Cache Memory Project

Version 1
4/25/2015 1:38:00 AM

Table of Contents

Main Page

Author:

Mark L. Short

Date:

Sept 25, 2014

CITE:

- Modern Processor Design, John Paul Shen, Mikko H. Lipasti, 2005
- Cache, Thomas Finley, May 2000, http://www.cs.cornell.edu/~tomf/notes/cps104/cache.html

ASSIGNMENT:

CMPS 5133 Advanced Computer Architecture Assignment #2 - Memory

Given the following benchmark code running in a four-way associative data cache with 16 blocks of 32 bytes and knowing that an integer variable occupies 4 bytes, and operations follow the usual priority, assume that at each operation the leftmost operand is fetched first and the address of A[0] is zero. Compute the number of cache misses, considering the loop index variable residing in a process register (and involved in the count of the misses) and that arrays A, B, and C reside consecutively in memory.

```
int A[512], B[512], C[512]
for (i = 0; i < 511, i++)
{
   A[i] = A[i] + B[i] + B[i+1] * C[i]
}
```

OUTCOME:

The executable compiled from the attached C++ achieves the stated purpose and outputs the results to the console window, with the following caveats:

1. No assumptions were made regarding the address of A[0] being zero. As a result, a more involved and accurate four-way associative data cache implementation was required.
2. The attached source code has been run on a x64 bit system, but compiled as a 32bit application. It is not designed or developed for porting or recompilation as a x64 bit executable. There are explicit types, type-casts and assumptions made throughout the code (i.e. casting memory address to 32-bit types) that limit it to a 32-bit executable.
3. "Handling Updates to a Block" was not considered at this time and would require further implementation.
4. Efforts were made to meaningfully test and debug the algorithms involved in this implementation. Code was added to test the validity of the data returned from cache. Due to the added coding precautions taken, an error was uncovered that would result in the following:
   ```
   Misses: 195 (est)
   Hits:  1850 (est)
   Errors:  12
   ```
5. The error was properly identified and diagnosed to be due to failure to make adjustments to the range of addresses loaded into cache to insure that they all mapped to the same corresponding "tag" + "index" address fields. This was corrected with no errors currently detected.
6. CACHE DESIGN
   - Block size = 32 bytes

- Block number = 16
- Number of sets = 4
- Block organization = (4 way) Set-associative
- Block replacement policy = FIFO
- Write policy = not implemented

7. Given the assignment formula of 'A[i] = A[i] + B[i] + B[i+1] * C[i]', the operands were accessed in the following order:
   - B[i+1]
   - C[i]
   - A[i]
   - B[i]
8. The current implementation results in the following final computation output:

```
Misses: 192
Hits:  1852
```

Errors: 0

Todo List

Member CVirtualAddress::DecodeIndex (void) const

: cleanup these hard-coded values

Member CVirtualAddress::DecodeOffset (void) const

: cleanup these hardcoded values

Class Index

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

File Index

File List

Here is a list of all files with brief descriptions:

Class Documentation

CCacheBlock Class Reference

```
#include <CacheBlock.h>
```

Public Member Functions

- CCacheBlock ()
  Default Constructor.
- ~CCacheBlock ()
  Destructor.
- void set_Tag (DWORD_PTR dwSet) throw ()
- DWORD_PTR get_Tag (void) const throw ()
- bool GetCacheData (size_t cbOffset, DWORD_PTR &dwData) const
- bool LoadCacheBlock (DWORD_PTR dwTag, const BYTE *pData, size_t cbLen=g_CACHE_BLOCK_SIZE)

Detailed Description

CCacheBlock class manages a logical unit of contiguous memory (i.e. block, entity)

Definition at line 49 of file CacheBlock.h.

Constructor & Destructor Documentation

CCacheBlock::CCacheBlock ()

Default Constructor.

Definition at line 14 of file CacheBlock.cpp.

CCacheBlock::~CCacheBlock ()

Destructor.

Definition at line 60 of file CacheBlock.h.

Member Function Documentation

DWORD_PTR CCacheBlock::get_Tag (void) const throw)

a simple data accessor

Return values:

DWORD_PTR	containing tag

Definition at line 76 of file CacheBlock.h.

bool CCacheBlock::GetCacheData (size_t *cbOffset*, DWORD_PTR & *dwData*) const

Attempts to retrieve data from cache memory based on offset

Parameters:

in	*cbOffset*	count of byte (cb) offset into cache block
out	*dwData*	output variable to return stored data value

Return values:

true	on cache hit, dwData is set
false	on cache miss, dwData is not set

Definition at line 22 of file CacheBlock.cpp.

bool CCacheBlock::LoadCacheBlock (DWORD_PTR *dwTag*, const BYTE * *pData*, size_t *cbLen* = g_CACHE_BLOCK_SIZE)

Loads a contiguous block of memory, upto CACHE_BLOCK_SIZE, into cache block. It further establishes an association with the updated data via the dwTag.

Parameters:

in	*dwTag*	value containing Tag to associate with this cache block
in	*pData*	pointer to contiguous block of memory to load
in	*cbLen*	count of bytes (cb) of data length (optional parameter)

Return values:

true	on success
false	on error

Definition at line 33 of file CacheBlock.cpp.

void CCacheBlock::set_Tag (DWORD_PTR *dwSet*) throw)

a simple data accessor

Parameters:

in	*dwSet*	

Definition at line 68 of file CacheBlock.h.

CCacheManager Class Reference

```
#include <CacheManager.h>
```

Public Member Functions

- CCacheManager ()
  Default Constructor.
- bool GetCacheData (const void *pAddress, DWORD_PTR &dwData)
- bool LoadCachePage (const void *pAddress)

Detailed Description

Definition at line 21 of file CacheManager.h.

Constructor & Destructor Documentation

CCacheManager::CCacheManager ()

Default Constructor.

Definition at line 28 of file CacheManager.h.

