorithms, this mode to perform the alignment of data, can improve significantly the performance.

Using ME and MC algorithms it is not possible to avoid unpredictable unaligned memory references. This is because to identify a pixel block for the reference frame that best matches the block to be coded, the reference block is generated by displacement from the location of the block to be coded in the frame of reference. This displacement is performed using the motion vector and can be arbitrary, therefore that results in a lot of unpredictable unaligned accesses. In point 3.2.5 there is an example of motion estimation.

Whenever that is necessary to situate a candidate in the reference image, the operation described above will be performed, and that results in a lot of unaligned accesses.

Performing the alignment of data in the DMA in this type of applications, a lot of work to the CPU is being saved and besides if the CPU doesn`t have to do this work, CPU can perform other tasks while the DMA makes the alignment and therefore the performance will be much better.

4.4. - Padding process

This section introduces the concept of padding and presents the proposal of a DMA capable of performing the padding process.

4.4.1. - Introduction

In applications like video coding and decoding that use ME and MC algorithms, when we want to search the block of the reference frame that best matches the block to be coded, we search only in a region known as the search area, then when a candidate (to be a pixel block from the reference that best matches the block to be coded) goes beyond this search area, it is necessary to make padding of data.

Figure 4.11 clarifies the need to use padding in video applications:

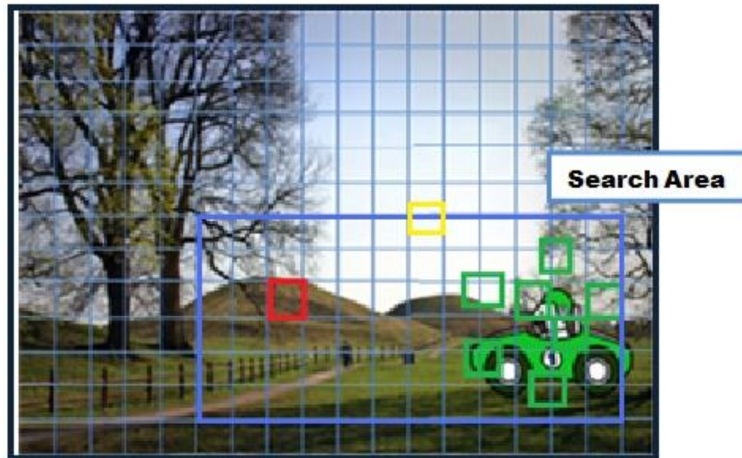


Figure 4.11.*Frame with the need to use padding*

In Figure 4.11, it can be seen outside of the search area a part of one candidate (the yellow block), which means that for draw this candidate, is necessary to make a padding of data. In this case, the padding of data is carried out by the top, that is, the contents of the last line of data that is within the search area, must be extended upwards.

There are eight different cases where there is a need for padding as can be seen in Figure 4.12:

Figure 4.12.*Cases that need to do padding*

- Horizontal padding right (brown block).
- Horizontal padding left (gray block).
- Vertical padding upwards (yellow block).
- Vertical padding downwards (orange block).
- Vertical padding upwards and horizontal padding right (blue block).
- Vertical padding upwards and horizontal padding left (green block).

- Vertical padding downwards and horizontal padding right (purple block).
- Vertical padding downwards and horizontal padding left (pink block).

The idea would be that the **'Enhanced DMA Unit'** in addition to the alignment of data can make padding of these, because would be reduced considerably the overhead of the CPU.

4.4.2. - DMA with padding

To make the padding of the data on the DMA Unit, it would be implemented as follow:

It is necessary to implement two new registers in the DMA Unit:

- **X_EDGE**: To indicate the number of words those are outside of the image. If it is positive means that the words are outside of the image to the left side, if it is negative means that the words are outside of the image to the right side.
- **Y_EDGE**: To indicate the number of segments those are beyond of the image. If it is positive means that the segments are outside of the image for the top, if it is negative means that the segments are outside of the image from the bottom.

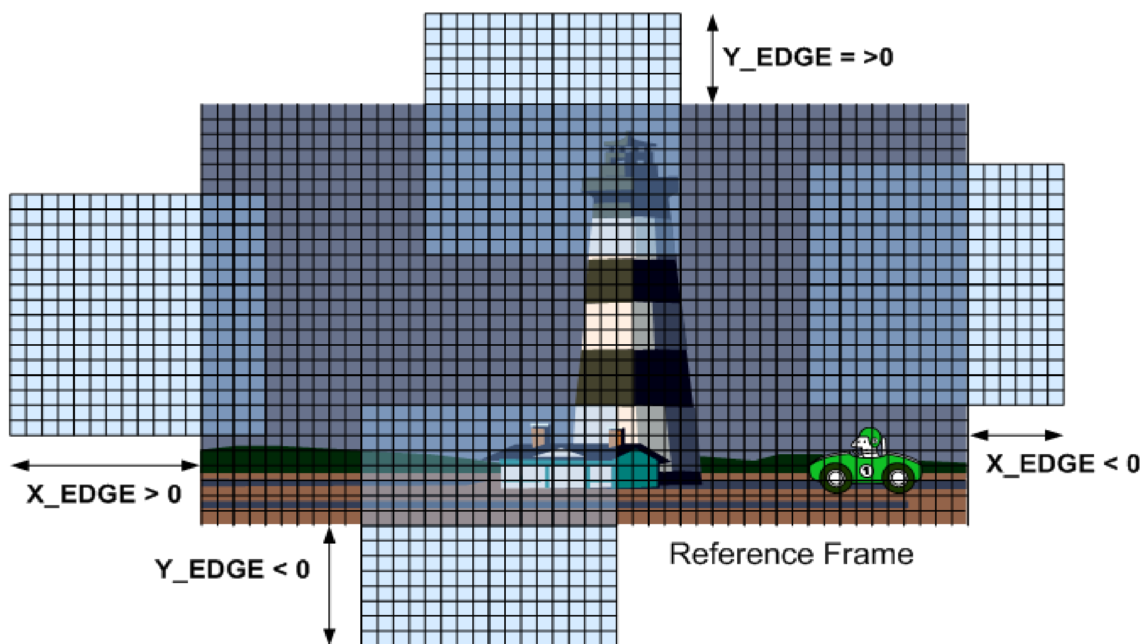


Figure 4.13.*DMA registers*

There are eight different cases where there is a need for padding, which will be explained in three blocks.

<u>CASE1:</u> The block is completely within the image, the block is outside of the image for the top or the block is outside of the image from the bottom.

Without alignment:

```
moai4k_mem.mem[pointer_to_mem]  =  moai4k_sdram.sdram  [pointer_to_sdram  +
(int_gap_length_reg* int_yedge)];
```

With alignment:

```
int_aux1_reg  =  moai4k_sdram.sdram  [pointer_to_sdram  +  (int_gap_length_reg*
int_yedge)];

int_aux2_reg  =  moai4k_sdram.sdram  [pointer_to_sdram  +  (int_gap_length_reg*
int_yedge)+1];

int_final_reg [63:8*int_row_reg] =   int_aux1_reg [((8- int_row_reg)*8)-1:0];

int_final_reg [(8* int_row_reg)-1:0] =  int_aux2_reg [63: (8- int_row_reg)*8];

moai4k_mem.mem [pointer_to_mem] = int_final_reg;
```

- If the block is completely within the image, so it is not is necessary to do the padding. It would be the following case:

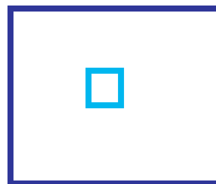


Figure 4.14. *The block is completely within the image*

The variable int_yedge will be zero, so the code is like the explained in the previous case.

- If the block is outside of the image for the top, it is necessary to do the padding. It would be the following case:

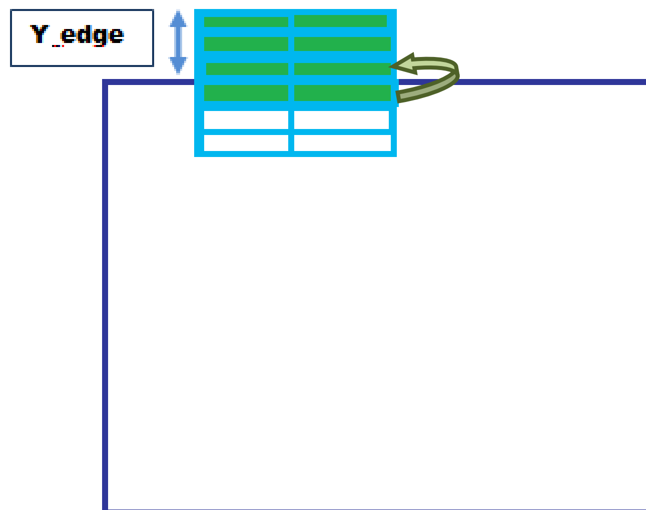


Figure 4.15. *The block is outside of the image for the top*

In this case, it will be necessary to copy the content of the first segment which is within the image, in the other segments that are outside of the image. So the pointer moves int_gap_length_reg* int_yedge and lies within the image.

When the DMA Unit arrived at the end of a segment, int_yedge= int_yedge-1 until it is in the image.

- If the block is outside of the image from the bottom, it is necessary to do the padding. It would be the following case:

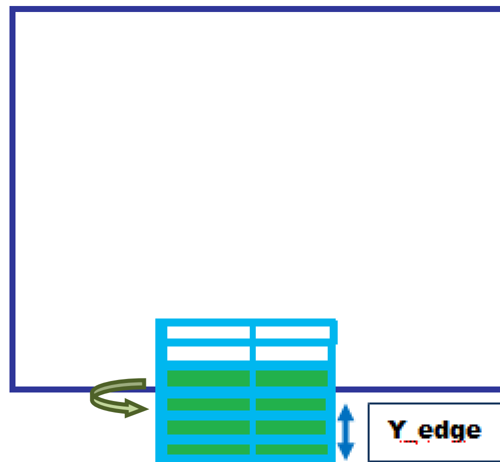


Figure 4.16.*The block is outside of the image for the bottom*

In this case, it will be necessary to copy the content of the last segment which is within the image, in the other segments that are outside of the image.

When int_yedge is negative, this variable will change its value to zero. The DMA will begin transmitting lines that are within the image, and when it has no more lines within the image, the DMA will have to re-transmit the contents of the last line. Therefore that point will be re-pointer to the last line in the image (int_gap_length_reg = 0).

```
pointer_to_sdram = int_ext_base_addr_reg [SDRAM_ADDR_W-1+3:0+3] + int_gap_length_reg

int_ext_base_addr_reg [SDRAM_ADDR_W-1+3:0+3] = int_ext_base_addr_reg [SDRAM_ADDR_W-1+3:0+3] + int_gap_length_reg
```

CASE 2: The block is outside of the image to the left side, the block is outside of the image to the left side and is outside of the image for the top, or the block is outside of the image to the left side and is outside of the image from the bottom.