Member Function Documentation

bool CCacheManager::GetCacheData (const void * *pAddress*, DWORD_PTR & *dwData*)

Attempts to retrieve data from cache memory based on address

Parameters:

in	*pAddress*	memory address to check for cache hit
out	*dwData*	output variable to return stored data value

Return values:

true	on cache hit, dwData is set
false	on cache miss, dwData is not set

Definition at line 18 of file CacheManager.cpp.

bool CCacheManager::LoadCachePage (const void * *pAddress*)

Loads a contiguous block of memory, upto CACHE_BLOCK_SIZE, based on the address pointer passed.

Parameters:

in	*pAddress*	address of memory the actual page load is based on

Return values:

true	on success
false	on error

Definition at line 54 of file CacheManager.cpp.

CCacheSet Class Reference

```
#include <CacheSet.h>
```

Public Member Functions

- CCacheSet ()
  Default Constructor.
- bool GetCacheData (DWORD_PTR dwTag, DWORD_PTR cbOffset, DWORD_PTR &dwData)
- bool LoadCacheBlock (DWORD_PTR dwTag, const void *pAddress)

Detailed Description

Note:

The following policy will be followed:

The FIFO policy simply keeps track of the insertion order of the candidates and evicts the entry that has resided in the cache for the longest amount of time. The mechanism that implements this policy is straightforward, since the candidate eviction set (all blocks in a fully associative cache, or all blocks in a single set in a set-associative cache) can be managed as a circular queue. The circular queue has a single pointer to the oldest entry which is used to identify the eviction candidate and the pointer is incremented whenever a new entry is placed in the queue. This results in a single update for every miss in the cache.

However, the FIFO policy does not always match the temporal locality characteristics inherent in a program's reference stream, since some memory locations are accessed continually throughout the execution (e.g., commonly referenced global variables). Such references would experience frequent misses under a FIFO policy, since the blocks used to satisfy them would be evicted at regular intervals, as soon as every other block in the candidate eviction set had been evicted.

Definition at line 128 of file CacheSet.h.

Constructor & Destructor Documentation

CCacheSet::CCacheSet ()

Default Constructor.

Definition at line 13 of file CacheSet.cpp.

Member Function Documentation

bool CCacheSet::GetCacheData (DWORD_PTR *dwTag*, DWORD_PTR *cbOffset*, DWORD_PTR & *dwData*)

Attempts to retrieve data from cache memory based on tag and offset

Parameters:

in	*dwTag*	tag associated with the cache block
in	*cbOffset*	count of byte (cb) offset into cache block

out	*dwData*	output variable to return stored data value

Return values:

true	on cache hit, dwData is set
false	on cache miss, dwData is not set

Definition at line 20 of file CacheSet.cpp.

bool CCacheSet::LoadCacheBlock (DWORD_PTR *dwTag*, const void * *pAddress*)

Loads a contiguous block of memory of CACHE_BLOCK_SIZE, into cache block. It further establishes an association with the updated data via the dwTag.

Parameters:

in	*dwTag*	value containing Tag to associate with this cache block
in	*pAddress*	pointer to contiguous block of memory to load

Return values:

true	if successful
false	on error

Definition at line 57 of file CacheSet.cpp.

CVirtualAddress Class Reference

```
#include <VirtualAddress.h>
```

Public Member Functions

- CVirtualAddress (const void *pAddress)
  Initialization Constructor.
- DWORD_PTR DecodeTag (void) const
- DWORD_PTR DecodeOffset (void) const
- DWORD_PTR DecodeIndex (void) const
- const DWORD_PTR DecodeAddress (void) const
- std::ostream & operator<< (std::ostream &os) const

Detailed Description

This class is used to translate physical memory addresses into "virtual" addresses by decoding relevant bit patterns into corresponding information fields needed to reference cache memory. The original address is partitioned into three portions: the index bits are used to select a block; the block offset bits are used to select a word within a selected block, and the tag bits are used to do a tag match against the tag stored in the tag field of the selected entry.

Set-associative caches permit the flexible placement of data among all the entries of a "set". The index bits select a particular set, the tag bits select an entry (i.e. block) within the set, and the block offset bits select the word within the selected entry (i.e. block).

Definition at line 50 of file VirtualAddress.h.

Constructor & Destructor Documentation

CVirtualAddress::CVirtualAddress (const void * *pAddress*)

Initialization Constructor.

Definition at line 63 of file VirtualAddress.h.

Member Function Documentation

const DWORD_PTR CVirtualAddress::DecodeAddress (void) const

Definition at line 91 of file VirtualAddress.h.

DWORD_PTR CVirtualAddress::DecodeIndex (void) const

Decodes and returns Cache Set Index from the underlying memory address

Return values:

Index	on success
DECODE_ERROR	on error

Todo:

: cleanup these hard-coded values

Definition at line 51 of file VirtualAddress.cpp.

DWORD_PTR CVirtualAddress::DecodeOffset (void) const

Decodes and returns Block Offset from the underlying memory address

Return values:

Offset	on success
DECODE_ERROR	on failure

Todo:

: cleanup these hardcoded values

Definition at line 39 of file VirtualAddress.cpp.

DWORD_PTR CVirtualAddress::DecodeTag (void) const

Decodes and returns Tag from the underlying memory address

Return values:

Tag	on success
DECODE_ERROR	on failure

Definition at line 27 of file VirtualAddress.cpp.

std::ostream & CVirtualAddress::operator<< (std::ostream & *os*) const

Definition at line 65 of file VirtualAddress.cpp.

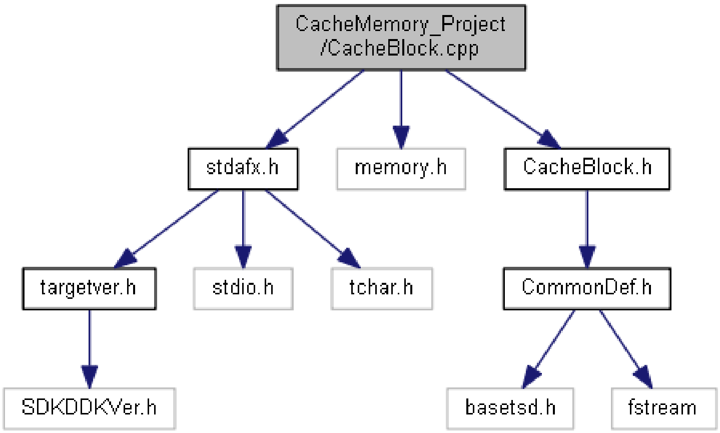
File Documentation

CacheMemory_Project/CacheBlock.cpp File Reference

CCacheBlock class implementation.

```
#include "stdafx.h"
#include <memory.h>
#include "CacheBlock.h"
```

Include dependency graph for CacheBlock.cpp:



Detailed Description

CCacheBlock class implementation.

Author:

Mark L. Short

Definition in file CacheBlock.cpp.

CacheBlock.cpp

```

#include "stdafx.h"
#include <memory.h>
#include "CacheBlock.h"


CCacheBlock::CCacheBlock ( )
    : m_dwTag (0)
{  // intentionally marking the memory block with fixed value
   // for testing and debugging purposes (yeah, like M$)
    memset (m_rgBlock, 'FE',  sizeof(m_rgBlock) );
};


bool CCacheBlock::GetCacheData (size_t cbOffset, DWORD_PTR& dwData) const
```

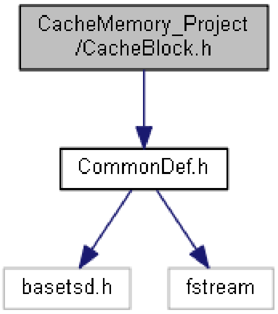
```
{
    bool bReturn = false;
    if ( cbOffset < ( sizeof (m_rgBlock) + sizeof (DWORD_PTR) - sizeof (BYTE)) )
    {
        dwData  = *reinterpret_cast<const DWORD_PTR*>(&m_rgBlock[cbOffset]);
        bReturn = true;
    }
    return bReturn;
}

bool CCacheBlock::LoadCacheBlock (DWORD_PTR dwTag, const BYTE* pData,
                                  size_t cbLen /* = g_CACHE_BLOCK_SIZE */)
{
    bool bReturn = false;
    if ( pData != nullptr )
    {
        if ( cbLen <= sizeof(m_rgBlock) )
        {
            memcpy (m_rgBlock, pData, cbLen);
            m_dwTag = dwTag;
            bReturn = true;
        }
    }
    return bReturn;
}
```

CacheMemory_Project/CacheBlock.h File Reference

CCacheBlock class interface.

`#include "CommonDef.h"`

Include dependency graph for CacheBlock.h:



Classes

- class CCacheBlock

Variables

- const int g_CACHE_BLOCK_SIZE = 32
  size, in bytes, of each cache block
- const int g_CACHE_NUM_BLOCKS = 16
  number of cache data blocks

Detailed Description

CCacheBlock class interface.

Author:

Mark L. Short

Definition in file CacheBlock.h.

Variable Documentation

const int g_CACHE_BLOCK_SIZE = 32

size, in bytes, of each cache block

Block size (sometimes referred to as line size) describes the granularity at which the cache operates. Each block is a contiguous series of bytes in memory and begins on a naturally aligned boundary.

For example, in a cache with 16 - byte blocks, each block would contain 16 bytes, and the first byte in each block would be aligned to 16 - byte boundaries in the address space, implying that the low - order 4 bits of the address of the first byte would always be zero(i.e., 0b ... 0000). The smallest usable block size is the natural word size of the processor (i.e., 4 bytes for a 32 - bit machine, or 8 bytes for a 64 - bit machine), since each access will require the cache to supply at least that many bytes, and splitting a single access over multiple blocks would introduce unacceptable overhead into the access path.

Whenever the block size is greater than 1 byte, the low-order bits of an address must be used to find the byte or word being accessed within the block. As stated above, the low-order bits for the first byte in the block must always be zero, corresponding to a naturally aligned block in memory. However, if a byte other than the first byte needs to be accessed, the low-order bits must be used as a block offset to index into the block to find the right byte.

The number of bits needed for the block offset is the log2 of the block size, so that enough bits are available to span all the bytes in the block. For example, if the block size is 64 bytes, log2(64) = 6 low-order bits are used as the block offset. The remaining higher-order bits are then used to locate the appropriate block in the cache memory.

Definition at line 42 of file CacheBlock.h.

const int g_CACHE_NUM_BLOCKS = 16

number of cache data blocks

Definition at line 43 of file CacheBlock.h.

CacheBlock.h

```
 1
 9 #if !defined(_COMMON_DEF_H__)
10     #include "CommonDef.h"
11 #endif
12
42 const int g_CACHE_BLOCK_SIZE      = 32;
```