- If the block is outside of the image to the left side, it is necessary to do the padding. It would be the following case:

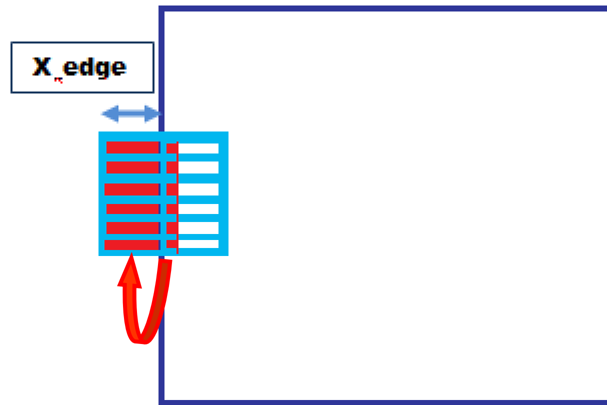


Figure 4.17. *The block is outside of the image to the left side*

In this case, it will be necessary to copy the content of the first subword which is within the image in the other words that are outside of the image. So the pointer moves int_xedge, lies within the image and copy the first word.

```
int_new_reg = moai4k_sdram.sdram [pointer_to_sdram + (int_gap_length_reg*
int_yedge) + int_x_edge]
```

In this case the variable int_yedge will be zero, therefore the pointer moves int_xedge. The DMA will store the last 8 bits of this register in another, because these data are those which will have to copy in all subword of the words that are out of the image.

```
int_seg_new_reg = int_new_reg [63:56]
```

Here are two cases:

- Only one word is outside of the image and it is necessary the alignment.

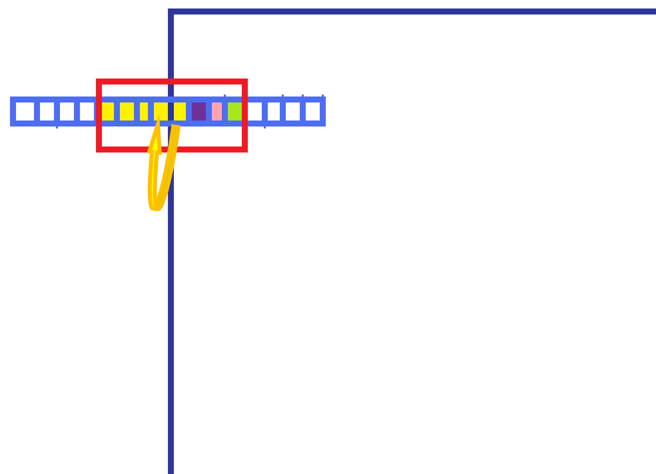


Figure 4.18. *The block is outside of the image to the left side and only one word is outside of the image*

In this case the DMA will have to copy the content of the first subword which is within the image in all subwords, until begins the word that is within the image. When the block is within the image, the DMA will copy the subword that have to copy of the first word which is within the image, to complete the word.

```
for (i = int_row_reg; i<8; i++) {

   int_def_reg [7+i*8: i*8] = int_seg_new_reg;

}

int_final_reg[63:8*int_row_reg]=int_def_reg[63:8*int_row_reg]

int_final_reg[(8*int_row_reg)-1:0]=int_new_reg[63:(8-

                                          int_row_reg)*8]

moai4k_mem.mem [pointer_to_mem]       =    int_final_reg
```

The variable int_xedge will be zero, because there are no more words outside of the image.

- There are more than a word that is outside of the image or only one word is outside of the image and it is not necessary the alignment because the word is aligned.

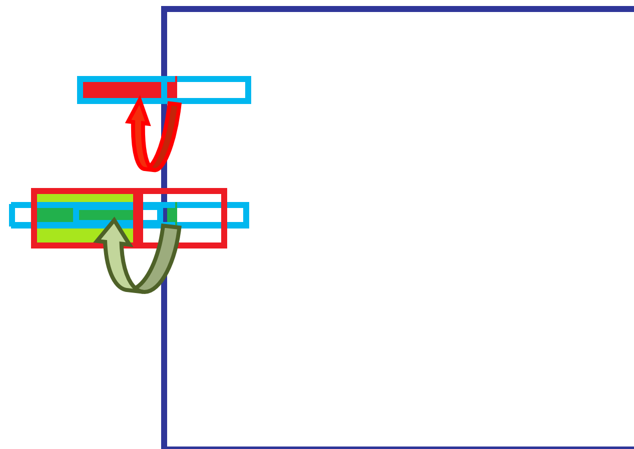


Figure 4.19.*The block is outside of the image to the left side and only one word is outside of the image (this word is aligned) or the block is outside of the image to the left side and more than a word is outside of the image.*

The DMA Unit do not care if there is alignment to do or not when there is more than a word outside of the image, because it will have to copy the content of the first subword which is within the image in all subwords of the words. If only one word is outside of the image and is not necessary the alignment because the word is aligned, also the DMA Unit will have to copy the content of the first subword which is within the image in all

subwords of the word. After this, int_xedge is decremented by 1 because there is a word less to transmit.

```
for (i = 0; i<8; i++) {
     int_def_reg [7+i*8: i*8] = int_seg_new_reg;
                        }
     moai4k_mem.mem [pointer_to_mem] = int_def_reg;
```

- If the block is outside of the image to the left side and too is outside of the image for the top, is necessary to do the padding. It would be the following case:

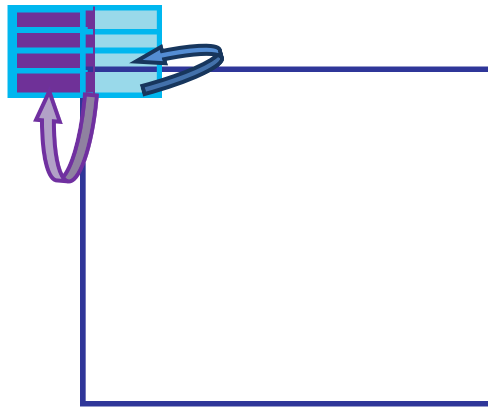


Figure 4.20.*The block is outside of the image to the left side and for the top*

The code and the cases will be the same as in previous case, only that in this case the variable int_yedge will not be zero, so the pointer moves (int_gap_length_reg* int_yedge) + int_xedge, and lies within the image.

```
int_new_reg  =  moai4k_sdram.sdram  [pointer_to_sdram  +  (int_gap_length_reg*
int_yedge) + int_x_edge]
```

When the DMA Unit arrived at the end of a segment, int_yedge= int_yedge-1 until it is vertically in the image.

- The block is outside of the image to the left side and too is outside of the image from the bottom, is necessary to do the padding, would be the following case:

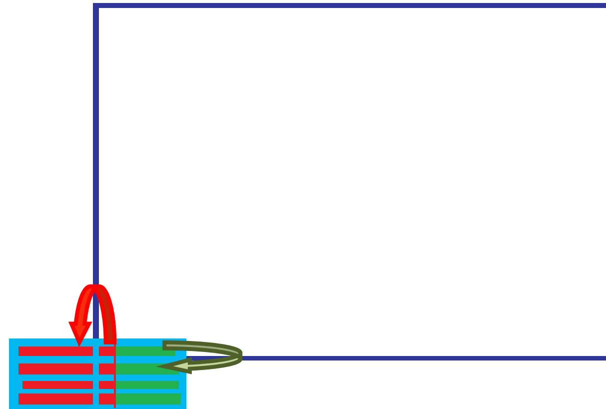


Figure 4.21.*The block is outside of the image to the left side and for the bottom*

The code and the cases will be the same as in previous case, when int_yedge is negative, this variable will change its value to zero therefore in this case, the variable int_yedge will be zero, so the pointer moves int_xedge.

```
int_new_reg = moai4k_sdram.sdram [pointer_to_sdram + (int_gap_length_reg*

                 int_yedge) + int_x_edge]
```

The DMA will begin transmitting lines that are vertically within the image, and when it has no more lines within the image, it will have to re-transmit the contents of the last line. Therefore that point will be re-pointer to the last line in the image (int_gap_length_reg = 0).

```
pointer_to_sdram = int_ext_base_addr_reg [SDRAM_ADDR_W-1+3:0+3] +

                        int_gap_length_reg

int_ext_base_addr_reg [SDRAM_ADDR_W-1+3:0+3] = int_ext_base_addr_reg

                                [SDRAM_ADDR_W-1+3:0+3] + int_gap_length_reg
```

CASE 3: The block is outside of the image to the right side, the block is outside of the image to the right side and too is outside of the image for the top, or the block is outside of the image to the right side and too is outside of the image from the bottom.

- The block is outside of the image to the right side.

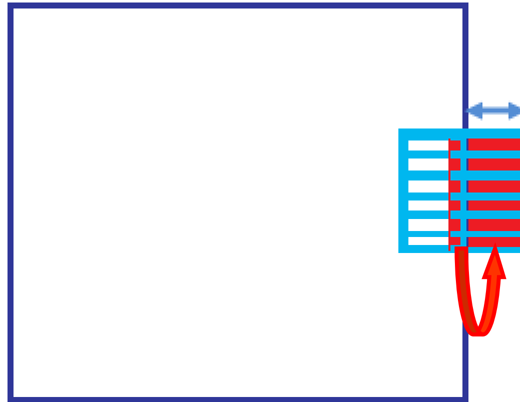


Figure 4.22. *The block is outside of the image to the right side*

When the block is outside of the image to the right side, the DMA Unit will have to transmit the words of the block that are within the image and when the block is outside, the DMA Unit will have to copy the content of the last subword which is within the image in the other words that are outside of the image.

To see the number of words that the DMA Unit should transmit before the block leave out, and therefore to copy the content of the last subword which is within the image in the other words that are outside of the image, it is necessary to create a variable to indicate and that only be update when changing line.

```
if (pointer_to_sdram = int_ext_base_addr_reg [SDRAM_ADDR_W-1+3:0+3]){
                int_seg_in = int_segment_length_reg_aux + int_xedge_aux;
}
```

The number of words of the line (or segment) minus the number of words that are out of the image gives the words that are inside (int_xedge in this case is negative).

Here are two cases:

- The words are still in the picture horizontally or there is a word of the block that is half within half outside of the image horizontally.

  Without alignment: there is a word of the block that is completely inside of the image horizontally or more than one word.

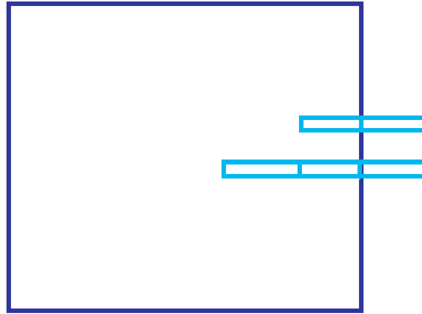


Figure 4.23.*The block is outside of the image to the right side and there is a word of the block that is completely inside of the image horizontally or more than one word.*

```
moai4k_mem.mem[pointer_to_mem] = moai4k_sdram.sdram [pointer_to_sdram
                                            +(int_gap_length_reg* int_yedge)];
```

In this case int_yedge is zero. Each time that the DMA transmits a word, the number of words inside the image is decremented a unit.

<u>With alignment:</u> Here are two cases.

- There is only a word of the block inside of the image; the word is half within half outside of the image horizontally.

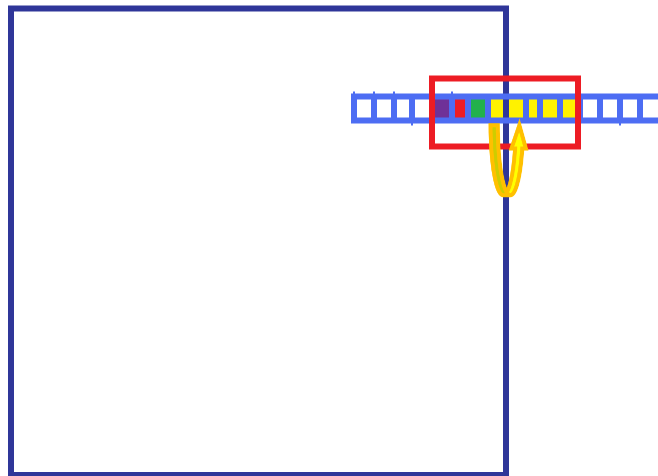


Figure 4.24.*The block is outside of the image to the right side and there is only a word of the block inside of the image; the word is half within half outside of the image horizontally*

The DMA Unit begins transmitting the subword that are within the image until it leaves out, in this moment the DMA Unit will have to copy the content of the last subword which is within the image in all subwords that are outside of the image.

```
int_new_reg = moai4k_sdram.sdram [pointer_to_sdram +
                              (int_gap_length_reg*int_yedge)]
int_seg_new_reg = int_new_reg [7:0]
for (i = 0; i<int_row_reg;i++) {
              int_def_reg [7+i*8: i*8] = int_seg_new_reg;
              }
int_final_reg [(8* int_row_reg)-1:0]   =   int_def_reg [7+
              (int_row_reg-1)*8:0];
int_final_reg [63:8*int_row_reg] =   int_new_reg [((8-
              int_row_reg)*8)-1:0];
moai4k_mem.mem [pointer_to_mem] = int_final_reg;
```

In this case int_yedge is zero. Each time that the DMA transmit a word, the number of words inside the image is decremented a unit.

- There is more than a word of the block inside of the image.

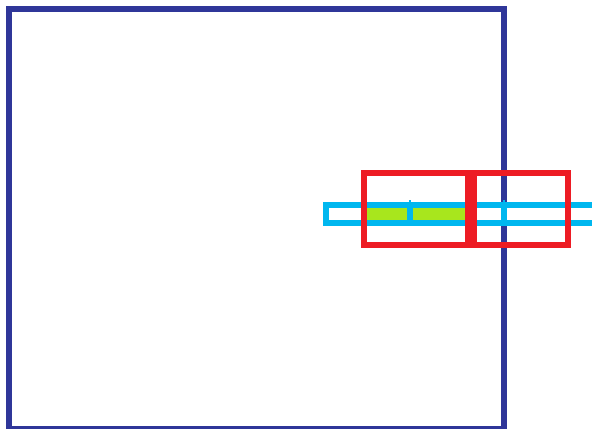


Figure 4.25. *The block is outside of the image to the right side and there is more than a word of the block inside of the image*

The DMA will perform the alignment as usual.

```
int_aux1_reg = moai4k_sdram.sdram [pointer_to_sdram +
                                        (int_gap_length_reg*int_yedge)];
int_aux2_reg = moai4k_sdram.sdram [pointer_to_sdram +
                                      (int_gap_length_reg*int_yedge)+1];
int_final_reg [63:8*int_row_reg] =    int_aux1_reg [((8-
                                                  int_row_reg)*8)-1:0];
int_final_reg [(8* int_row_reg)-1:0] =  int_aux2_reg [63: (8-
                                                      int_row_reg)*8];
moai4k_mem.mem [pointer_to_mem] = int_final_reg;
```

In this case int_yedge is zero. Each time that the DMA transmit a word, the number of words inside the image is decremented a unit.

- The words are outside of the picture horizontally.

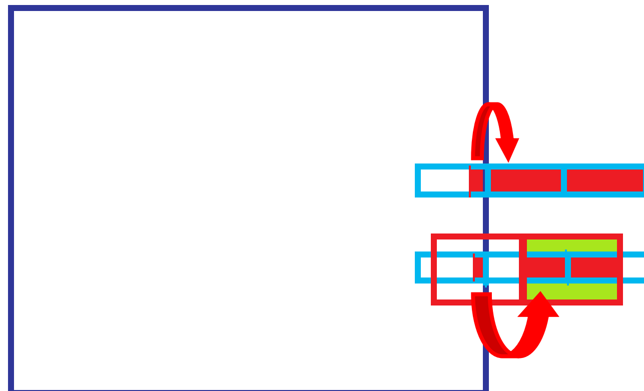


Figure 4.26.*The block is outside of the image to the right side and the words are outside of the image horizontally*

DMA Unit does not care if there is alignment to do or not when there is more than a word outside of the image, because the DMA will have to copy the content of the last subword which is within the image in all subwords of the words. If only one word is outside of the image and is not necessary the alignment because the word is aligned, also the DMA Unit will have to copy the content of the last subword which is within the image in all subwords of the word.

A variable, that increases one unit each time you go to transmit a word that is outside of the image, has been created to be able to point to the last word that is within the image.

```
int_new_reg = moai4k_sdram.sdram [pointer_to_sdram +
                                 int_gap_length_reg* int_yedge) - int_a];
int_seg_new_reg = int_new_reg [7:0];
for (i = 0; i<8; i++) {
       int_def_reg [7+i*8: i*8] = int_seg_new_reg;
       }
moai4k_mem.mem [pointer_to_mem] = int_final_reg;
int_xedge = int_xedge +1;
```

In this case int_yedge is zero. Each time that the DMA transmit a word, the number of words outside the image is incremented a unit because int_xedge in this case is negative.

- If the block is outside of the image to the right side and is outside of the image for the top, is necessary to do the padding. It would be the following case:

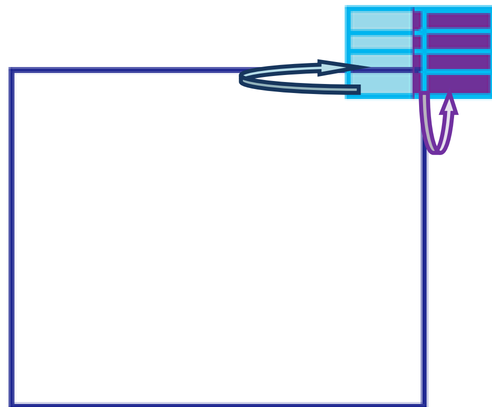


Figure 4.27.*The block is outside of the image to the right side and also for the top*

The code will and the cases will be the same as in previous case, only that in this case the variable int_yedge will not be zero.

When DMA Unit arrived at the end of a segment (or line), int_yedge= int_yedge-1 until it will be vertically in the image.

- If the block is outside of the image to the right side and too is outside of the image from the bottom, is necessary to do the padding. It would be the following case:

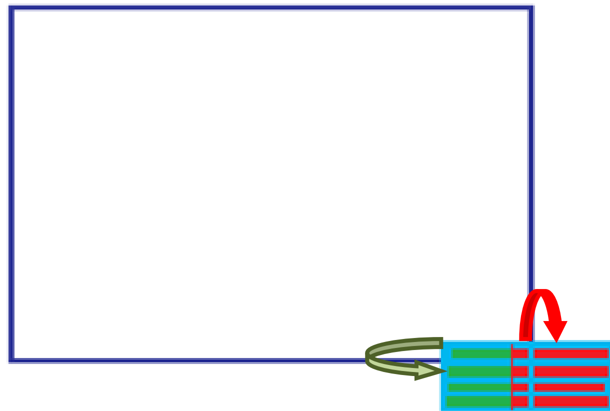


Figure 4.28.*The block is outside of the image to the right side and also from the bottom*

The code and the cases will be the same as in previous case, when int_yedge is negative, this variable will change its value to zero.

The DMA Unit will begin transmitting lines that are vertically within the image, and when it has no more lines within the image, will have to re-transmit the contents of the last line. Therefore the pointer will be re-pointer to the last line in the image (int_gap_length_reg = 0).

```
pointer_to_sdram = int_ext_base_addr_reg [SDRAM_ADDR_W-1+3:0+3] +
                   int_gap_length_reg
int_ext_base_addr_reg [SDRAM_ADDR_W-1+3:0+3] = int_ext_base_addr_reg
                         [SDRAM_ADDR_W-1+3:0+3] + int_gap_length_reg
```

In applications like video coding and decoding that use ME and MC algorithms, perform these padding with DMA would reduce considerably the overhead of the CPU, because every time that a candidate goes beyond the search area, the CPU must do extra work, a work that will save if it is performed by the DMA. Besides if the CPU doesn`t have to do this work, CPU can perform other tasks while the DMA makes the padding.

Therefore performing the padding with DMA, the results of video applications that use ME and MC algorithms will significantly improve.

4.5. - Queue implementation

This section introduces the concept of queues and presents the proposal of a DMA with a queue in order to be able to program all DMA transfers at one.

4.5.1. - Introduction

In applications like video coding and decoding that use ME and MC algorithms, when we located the candidates in the reference frame, we must transmit several blocks, and every time you transmit a block, it is necessary to interrupt the CPU and make it work (Register-Based DMA) loading data for the next transmission.

Therefore it would be interesting that the DMA was able to store several transmission in a queue, because in this way when I finish to make a transfer, will not be necessary to re-schedule the next because that will have queued and directly will load the necessary data to perform the next transmission. This also would reduce considerably the overhead of the CPU.

The idea would be that the **'Enhanced DMA Unit'** in addition to the alignment of data and make padding of data, can store several transmissions in a queue, because so the DMA would reduce considerably the overhead of the CPU.

4.5.2. - DMA with queue

A DMA Unit capable of storing several transmissions in a queue has been implemented. When a transfer finishes, it will not be necessary to schedule in the application code the configuration of the next transfer because that will have queued and directly will load the necessary data to perform the next transmission.

To implement the DMA Unit a new register has been created in the DMA Unit:

- **DMA_READ_OUT**: It is a register of one bit. When the user wants to know if the DMA has finished with a determined transfer or if it is still transmitting a determined transfer. This register can be used to increment an internal read pointer.

Moreover to implement the DMA are necessary three internal pointers:

- **Write**: Indicates the number of transmissions written. It is incremented whenever the data of a new transmission (a new block) are written.
- **Read**: Indicates the number of transmission of which the user wants to know the state. It is incremented whenever that the register DMA_READ_OUT is high.

- **Transfer**: Indicates the number of the actual transmission. It is incremented whenever a transmission of a block has finished.

When one of these variables reaches the limit of the queue, is set to zero, and the registers that use this variable as index, are overwritten.

To better understand the mode of operation, herewith is shown the Figure 4.29:

DMA Transmission

DMA Transmission

DMA_READ_OUT=1 because the user need to know when the second transmission has finished for an operation that makes in the future.