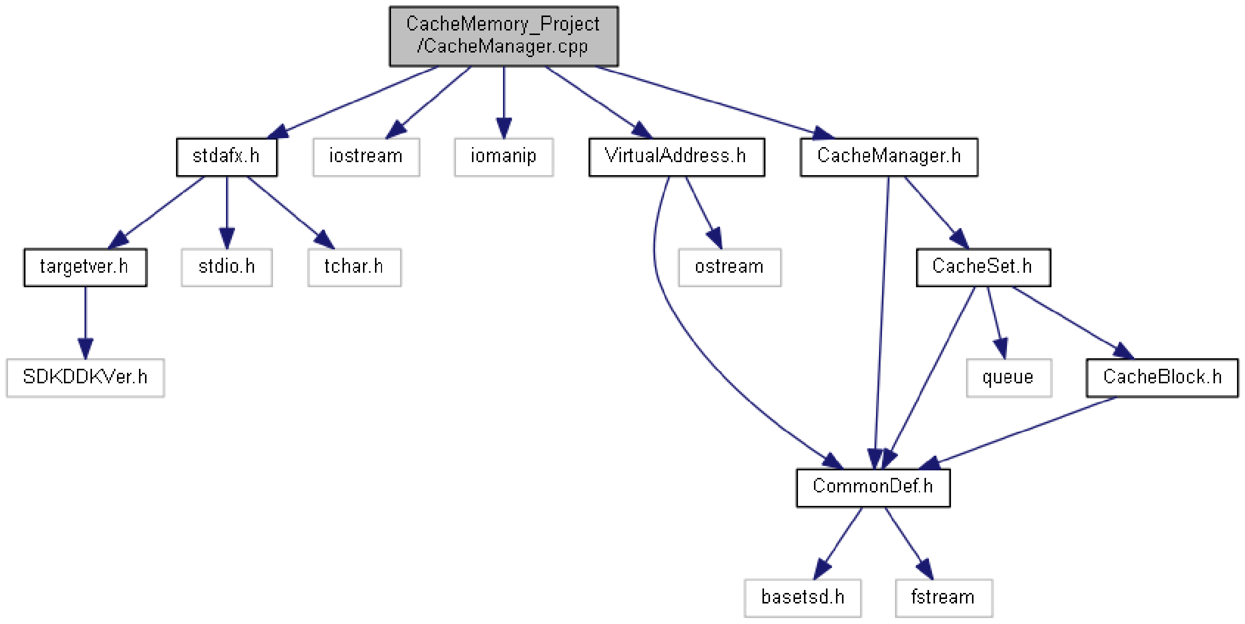
```
const int g_CACHE_NUM_BLOCKS     = 16;

class CCacheBlock
{
    DWORD_PTR     m_dwTag;
    BYTE          m_rgBlock[g_CACHE_BLOCK_SIZE];

public:
    CCacheBlock ( );

    ~CCacheBlock ( )
    { };

    void set_Tag (DWORD_PTR dwSet) throw()
    { m_dwTag = dwSet; };

    DWORD_PTR get_Tag (void) const throw()
    { return m_dwTag; };

    bool GetCacheData   (size_t cbOffset, DWORD_PTR& dwData) const;

    bool LoadCacheBlock (DWORD_PTR dwTag, const BYTE* pData,
                         size_t cbLen = g_CACHE_BLOCK_SIZE);

private:
    CCacheBlock (const CCacheBlock& rhs)
    { };

    CCacheBlock& operator = (const CCacheBlock& rhs)
    { };

};

```

CacheMemory_Project/CacheManager.cpp File Reference

CCacheManager class implementation.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "VirtualAddress.h"
#include "CacheManager.h"
```

Include dependency graph for CacheManager.cpp:



Detailed Description

CCacheManager class implementation.

Author:

Mark L. Short

Definition in file CacheManager.cpp.

CacheManager.cpp

```

#include "stdafx.h"

#include <iostream>
#include <iomanip>

#include "VirtualAddress.h"
#include "CacheManager.h"



bool CCacheManager::GetCacheData (const void* pAddress, DWORD_PTR& dwData)
{
    bool bReturn = false;
    // we need to decode pAddress and see if it maps to what we have in cache
    if ( pAddress )
    {
        CVirtualAddress vAddress (pAddress);
        DWORD_PTR dwIndex = vAddress.DecodeIndex ( );

        if ( (dwIndex < _countof(m_rgCacheSets) ) && (dwIndex != DECODE_ERROR) )
        {
            DWORD_PTR dwTag    = vAddress.DecodeTag ( );
            DWORD_PTR dwOffset = vAddress.DecodeOffset ( );
#ifdef _DEBUG
```

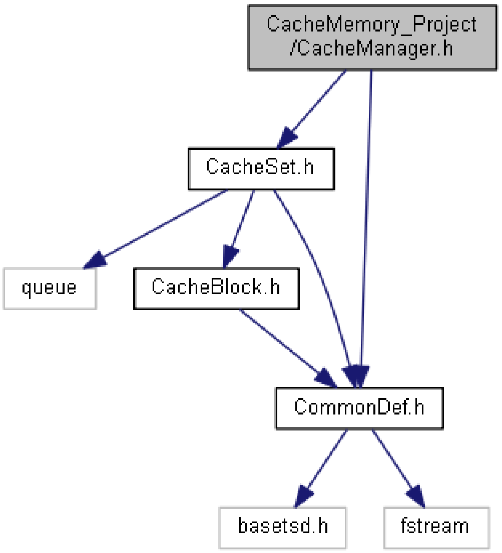
```
            std::cout << "  Checking Cache Set [" << dwIndex  << "] "
                      << "for Tag ["              << dwTag    << "] "
                      << "Offset ["               << dwOffset << "]" << std::endl;
#endif
/*
   On each lookup, we must read the tag and compare it with the address bits of
   the reference being performed to determine whether a hit or miss has occurred.

   A compromise between the indexed memory and the associative memory is the
   set-associative memory which uses both indexing and associative search; An
   address is used to index into one of the sets, while the multiple entries
   within a set are searched with a key to identify one particular entry. This
   compromise provides some flexibility in the placement of data without
   incurring the complexity of a fully associative memory.
*/
            bReturn = m_rgCacheSets[dwIndex].GetCacheData (dwTag, dwOffset, dwData);
        }
    }

    return bReturn;
}

bool CCacheManager::LoadCachePage (const void* pAddress)
{
    bool bReturn = false;

    if ( pAddress )
    {
        CVirtualAddress vAddress (pAddress);
        DWORD_PTR dwIndex = vAddress.DecodeIndex ( );
        if ( (dwIndex < _countof(m_rgCacheSets) ) && (dwIndex != DECODE_ERROR) )
        {
            bReturn = m_rgCacheSets[dwIndex].LoadCacheBlock(vAddress.DecodeTag( ),
                                                            pAddress);
        }
    }
    return bReturn;
}


```

CacheMemory_Project/CacheManager.h File Reference

CCacheManager class interface.

```
#include "CacheSet.h"
#include "CommonDef.h"
```

Include dependency graph for CacheManager.h:



Classes

- class CCacheManager

Detailed Description

CCacheManager class interface.

Author:

Mark L. Short

Definition in file CacheManager.h.

CacheManager.h

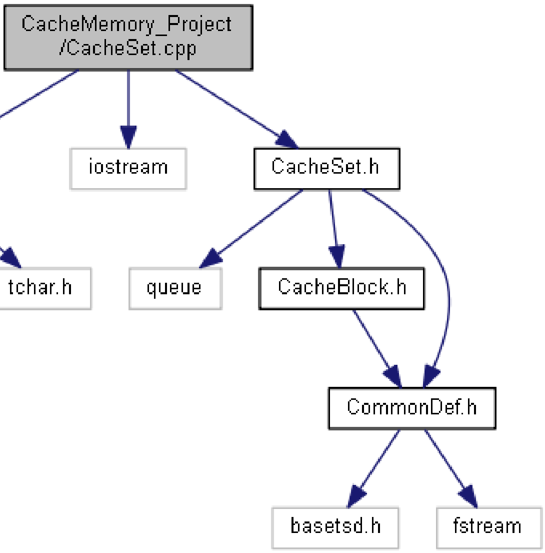
```

#if !defined(_CACHE_MANAGER_H__)
#define _CACHE_MANAGER_H__

#if !defined(_COMMON_DEF_H__)
    #include "CommonDef.h"
#endif

#if !defined(_CACHE_SET_H__)
    #include "CacheSet.h"
#endif


class CCacheManager
{
    CCacheSet m_rgCacheSets[g_CACHE_SETS];

public:

    CCacheManager ( )
    { };

    bool GetCacheData  (const void* pAddress, DWORD_PTR& dwData);

    bool LoadCachePage (const void* pAddress);

};



#endif
```

CacheMemory_Project/CacheSet.cpp File Reference

CCacheSet class implementation.

```
#include "stdafx.h"
#include <iostream>
#include "CacheSet.h"
```

Include dependency graph for CacheSet.cpp:



Detailed Description

CCacheSet class implementation.

Author:

Mark L. Short

Definition in file CacheSet.cpp.

CacheSet.cpp

```

#include "stdafx.h"
#include <iostream>
#include "CacheSet.h"
```

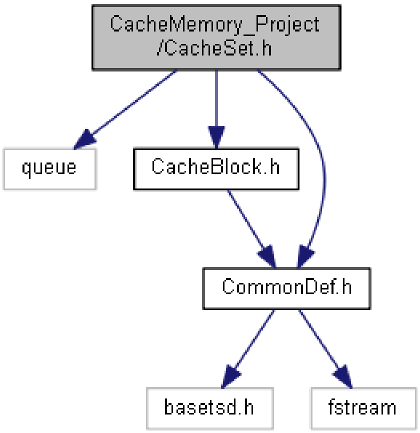
```

CCacheSet::CCacheSet ( )
    : m_queAvailableBlocks ( )
{
    for ( int i = 0; i < _countof(m_rgCacheBlock); i++ )
        m_queAvailableBlocks.push (&m_rgCacheBlock[i]);
};

bool CCacheSet::GetCacheData (DWORD_PTR dwTag, DWORD_PTR dwOffset, DWORD_PTR& dwData)
{
    bool bReturn = false;
    // lets iterate through our cache blocks and see if any matches 'dwTag'
    // Actually, this should be done in multiple threads simultaneously
    bool bFound = false;
    int i;
    for ( i = 0; i < _countof(m_rgCacheBlock); i++ )
    {
        DWORD_PTR dwCacheTag = m_rgCacheBlock[i].get_Tag ( );

#ifdef _DEBUG
        std::cout << "    Checking Cache Block [" << i << "] "
                  << "Cache Tag ["            << dwCacheTag << "]" << std::endl;
#endif
        if ( dwTag == dwCacheTag )
        {
            bFound = true;
            break;
        }
    }

    if ( bFound )
    {
        bReturn = m_rgCacheBlock[i].GetCacheData (dwOffset, dwData);
#ifdef _DEBUG
        std::cout << "    ** Cache Hit ** ";
        if (bReturn)
            std::cout << "Data returned [" << dwData << "]" << std::endl;
        else
            std::cout << "Error retrieving data!" << std::endl;
#endif
    }

    return bReturn;
}

bool CCacheSet::LoadCacheBlock (DWORD_PTR dwTag, const void* pAddress)
{
    bool bReturn = false;
    // lets find a stale CacheBlock to load
    CCacheBlock* pCacheBlock = m_queAvailableBlocks.front ( );
    m_queAvailableBlocks.pop ( );
/*
     In order to keep everything matching up correctly with our Tag association,
     we need to load memory addresses that would have the same tag and index
     fields when subsequently broken down.

     to do this, i am going clear the Offset bits (5 bits) of the pAddress parameter
     to generate an address that points to memory that is properly aligned to match
     up with our tag + index field associations
*/
    DWORD_PTR dwAdjustedAddress = (reinterpret_cast<DWORD_PTR>(pAddress) & ~(0x001F));

    bReturn = pCacheBlock->LoadCacheBlock(dwTag,
                                 reinterpret_cast<const BYTE*>(dwAdjustedAddress));

    m_queAvailableBlocks.push (pCacheBlock);

    return bReturn;
}

```

CacheMemory_Project/CacheSet.h File Reference

CCacheSet class interface.

```
#include <queue>
#include "CacheBlock.h"
#include "CommonDef.h"
```

Include dependency graph for CacheSet.h:



Classes

- class CCacheSet

Variables

- const int g_CACHE_SETS = g_CACHE_NUM_BLOCKS / 4
  Number of caches sets needed.
- const int g_CACHE_BLOCKS_PER_SET = g_CACHE_NUM_BLOCKS / g_CACHE_SETS

Detailed Description

CCacheSet class interface.

Author:

Mark L. Short

Definition in file CacheSet.h.

Variable Documentation

const int g_CACHE_BLOCKS_PER_SET = g_CACHE_NUM_BLOCKS / g_CACHE_SETS

Definition at line 26 of file CacheSet.h.

const int g_CACHE_SETS = g_CACHE_NUM_BLOCKS / 4

Number of caches sets needed.

Definition at line 25 of file CacheSet.h.

CacheSet.h