DMA Transmission

…

OPERATION

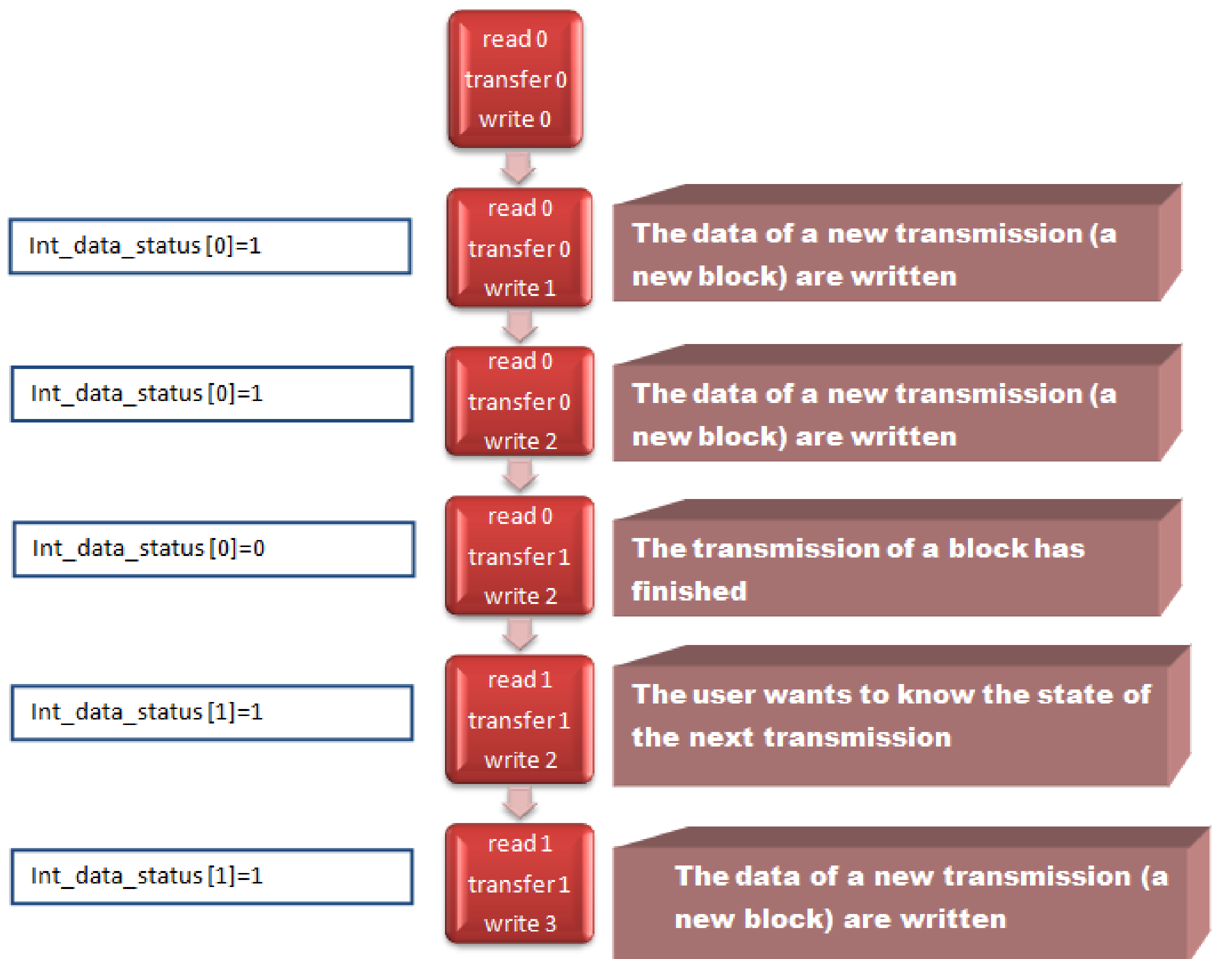


Figure 4.29.*Mode of operation of the queues*

To better understand the use of DMA_READ_OUT, herewith is shown the Figure 4.30:

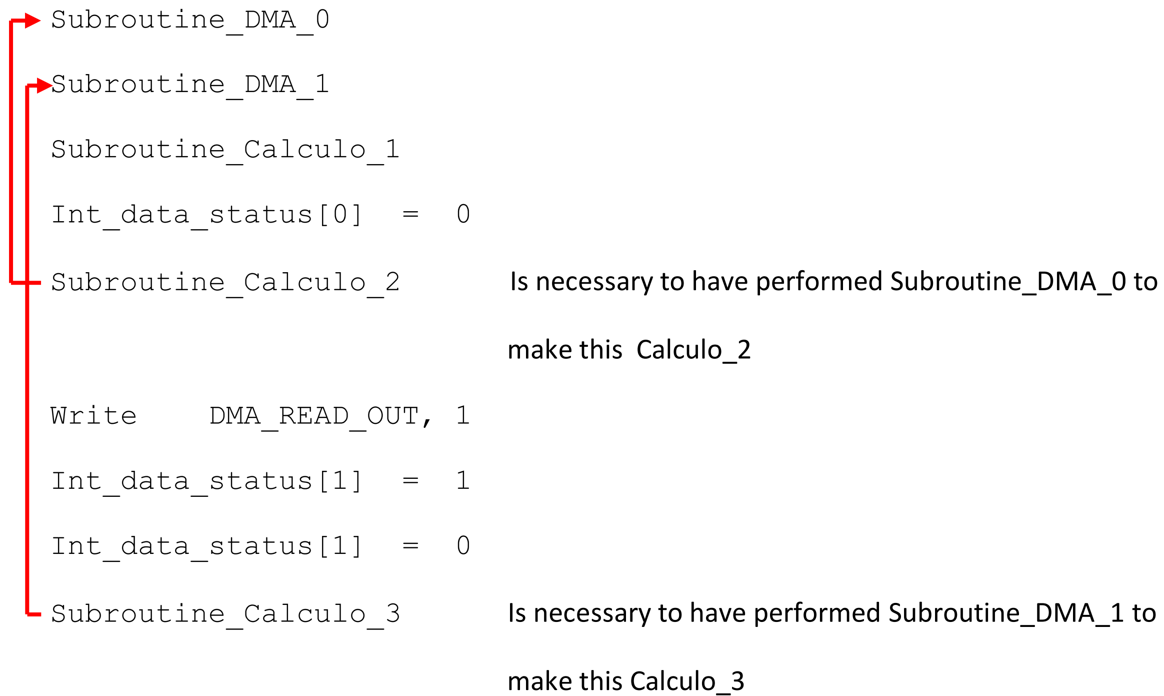


Figure 4.30.*Use of DMA_READ_OUT*

In applications like video coding and decoding that use ME and MC algorithms, a DMA Unit that can store several transmissions in a queue, reducing considerably the overhead of the CPU.

Chapter 5

Evaluation of the proposed enhanced DMA Unit

This section presents an evaluation of the proposal DMA Unit using Motion Estimation algorithm. The design of the exploration space, the characteristics of the application (motion estimation) and the results are explained in this section.

5.1. - Design Space Exploration

In this section it is explained the system environment and the motion estimation algorithm, indicating the reasons why it has been chosen as application.

5.1.1. - System description

The system with which works this project is the following:

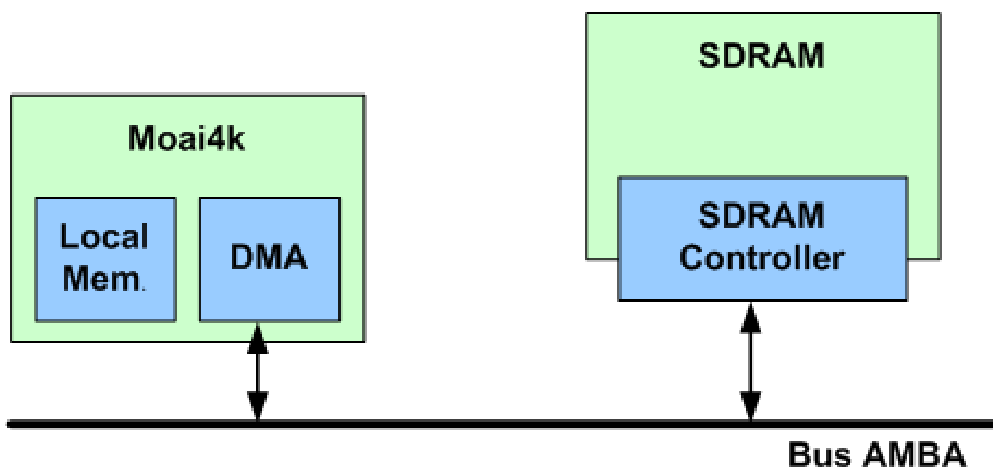


Figure 5.1.*System*

The project works with the **'Enhanced DMA Unit'**, which is inside the Moai4k processor.

There are three profiles of the DMA Unit that will be analyzed:

- **Basic Profile**, is the basic DMA Unit.
- **Advanced Profile 2**, is the Basic Profile with alignment and padding.
- **Advanced Profile 3**, is the Advanced Profile 2 with queues.

The **'Enhanced DMA Unit'**, communicates with the external memory (SDRAM) through the bus AMBA.

With the aim of obtaining results more realistic, **latency** will be introduced into the external memory controller. Therefore it will be a latency each time a line of a block of data is transmitted. The results will be obtained with different values for this latency.

5.1.2. - Motion Estimation task

Using the **'Enhanced DMA Unit'** proposed and the motion estimation algorithm as an application, the advantages of this DMA Unit will be demonstrated.

The motion estimation algorithm has been selected for the following reasons:

- With motion estimation algorithm, it is not possible to avoid unpredictable unaligned memory references, which result in lot of **unpredictable unaligned accesses**.

  This is because to identify a pixel block for the reference frame that best matches the block to be coded, the reference block is generated by displacement from the location of the block to be coded in the frame of reference. This displacement is performed using the motion vector and can be arbitrary.

  In applications like video coding and decoding that use ME algorithm, current software optimizations become unsuccessful because the instructions that are necessary for doing the data realignment in software subject the CPU to a heavy overhead.

- With motion estimation algorithm, to search the block of the reference frame that best matches the block to be coded, the search is only in a region known as the search area, then when a candidate (to be a pixel block from the reference that best matches the block to be coded) goes beyond this search area, it is necessary to perform **padding**.

In applications like video coding and decoding that use ME algorithm, if in addition to the alignment of data, padding of these is made, would be reduced considerably the overhead of the CPU.

- With motion estimation algorithm, when the candidates are located in the reference frame, the DMA Unit must transmit several blocks, and every time you transmit a block, it is necessary to interrupt the CPU and make it work (Register-Based DMA) loading data for the next transmission.

  In applications like video coding and decoding that use ME algorithm, it is interesting that the DMA Unit was able **to store several transfer configuration in a queue**, because in this way when a transfer finishes, it will not be necessary to configure the next transfer because this configurations was stored in a queue before, then the DMA can directly start the next transmission. This, in addition to the alignment of data and padding of data, would reduce considerably the overhead of the CPU.

Consequently, with this 'Enhanced DMA Unit', the results of video applications that use ME algorithm will be significantly improved.

The motion estimation algorithm has been programmed in assembly language. The structure of the code is shown in figure 5.2.

As can be seen in Figure 5.2, every time the DMA Unit must send something, at the same time other tasks are being executed. Thus, the algorithm is stopped the shortest time possible.

Once the first candidate has been obtained, the algorithm goes into a loop. Just enter into the loop, the next candidate is requested, and at the same time another tasks are being executed on the current candidate. This loop is repeated until there are no more candidates.