```
  1 
  9 #if !defined(_CACHE_SET_H__)
 10 #define _CACHE_SET_H__
 11 
 12 #if !defined(_COMMON_DEF_H__)
 13     #include "CommonDef.h"
 14 #endif
 15 
 16 #ifndef _QUEUE_
 17     #include <queue>
 18 #endif
 19 
 20 #if !defined(_CACHE_BLOCK_H__)
 21     #include "CacheBlock.h"
 22 #endif
 23 
 25 const int g_CACHE_SETS           = g_CACHE_NUM_BLOCKS / 4;
 26 const int g_CACHE_BLOCKS_PER_SET = g_CACHE_NUM_BLOCKS / g_CACHE_SETS;
 27 
 28 /*
 29     The simplest approach, direct-mapped, forces a many-to-one mapping between
 30     addresses and the available storage locations in the cache. In other words,
 31     a particular address can reside only in a single location in the cache; that
 32     location is usually determined by extracting n bits from the address and using
 33     those n bits as a direct index into one of 2n possible locations in the cache.
 34 
 35     Of course, since there is a many-to-one mapping, each location must also store
 36     a tag that contains the remaining address bits corresponding to the block of
 37     data stored at that location. On each lookup, the hardware must read the tag
 38     and compare it with the address bits of the reference being performed to
 39     determine whether a hit or miss has occurred.
 40 
 41     In the degenerate case where a direct-mapped memory contains enough storage
 42     locations for every address block (i.e., the n index bits include all bits of
 43     the address), no tag is needed, as the mapping between addresses and storage
 44     locations is now one-to-one instead of many-to-one. The register file inside
 45     the processor is an example of such a memory; it need not be tagged since all
 46     the address bits (all bits of the register identifier) are used as the index
 47     into the register file.
 48 
 49     Set-associative, is a many-to-few mapping between addresses and storage locations.
 50     On each lookup, a subset of address bits is used to generate an index, just
 51     as in the direct-mapped case. However, this index now corresponds to a set
 52     of entries, usually two to eight, that are searched in parallel for a matching
 53     tag. In practice, this approach is much more efficient from a hardware
 54     implementation perspective, since it requires fewer address comparators than
 55     a fully associative cache, but due to its flexible mapping policy behaves
 56     similarly to a fully associative cache.
 57 
 58     Set-associative caches permit the flexible placement of data among all the entries
 59     of a set.
 60 
 61     - index bits            - select a particular set,
 62     - tag bits              - select an entry within the set,
 63     - block offset bits     - select the word within the selected entry.
```

```

*/
// n-way set associative
// - Each set contains n entries
// - Block number determines which set
//      - (Block number) mod (Sets in cache)
// - Search all entries in a given set at once
//
// Four-way set-associative data cache
//
// Blocks     = 16;
// BlockSize  = 32 (Bytes);
//
// Sets (s)   = Blocks / 4;
//         s  = 16 / 4;
//         s  = 4;
// Offset (o) - Select the word within each block
//         o  = lg (BlockSize)
//         o  = lg (32)
//         o  = 5;
// Index (i)  = Select set of blocks
//         i  = lg (Number of Sets)
//         i  = lg (4)
//         i  = 2
// Tag (t)    = ID blocks within a set
//         t  = 32 - o - i;
//         t  = 32 - 5 - 2;
//         t  = 25;
//
//           32 bit Address
//     Tag       (set) Index   (blocK) Offset
// |   bits[t]  |   bits[i]    |   bits[o]    |
// |    25      |     2         |      5       |
// |   0..24    |   25..26     |   27..31     |
//
//           64 bit Address
//     Tag       (set) Index   (block) Offset
// |   bits[t]  |   bits[i]    |   bits[o]    |
// |    57      |     2         |      5       |
// |   0..56    |   57..58     |   59..63     |


class CCacheSet
{
    CCacheBlock              m_rgCacheBlock[g_CACHE_BLOCKS_PER_SET];
    std::queue<CCacheBlock*> m_queAvailableBlocks;

public:
    CCacheSet ( );
    bool GetCacheData (DWORD_PTR dwTag, DWORD_PTR cbOffset, DWORD_PTR& dwData);

    bool LoadCacheBlock (DWORD_PTR dwTag, const void* pAddress);

private:

    CCacheSet(const CCacheSet& rhs) { };
    CCacheSet& operator=(const CCacheSet& rhs) { };
};

#endif
```

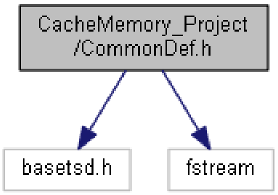
CacheMemory_Project/CommonDef.h File Reference

Common type definitions.

`#include <basetsd.h>`

```
#include <fstream>
```

Include dependency graph for CommonDef.h:



Macros

- #define _COMMON_DEF_H__

Typedefs

- typedef unsigned __int8 BYTE
- typedef unsigned __int32 DWORD

Variables

- std::ofstream oflog

Detailed Description

Common type definitions.

Author:

Mark L. Short

Definition in file CommonDef.h.

Macro Definition Documentation

#define _COMMON_DEF_H__

Definition at line 11 of file CommonDef.h.

Typedef Documentation

typedef unsigned __int8 BYTE

Definition at line 17 of file CommonDef.h.

typedef unsigned __int32 DWORD

Definition at line 18 of file CommonDef.h.

Variable Documentation

std::ofstream oflog

CommonDef.h

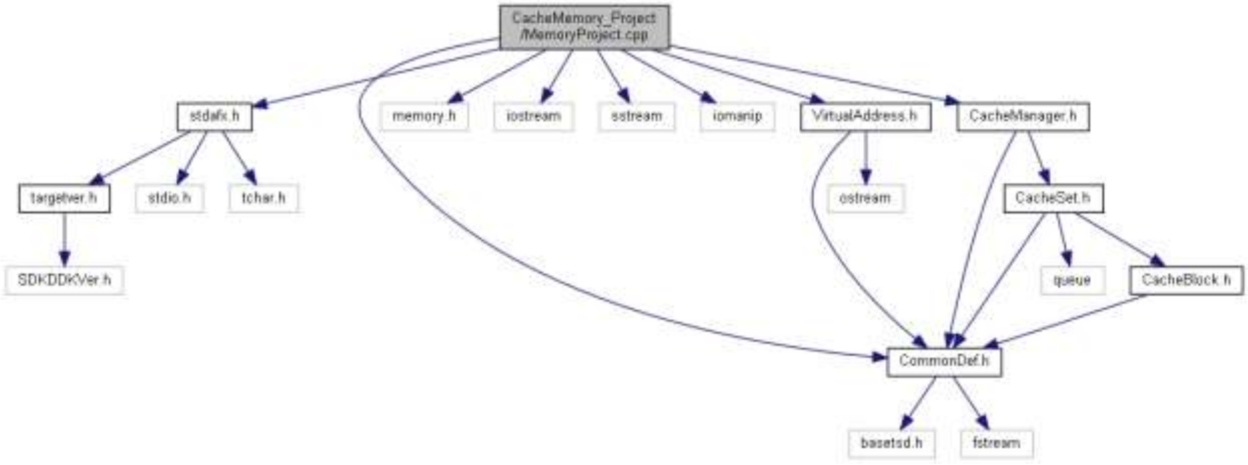
```

#pragma once

#if !defined(_COMMON_DEF_H__)
#define _COMMON_DEF_H__

#ifndef _BASETSD_H_
    #include <basetsd.h>
#endif

typedef unsigned __int8  BYTE;
typedef unsigned __int32 DWORD;

#ifndef _FSTREAM_
    #include <fstream>
#endif

extern std::ofstream oflog;


#endif
```

CacheMemory_Project/MemoryProject.cpp File Reference

Main source file for implementation of data cache simulation.

```
#include "stdafx.h"
#include "CommonDef.h"
#include <memory.h>
#include <iostream>
#include <sstream>
#include <iomanip>
#include "VirtualAddress.h"
#include "CacheManager.h"
```

Include dependency graph for MemoryProject.cpp:



Functions

- __declspec (align(32)) int g_rgA[g_MAX_ARRAY_SIZE] = { 0 }
- std::ostream & PrintIterationHeader (std::ostream &os, int iIteration)
- int _tmain (int argc, _TCHAR *argv[])

Variables

- const int g_MAX_ARRAY_SIZE = 512
- const int g_DR_PASSOS_LOOP = 511

Detailed Description

Main source file for implementation of data cache simulation.

Definition in file MemoryProject.cpp.

Function Documentation

__declspec (align(32)) = { 0 }

int _tmain (int *argc*, _TCHAR * *argv*[])

Definition at line 124 of file MemoryProject.cpp.

std::ostream& PrintIterationHeader (std::ostream & *os*, int *iIteration*)

Definition at line 115 of file MemoryProject.cpp.

Variable Documentation

const int g_DR_PASSOS_LOOP = 511

Definition at line 104 of file MemoryProject.cpp.

const int g_MAX_ARRAY_SIZE = 512

Definition at line 103 of file MemoryProject.cpp.

MemoryProject.cpp

```
  1
 91 #include "stdafx.h"
 92 #include "CommonDef.h"
 93
 94 #include <memory.h>
 95 #include <iostream>
 96 #include <sstream>
 97 #include <iomanip>
 98
 99 #include "VirtualAddress.h"
100 #include "CacheManager.h"
101
102
103 const int g_MAX_ARRAY_SIZE = 512;
104 const int g_DR_PASSOS_LOOP = 511;
105
106
107 __declspec(align(32)) int g_rgA[g_MAX_ARRAY_SIZE] = { 0 };
108 __declspec(align(32)) int g_rgB[g_MAX_ARRAY_SIZE] = { 0 };
109 __declspec(align(32)) int g_rgC[g_MAX_ARRAY_SIZE] = { 0 };
110 #ifdef _DEBUG
111 __declspec(align(32)) int g_rgD[g_MAX_ARRAY_SIZE] = { -1 };  // for debugging purposes
112 #endif
113
114
115 std::ostream& PrintIterationHeader(std::ostream& os, int iIteration)
116 {
117     os  << std::dec << std::endl;
118     os  << "Cache Miss(es) in Iteration[" << iIteration + 1 << "]"  << std::endl;
119     os  << "-----------------------------------------------------"  << std::endl;
120
121     return os;
122 }
123
124 int _tmain (int argc, _TCHAR* argv[])
125 {
126     std::ofstream oflog;
127     std::stringstream ss;
128
129     // Let's build our output filename based on the memory address
130     // we get for our 1st global variable that we use in our cache
131     // simulation.  The only uniqueness in the output is going to be
132     // based off of that memory address, so we might as well keep
133     // the data around for testing and comparison purposes.
134
135     ss << "..\\Data\\CacheMisses_" << std::hex << std::setw(2 * sizeof(DWORD_PTR) )
136        << std::setfill('0')
137        << reinterpret_cast<DWORD_PTR>(&g_rgA[0]) << ".txt";
138
139     oflog.open(ss.str().c_str());
```

```
    // seeding the global data arrays with some data that we
    // can potentially use to verify if our cache is storing
    // and retrieving correct values
    for ( int i = 0; i < g_MAX_ARRAY_SIZE; i++)
        g_rgA[i] = i + 0x1100;

    for ( int i = 0; i < g_MAX_ARRAY_SIZE; i++ )
        g_rgB[i] = i + 0x2200;

    for ( int i = 0; i < g_MAX_ARRAY_SIZE; i++ )
        g_rgC[i] = i + 0x3300;

    CCacheManager cacheManager;

    // let's keep track of some cache statistics
    int iCacheMisses = 0;
    int iCacheHits   = 0;
    int iCacheErrors = 0;

    // also need some local variables to store data
    int iA;
    int iB;
    int iB1;
    int iC;
//
// The following is the benchmark code from Assignment #2
//
//    A[i] = A[i] + B[i] + B[i + 1] * C[i]
//
//
#if defined(_WIN64)
    oflog << "Executing an x64 build" << std::endl;
#else
    oflog << "Executing an x32 build" << std::endl;
#endif

    oflog << "Following based on the physical address of A[0] being 0x"
          << std::hex << std::setw(2*sizeof(DWORD_PTR)) << std::setfill('0')
          << reinterpret_cast<DWORD_PTR>(&g_rgA[0]) << std::endl;
    oflog << "================================================================="
          << std::endl;

    for ( int i = 0; i < g_DR_PASSOS_LOOP; i++ )
    {
        DWORD_PTR dataFromCache;
        bool bCacheMissThisIteration = false;