The algorithm has been implemented in this way because it has been proved that is the optimal way to schedule the transfer if we have to wait to complete a transfer in order to program the following.

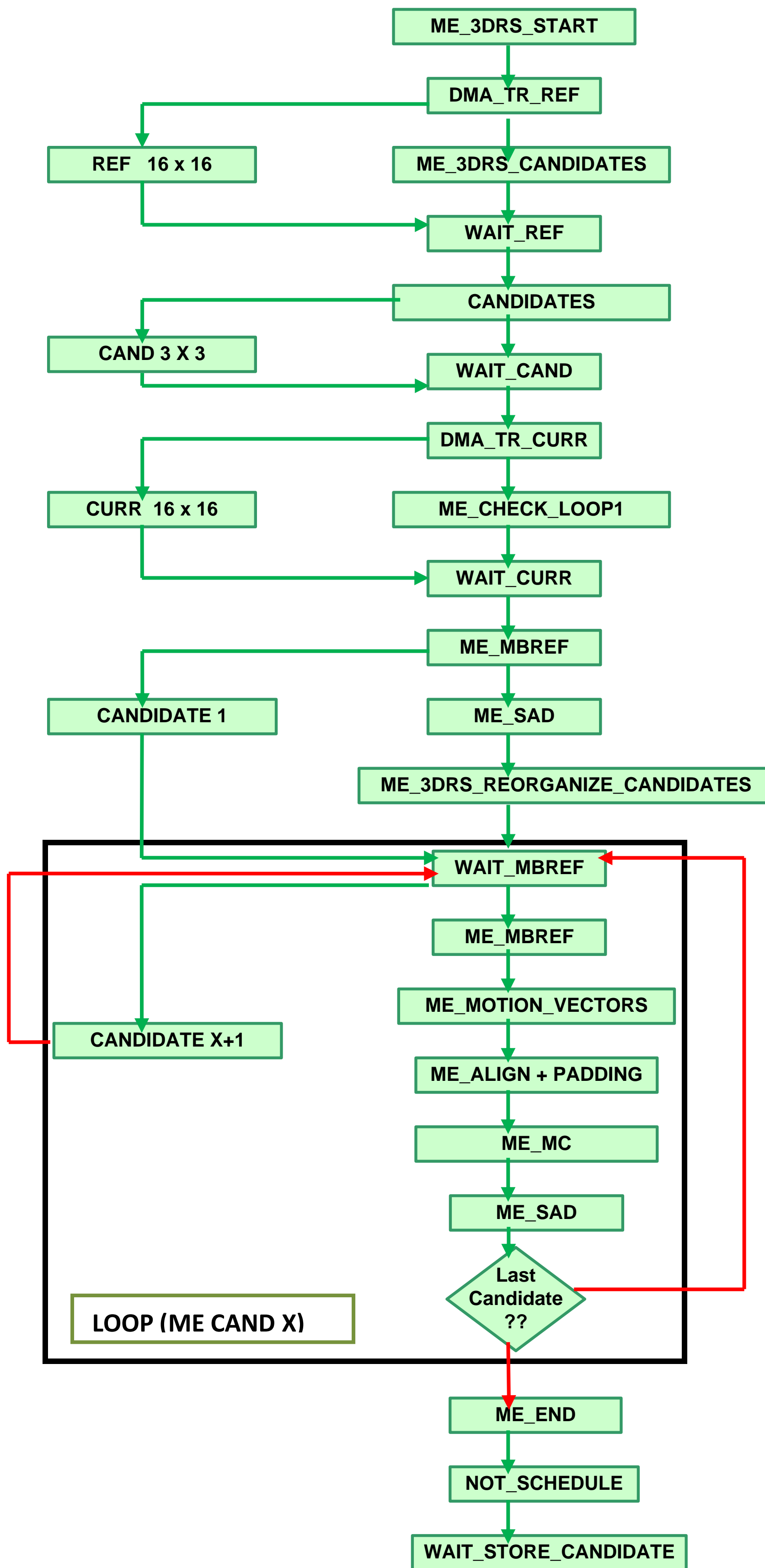


Figure 5.2.*Motion Estimation algorithm*

5.2. - Characteristics of motion estimation algorithm

The characteristic of motion estimation are the following:

- Use SAD for calculating block matching.
- Employ the algorithm 3DRS to search the candidates.
- The search area is the entire image.
- Although the standard H.264/AV allowed working with multiple images of reference, the application only works with the previous image as a reference.
- In the application is possible to point to candidate blocks placed only at half-pixel (of course also in full position) and will have the following cases:

  ✓ FH: Average vertical (green circle).
  ✓ HF: Average horizontal (blue circle).
  ✓ HH: Average vertical and average horizontal (yellow circle).
  ✓ FF: copy (red circle).

  These cases are presented in Figure 3.9.

- Really Works with blocks of 16x16, but as is possible to point to candidate blocks placed at half-pixel, the DMA Unit will take of memory blocks of 17x17 due to half-pixel.

5.3. - Results

Using the proposed **'Enhanced DMA Unit'** and the motion estimation algorithm as an application, the following analysis has been made:

- **Basic Profile** with different latencies between 15 and 55 cycles. The code used for the motion estimation algorithm is the explained in 5.1.2.

- **Advanced Profile 2** with different latencies between 15 and 55 cycles. The code used for the motion estimation algorithm is the explained in 5.1.2 but without ME_ALIGN +PADDING, because now the alignment and padding process, it is performed by the DMA Unit.

- **Advanced Profile 3** with different latencies between 15 and 55 cycles. In this case, it will be not necessary to wait an end of a transfer to schedule the next, with advanced profile 3 several transfers may be scheduled at the same time (in the code explained in 5.1.2, the transfer of the reference block, the current block and the candidates block may be scheduled at the same time). Therefore in the code explained in 5.1.2, the WAITS only will be used in case that is

necessary to expect a transmission, but now are not required to configure the next transfer. In the loop, in this case, there are two candidates in the queue while the previous candidate is performing its operations.

5.3.1. - Results of Basic Profile (BP)

For Basic Profile with a latency of 15 cycles, the results are shown in figure 5.3.

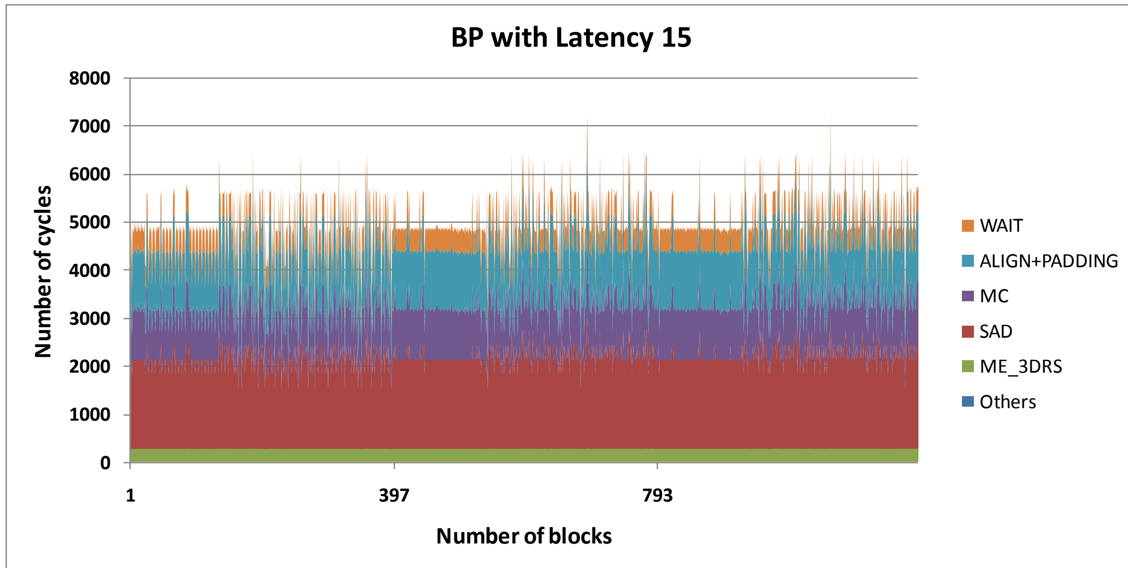


Figure 5.3.*Basic Profile with latency of 15 cycles*

On the x-axis is represented the number of block, indicating the number of frames, the graphic has a number of blocks corresponding to three frames. In the y-axis are represented the number of cycles. In the graphic can be seen the number of cycles that each block use in performing wait, alignment + padding, motion compensation, SAD, ME_3DRS and other tasks. Also it can be seen the total number of cycles used for each block by identifying a block from the reference frame that best matches the current block.

Peaks appears on the graphic because when there are two equals candidates, two motions vectors equal, it is used only once this motion vector with the motion compensation, therefore we have a candidate less.

Computing the average cycles which employs each of the blocks in performing each task that is shown in the graphic, we get the following percentages of executed cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm:

- WAIT : 9,4%
- ALIGN+PADDING: 25,1%
- MC: 21,6%
- SAD: 38,1%

- ME_3DRS: 5,6%
- Others: 0,2%

The total number of cycles employees in average by each blocks in running the algorithm is 4990 cycles.

For Basic Profile with a latency of 30 cycles, the results are shown in figure 5.4.

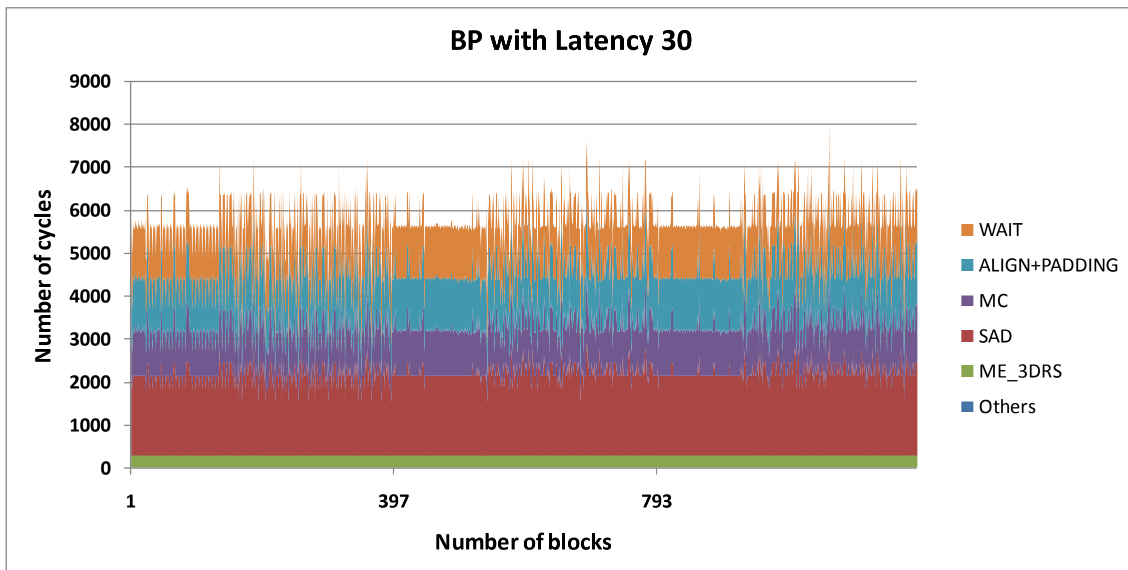


Figure 5.4.*Basic Profile with latency of 30 cycles*

Computing the average cycles which employs each of the blocks in performing each task that is shown in the graphic, we get the following percentages of executed cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm:

- WAIT : 21,1%
- ALIGN+PADDING: 21,8%
- MC: 18,9%
- SAD: 33,1%
- ME_3DRS: 4,9%
- Others: 0,2%

The total number of cycles employees in average by each blocks in running the algorithm is 5734 cycles.

For Basic Profile with a latency of 45 cycles, the results are shown in figure 5.5.

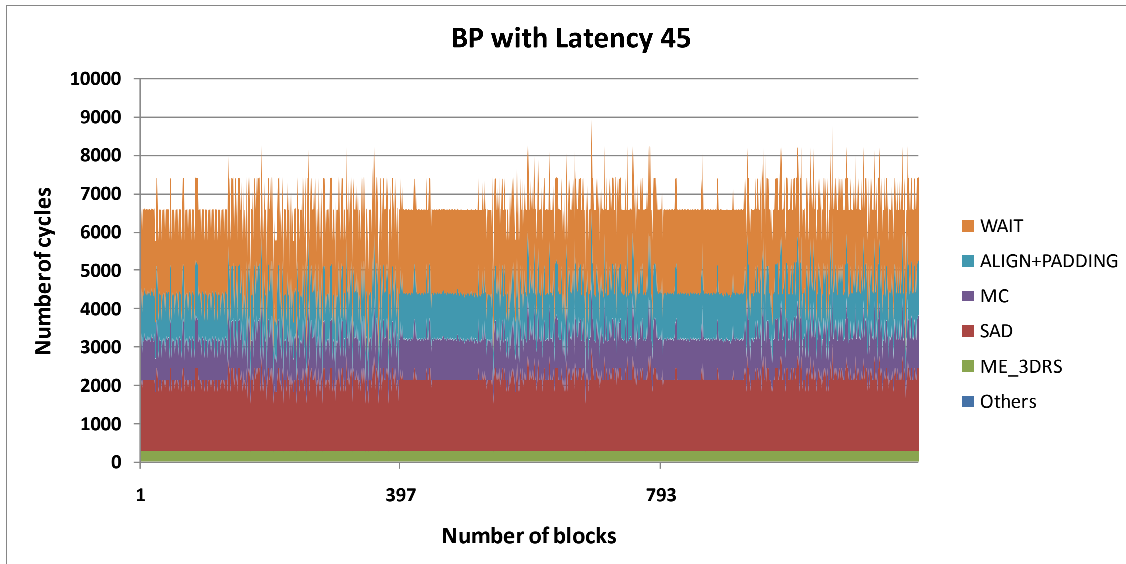


Figure 5.5.*Basic Profile with latency of 45 cycles*

Computing the average cycles which employs each of the blocks in performing each task that is shown in the graphic, we get the following percentages of executed cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm:

- WAIT : 32,8%
- ALIGN+PADDING: 18,6%
- MC: 16%
- SAD: 28,2%
- ME_3DRS: 4,2%
- Others: 0,2%

The total number of cycles employees in average by each blocks in running the algorithm is 6728 cycles.

Analysis of the results with different latencies

It is easy to appreciate in the previous graphics that when the latency increases, increases the total number of cycles required to identify a block from the reference frame that best matches the current block. This is due to increasing the latency is equivalent to increase the number of cycles that each block use in performing WAIT, because when the latency increases, is necessary to wait longer for the end of the transfers.

To observe it clearly, Table 5.1 shows the percentages of used cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm and the total number of cycles employees in average by each blocks in running the algorithm for the different latencies:

BASIC PROFILE			
WAIT %	9,4	21,1	32,8
ALIGN+PADDING %	25,1	21,8	18,6
MC %	21,6	18,9	16
SAD %	38,1	33,1	28,2
ME_3DRS %	5,6	4,9	4,2
Others %	0,2	0,2	0,2
Total number of Cycles in average	4990	5734	6728

Table 5.1.*Basic profile with different latencies*

When the latency increases, increases the total number of cycles and the percentage of cycles that each block use in performing WAIT.

5.3.2. - Results of Advanced Profile 2 (AP2)

Now the alignment and padding of data is performed in the DMA Unit. Therefore the processor executes less code. Because of this, in all the graphics that appears in this section, the alignment +padding process require 0 cyles.

For Advanced Profile 2 with a latency of 15 cycles, the results are shown in figure 5.6.

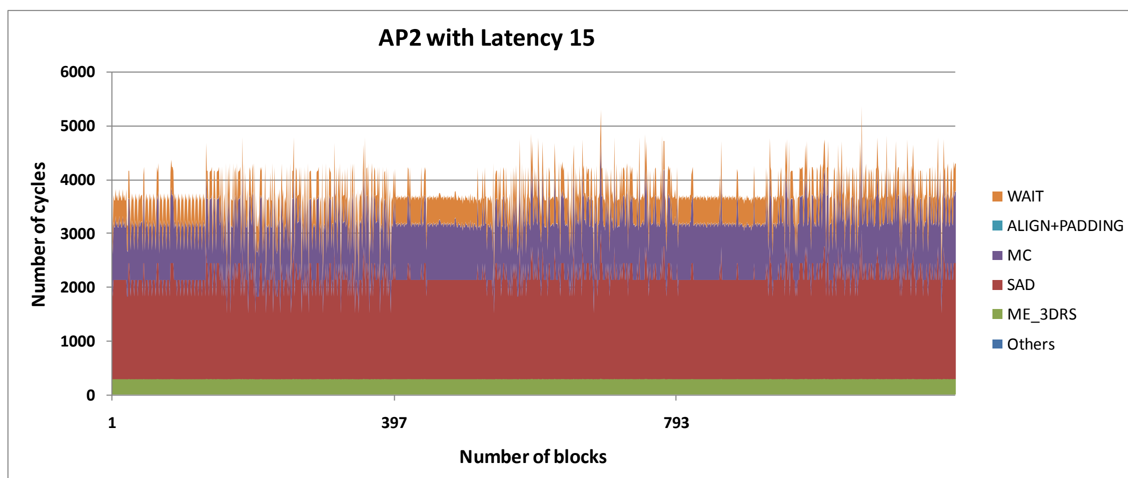


Figure 5.6.*Adavanced Profile 2 with latency of 15 cycles*

Computing the average cycles which employs each of the blocks in performing each task that is shown in the graphic, we get the following percentages of executed cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm:

- WAIT : 13,3%
- ALIGN+PADDING: 0%
- MC: 28,3%
- SAD: 50,6%
- ME_3DRS: 7,5%
- Others: 0,3%

The total number of cycles employees in average by each blocks in running the algorithm is 3749 cycles.

For Advanced Profile 2 with a latency of 30 cycles, the results are shown in figure 5.7.

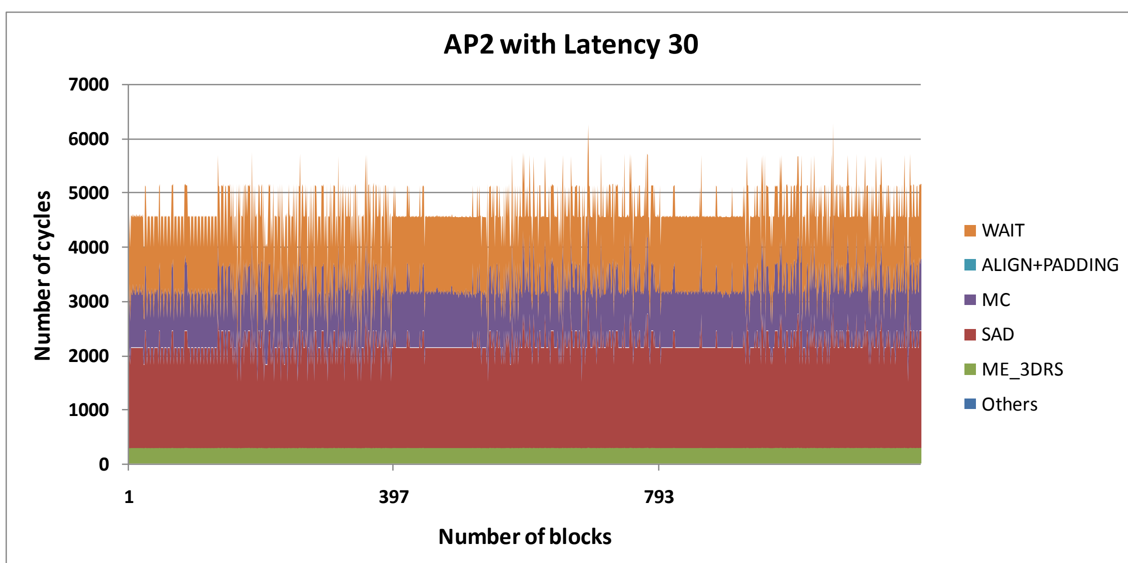


Figure 5.7.*Advanced Profile 2 with latency of 30 cycles*

Computing the average cycles which employs each of the blocks in performing each task that is shown in the graphic, we get the following percentages of executed cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm:

- WAIT : 30,2%
- ALIGN+PADDING: 0%
- MC: 22,8%
- SAD: 40,7%
- ME_3DRS: 6%
- Others: 0,3%

The total number of cycles employees in average by each blocks in running the algorithm is 4660 cycles.

For Advanced Profile 2 with a latency of 45 cycles, the results are shown in figure 5.8.

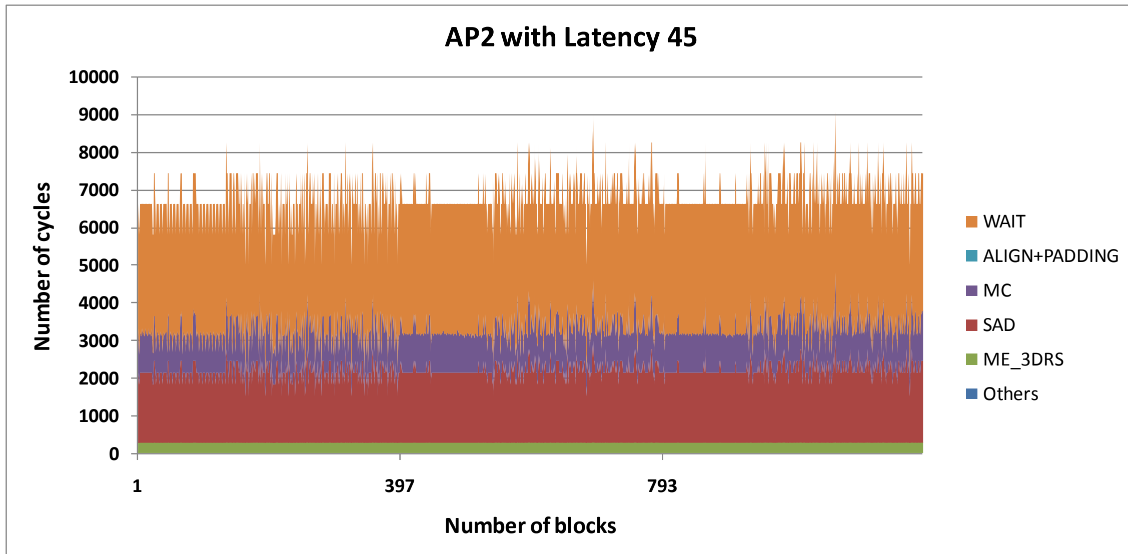


Figure 5.8.*Advanced Profile 2 with latency of 45 cycles*

Computing the average cycles which employs each of the blocks in performing each task that is shown in the graphic, we get the following percentages of executed cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm:

- WAIT : 51,9%
- ALIGN+PADDING: 0%
- MC: 15,7%
- SAD: 28,1%
- ME_3DRS: 4,2%
- Others: 0,1%

The total number of cycles employees in average by each blocks in running the algorithm is 6755 cycles.

Analysis of the results with different latencies

It is easy to appreciate in the previous graphics that when the latency increases, increases the total number of cycles used for each block by identifying a block from the reference frame that best matches the current block. This is due to increasing the latency is equivalent to increase the number of cycles that each block use in performing WAIT, because when the latency increases, is necessary to wait longer for the end of the transfers.

To observe it clearly, Table 5.2 shows the percentages of used cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm and the total number of cycles employees in average by each blocks in running the algorithm for the different latencies:

ADVANCED PROFILE 2			
WAIT %	13,3	30,2	51,9
ALIGN+PADDING %	0	0	0
MC %	28,3	22,8	15,7
SAD %	50,6	40,7	28,1
ME_3DRS %	7,5	6	4,2
Others %	0,3	0,3	0,1
Total number of Cycles in average	3749	4660	6755

Table 5.2.*Advanced Profile 2 with different latencies*

When the latency increases, increases the total number of cycles and the percentage of cycles that each block use in performing WAIT.

5.3.3. - Results of Advanced Profile 3 (AP3)

In this case, it will not be necessary to wait an end of a transfer to schedule the next, with advanced profile 3 several transfers can be programmed at the same time.

For Advanced Profile 3 with a latency of 15 cycles, the results are shown in figure 5.9.

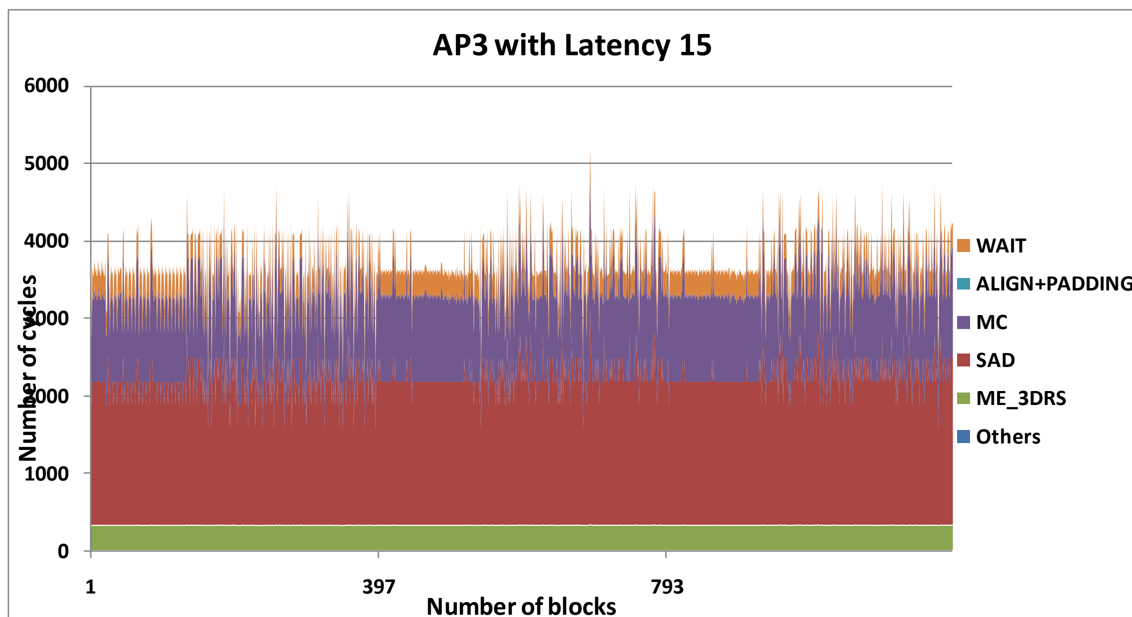


Figure 5.9.*Adavanced Profile 3 with latency of 15 cycles*

Computing the average cycles which employs each of the blocks in performing each task that is shown in the graphic, we get the following percentages of executed cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm:

- WAIT : 8,6%
- ALIGN+PADDING: 0%
- MC: 30,9%
- SAD: 51,5%
- ME_3DRS: 8,7%
- Others: 0,3%

The total number of cycles employees in average by each blocks in running the algorithm is 3672 cycles.

For Advanced Profile 3 with a latency of 30 cycles, the results are shown in figure 5.10.

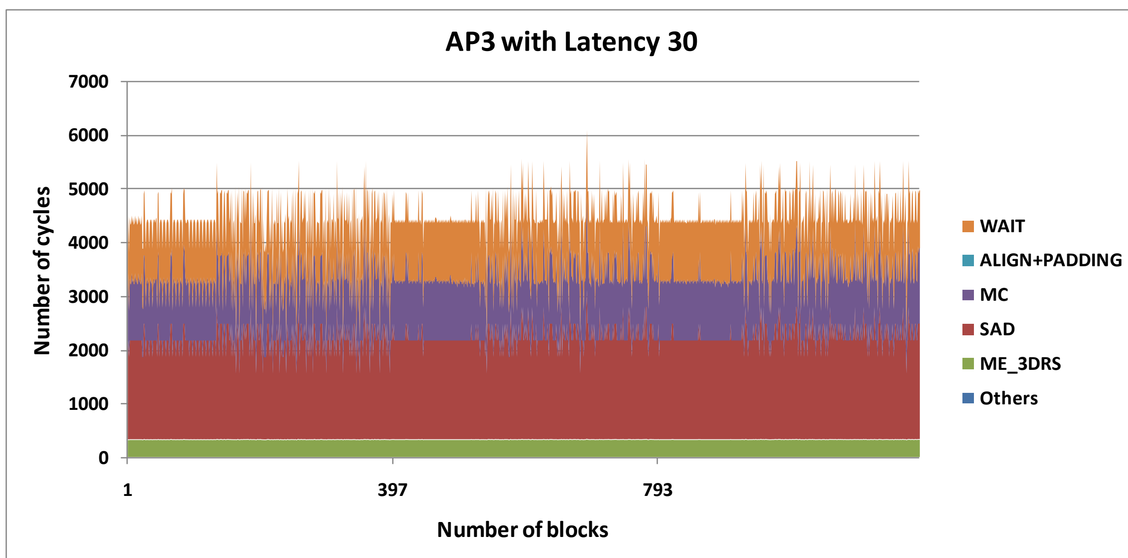


Figure 5.10.*Advanced Profile 3 with latency of 30 cycles*

Computing the average cycles which employs each of the blocks in performing each task that is shown in the graphic, we get the following percentages of executed cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm:

- WAIT : 25,2%
- ALIGN+PADDING: 0%
- MC: 25,3%
- SAD: 42,1%
- ME_3DRS: 7,1%
- Others: 0,3%

The total number of cycles employees in average by each blocks in running the algorithm is 4491 cycles.

For Advanced Profile 3 with a latency of 45 cycles, the results are shown in figure 5.11.

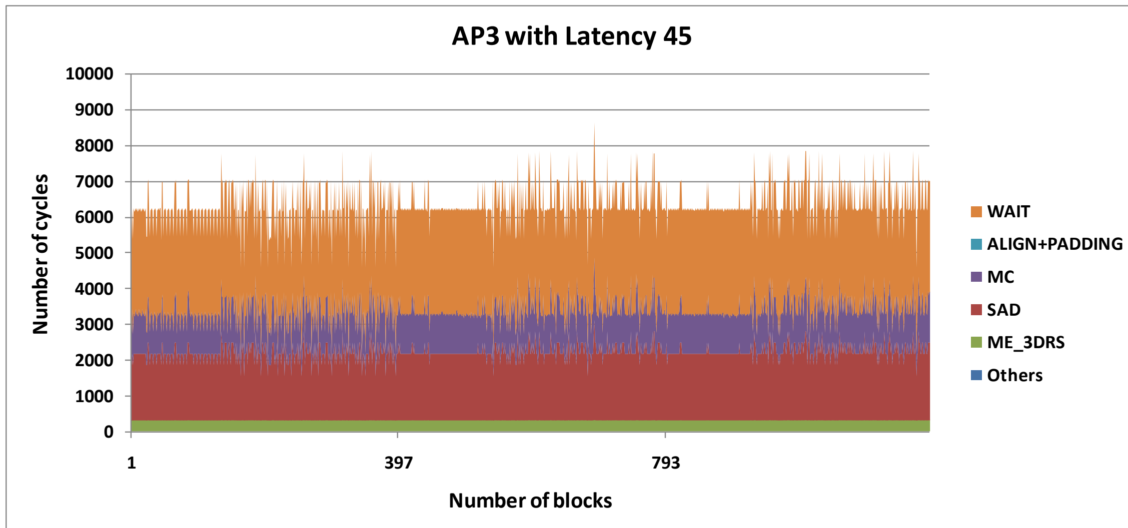


Figure 5.11.*Advanced Profile 3 with latency of 45 cycles*

Computing the average cycles which employs each of the blocks in performing each task that is shown in the graphic, we get the following percentages of executed cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm:

- WAIT : 47%
- ALIGN+PADDING: 0%
- MC: 18%
- SAD: 29,9%
- ME_3DRS: 5%
- Others: 0,1%

The total number of cycles employees in average by each blocks in running the algorithm is 6325 cycles.

Analysis of the results with different latencies

It is easy to appreciate in the previous graphics that when the latency increases, increases the total number of cycles used for each block by identifying a block from the reference frame that best matches the current block. This is due to increasing the latency is equivalent to increase the number of cycles that each block use in performing WAIT, because when the latency increases, is necessary to wait longer for the end of the transfers.

To observe it clearly, Table 5.3 shows the percentages of used cycles in performing each task with respect to the total number of cycles employees in average by each blocks in running the algorithm and the total number of cycles employees in average by each blocks in running the algorithm for the different latencies:

ADVANCED PROFILE 3			
WAIT %	8,6	25,2	47
ALIGN+PADDING %	0	0	0
MC %	30,9	25,3	18
SAD %	51,5	42,1	29,9
ME_3DRS %	8,7	7,1	5
Others %	0,3	0,3	0,1
Total number of Cycles in average	3672	4491	6325

Table 5.3.*Advanced Profile 3 with different latencies*

When the latency increases, increases the total number of cycles and the percentage of cycles that each block use in performing WAIT.

5.3.4. - Comparison of the different profiles

At this point has been made a comparison between the three profiles, taking into account the data that appears in tables 5.1, 5.2, 5.3.

If we compare the Basic Profile (Table 5.1) with the Advanced Profile 2 (Table 5.2), it can be seen that the number of cycles dedicated to performing alignment and padding has disappeared, this is because now this operations are made by DMA Unit. By removing alignment and padding, decreases the total number of cycles. Table 5.4 shows the data for a latency of 15 cycles with BP and AP2:

	BASIC PROFILE	ADVANCED PROFILE 2
WAIT %	9,4	13,3
ALIGN+PADDING %	25,1	0
MC %	21,6	28,3
SAD %	38,1	50,6
ME_3DRS %	5,6	7,5
Others %	0,2	0,3
Total number of Cycles in average	4990	3749

Table 5.4.*Data with Basic Profile and Advanced Profile 2 for a latency of 15 cycles*

If we compare the Advanced Profile 2 (Table 5.2) with the Advanced Profile 3 (Table 5.3), it can be seen that the number of cycles dedicated to wait has decreased. This is due to in this case it isn't necessary to wait the end of a transfer to configure the next one, with

advanced profile 3 several transfers can be configured at the same time. Because of that the total number of cycles has decreased. Table 5.6 shows the data for a latency of 45 cycles with AP2 and AP3:

	ADVANCED PROFILE 2	ADVANCED PROFILE 3
WAIT %	51,9	47
ALIGN+PADDING %	0	0
MC %	15,7	18
SAD %	28,1	29,9
ME_3DRS %	4,2	5
Others %	0,1	0,1
Total number of Cycles in average	6755	6325

Table 5.6.*Data with Advanced Profile 2 and Advanced Profile 3 for a latency of 45 cycles*

In conclusion we can say that AP3 is considerably better than BP and AP2 because of the total number of cycles is lower for all the latencies.

The Figure 5.12 shows a comparison of the three profiles at time:

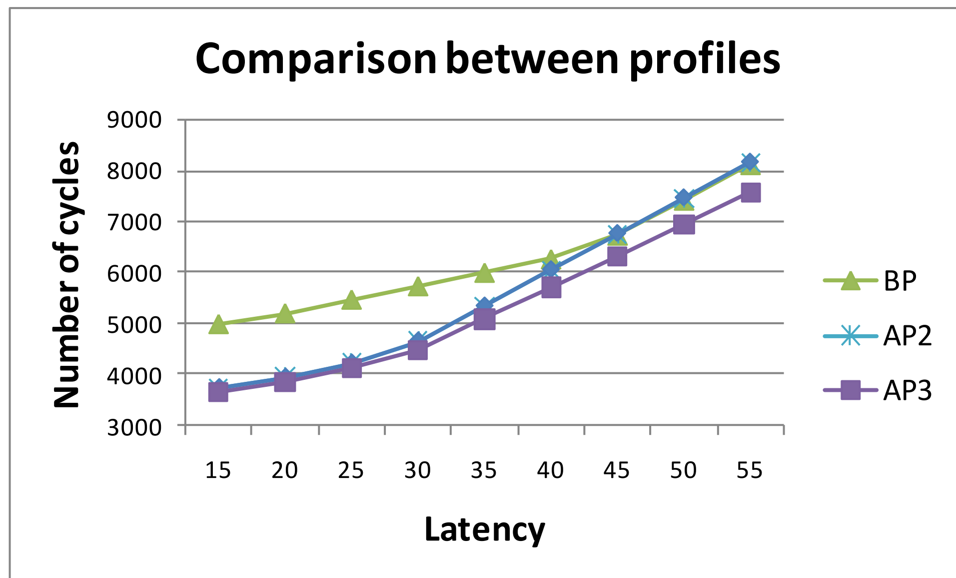


Figure 5.12.*Comparison of BP, AP2 and AP3*

On the x-axis is represented the latency, and in the y-axis is represented the average number of cycles employees by each blocks in running the algorithm.

- When the latency increases, increases the total number of cycles used for each block by identifying a block from the reference frame that best matches the current block. This is due to increasing the latency is equivalent to increase the number of cycles that each block use in performing WAIT, because when the latency increases, is necessary to wait longer for the end of the transfers.

- If we compare the Basic Profile with the Advanced Profile 2, it can be seen that with short latencies, decreases the total number of cycles with AP2 that is because it the alignment and padding process is performed in the DMA. But with large latencies, there is a latency where it takes more or less the same number of cycles in total for AP2 and BP. That is because with short latencies, for example in the loop of Figure 5.2, it will take longer to perform tasks in parallel that the time required to transfer the next candidate. Therefore, it will not be necessary to wait for the candidate and when alignment and padding is performed in the DMA, the total number of cycles is significantly reduced. By contrast, with very large latencies it will take a long time to send the candidate, more than to perform tasks in parallel, therefore it will be necessary to wait and no matter saving cycles performing alignment and padding in the DMA.
- Comparing AP2 with AP3 is easy to appreciate that AP3 uses fewer cycles than AP2 and that is more noticeable with increasing latency. With short latencies in most of the cases do not need to wait, but with large latencies it will be necessary to wait, and with AP2 we must wait longer. This is due to with AP2 is necessary to wait the end of the transfer to schedule the next and with AP3 is not necessary to wait the end of the transfers for scheduling the next.
- The code for AP2 has been implemented in order to obtain the best results taking into account that to begin a transfer must be completed the previous and because of this, the difference between the number of cycles used by AP2 and AP3 is less noticeable.
- With AP3 besides it has been saved the time required to consider the best way to implement the code taking into account that to begin a transfer must be completed the previous.
- In conclusion, with AP3 it has been decreased the required execution time, because it has been removed application code (alignment and padding subroutines) and it is not necessary to wait the end of the transfers for configuring the next. Besides with AP3, is saved the time required to consider the best way to implement the code.

Chapter 6

Conclusions

In this work it has been analyzed the VLIW and SIMD processors, the basics of Direct Memory Access Units and the environment of video coding standards. Motion estimation (a particular video coding task) has been studied, analyzing the algorithm and the transfer´s requirements.

In applications like video coding and decoding that use motion estimation, it is not possible to avoid unpredictable unaligned memory references. Motion estimation algorithm searches a block that best matches with a block from the frame to be coded. This reference block is generated by displacement from the location of the block to be coded in the frame of reference. This displacement is performed using the motion vector and can be arbitrary and therefore, that results in a lot of unpredictable unaligned accesses.

In this type of applications, current software optimizations become unsuccessful because the instructions that are necessary for doing the data realignment in software subject the CPU to a heavy overhead.

Knowing that **DMA** is an essential feature of all modern computers due to it allows any device to transfer data without subjecting the CPU to a heavy overhead, the solution adopted in this project has been to design an **'Enhanced DMA Unit'** capable of performing the alignment of data.

This **'Enhanced DMA Unit'** in addition to the alignment process can perform the padding process. For example in applications like video coding and decoding that use motion estimation algorithm, to search the block of the reference frame that best matches the block to be coded, the search is performed only in a region known as the search area, then when a candidate (to be a pixel block from the reference that best matches the block to be coded) goes beyond this search area, it is necessary to make padding.

Moreover this **'Enhanced DMA Unit'** is able to store several transfer configurations in a queue, in this way when the transfer finishes, it will be immediately initiated the next transfer. Now is not necessary to wait a previous transfer in order to program a new one.

With the proposed **'Enhanced DMA Unit'**, which is inside the Moai4k processor and communicates it with the external memory (SDRAM) through the AMBA bus, and using Motion Estimation algorithm as an application, it has been analyzed the results obtained with three different profiles:

- **Basic Profile** is a basic DMA Unit.

- **Advanced Profile 2** is a Basic Profile with the inclusion of alignment and padding process.

- **Advanced Profile 3** is the Advanced Profile 2 with an additional queue to store several transfer configurations.

With the aim of obtaining results more realistic, latency has been introduced into the external memory controller and the results has been obtained with different values for this latency (between 15 and 55 cycles).

We can conclude that:

- When the latency increases, it also increases the total number of executed cycles required to identify a block from the reference frame that best matches with a current block.

- Comparing the Basic Profile with the Advanced Profile 2, the total number of executed cycles decreases because the alignment and padding process is done by the DMA.

- Comparing AP2 with AP3, is easy to appreciate that AP3 uses fewer cycles than AP2. With an AP2 DMA it is necessary to wait the end of the transfer in order to start the next one. This "wait transfer scheduling", which is programmed in the instruction code, is avoid when using an AP3 DMA.

In conclusion with AP3 has been decreased the required execution time, because it has been removed instruction code (alignment and padding process) and is not necessary to wait the end of the transfers for programming a new one. Moreover, with an AP3, the time required to consider the best place in the application code to wait a previously transfer in order to start a new one, has been saved.

To complete this project, some ideas are proposed indicating possible future modifications.

When the padding is being done, it is necessary to copy the contents of a line in all the lines that are outside of the image, this was implemented reading all the time the same line. Therefore, reading all the time the content of this line is not necessary and it produces an unnecessary transfers expense that can be avoided by:

- Adjusting the size of the block to the contents of this that remaining within the image.

- Storing this content in the local memory, and copying these lines that are equal at this time.

The implementation of these future modifications would save a lot of resources but will be the DMA more complex.

References

1. Ruby B. Lee, “Subword Parallelism with MAX-2”, IEEE Micro, p.51-59, August 1996.

2. Guillermo Payá-Vayá, Javier Martín-Langerwerf, Piriya Taptimthong, and Peter Pirsch, “Design Space Exploration of Media Processors: A Generic VLIW Architecture and a Parameterized Scheduler”, ARCS 2007,LNCS4415, p. 154-267, March 2007.

3. A. Dasu and S. Panchanathan, “A Survey of Media Processing Approaches”, IEEE Transactions on Circuits and System for Video Technology, p. 633-645, August 2002.

4. M. Alvarez, E. Salami, A. Ramirez, and M. Valero, “Performance Impact of Unaligned Memory Operations in SIMD Extensions for Video Codec Applications”, ISPASS 2007, p. 62-71, April 2007.

5. Analog Devices, Inc One Technology Way Norwood, “Blackfin Processor Hardware Reference”, December 2005.

6. David Katz and Rick Gentile, “Using Direct Memory Access effectively in media-based embedded applications”, www.enbedded.com, November 2007.

7. John L. Hennessy and David A. Patterson (2003). “Computer Architecture: A Quantitative Approach”, The Morgan Kaufmann Series in Computer Architecture and Design.

8. Milind Phadtare, “Motion estimation techniques in video processing”, August 2007.

9. Jörn Ostermann, Jan Bormans, Peter List, Detlev Marpe, Matthias Narroschke, Fernando Pereira, Thomas Stockhammer, and Thomas Wedi, “Video coding with H.264/AVC: Tools, Performance, and Complexity”, Circuits and Systems Magazine, IEEE, p. 7-28, 2004.

10. Yasushi Ooi, “Motion Estimation System Design”, Digital Signal Processing for multimedia systems, K.K. Parhi and T. Nishitani, Marcel Dekker, Chap.12, pp. 299-327, 1999.

11. Songnan Li, Jianguo Du, Debin Zhao, Qian Huang, Wen Gao, “An Improved 3DRS Algorithm for Video De-interlacing” in Proc.Picture Coding Symp., Bejing, China, April 2006.

12. John Watkinson, “The Engineer’s Guide to Motion Compensation”, 1994: Snell & Willcox Lt.