// Attempting to access 1st operand 'B[i + 1]'

        if ( cacheManager.GetCacheData ( &g_rgB[i + 1], dataFromCache) == false )
        {
            iCacheMisses++;
            cacheManager.LoadCachePage ( &g_rgB[i + 1] );
            iB1 = g_rgB[i + 1]; // cache-miss, load the data directly

#ifdef _DEBUG

            CVirtualAddress va ( &g_rgB[i + 1] );

            if (bCacheMissThisIteration == false)
            { // then lets print the iteration header
                bCacheMissThisIteration = true;
                PrintIterationHeader(oflog, i);
            }

            oflog << std::dec
                  << "   Cache Miss["    << iCacheMisses << "] "
                  << "for 'B[" << i << " + 1]'" << std::endl;

            oflog << "   " << va << std::endl;
```

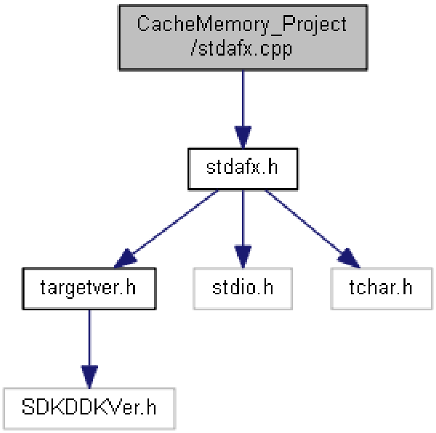
```
#endif

        }
        else
        {
            iCacheHits++;
            iB1 = dataFromCache; // cache-hit, use the data retrieved from cache
#ifdef _DEBUG
        // ok, let's verify if our cache actually stored and retrieved the correct
        // data values
            if (iB1 != g_rgB[i + 1] )
            {
                std::cout << "B[i+1] Data cache inconsistency detected"
                          << std::endl;
                iCacheErrors++;
            }
#endif
        }

// Attempting to access 2nd operand 'C[i]'

        if ( cacheManager.GetCacheData ( &g_rgC[i], dataFromCache) == false )
        {
            iCacheMisses++;
            cacheManager.LoadCachePage ( &g_rgC[i] );
            iC = g_rgC[i]; // cache-miss, load the data directly

#ifdef _DEBUG
            CVirtualAddress va (&g_rgC[i]);

            if ( bCacheMissThisIteration == false )
            { // then lets print the iteration header
                bCacheMissThisIteration = true;
                PrintIterationHeader (oflog, i);
            }

            oflog << std::dec
                  << "   Cache Miss[" << iCacheMisses << "] "
                  << "for 'C[" << i << "]'" << std::endl;

            oflog << "   " << va << std::endl;
#endif

        }
        else
        {
            iCacheHits++;
            iC = dataFromCache; // cache-hit, use the data retrieved from cache
#ifdef _DEBUG
            if ( iC != g_rgC[i] ) // now let's verify the retrieved cache data
            {
                std::cout << "C[i] Data cache inconsistency detected"
                          << std::endl;
                iCacheErrors++;
            }
#endif
        }

// Attempting to access 3rd operand 'A[i]'

        if ( cacheManager.GetCacheData (&g_rgA[i], dataFromCache) == false )
        {
            iCacheMisses++;
            cacheManager.LoadCachePage (&g_rgA[i]);
            iA = g_rgA[i]; // cache-miss, load the data directly

#ifdef _DEBUG
            CVirtualAddress va (&g_rgA[i]);

            if ( bCacheMissThisIteration == false )
            { // then lets print the iteration header
```

```
                bCacheMissThisIteration = true;
                PrintIterationHeader (oflog, i);
            }

            oflog << std::dec
                  << "   Cache Miss[" << iCacheMisses << "] "
                  << "for 'A["          << i << "]'" << std::endl;

            oflog << "   " << va << std::endl;
#endif
        }
        else
        {
            iCacheHits++;
            iA = dataFromCache; // cache-hit, use the data retrieved from cache

#ifdef _DEBUG
            if ( iA != g_rgA[i] ) // now let's verify the retrieved cache data
            {
                std::cout << "A[i] Data cache inconsistency detected"
                          << std::endl;
                iCacheErrors++;
            }
#endif
        }

// Attempting to access 4th operand 'B[i]'

        if ( cacheManager.GetCacheData ( &g_rgB[i], dataFromCache) == false )
        {
            iCacheMisses++;
            cacheManager.LoadCachePage (&g_rgB[i]);
            iB = g_rgB[i]; // cache-miss, load the data directly

#ifdef _DEBUG
            CVirtualAddress va(&g_rgB[i]);

            if ( bCacheMissThisIteration == false )
            {  // then lets print the iteration header
                bCacheMissThisIteration = true;
                PrintIterationHeader(oflog, i);
            }

            oflog << std::dec
                  << "   Cache Miss[" << iCacheMisses << "] "
                  << "for 'B["          << i << "]'" << std::endl;
            oflog << "   " << va << std::endl;
#endif
        }
        else
        {
            iCacheHits++;
            iB = dataFromCache; // cache-hit, use the data retrieved from cache
#ifdef _DEBUG
            if ( iB != g_rgB[i] ) // now let's verify the retrieved cache data
            {
                std::cout << "B[i] Data cache inconsistency detected"
                          << std::endl;
                iCacheErrors++;
            }
#endif
        }

        // parenthesis used to denote explicit operation
        int iResult = iA + iB + (iB1 * iC);

        g_rgA[i] = iResult;

        std::cout << "Iteration[ i=" << i << " ]" << std::endl;
        std::cout << "A[i] + B[i] + B[i + 1] * C[i] Computation Result:"
                  << iResult
                  << std::endl;
        std::cout << "-------------------------------------------------------------"
                  << std::endl;
        std::cout << std::dec;
        std::cout << "Cache Misses:" << iCacheMisses  << std::endl;
        std::cout << "Cache Hits:  " << iCacheHits    << std::endl;
        std::cout << "Cache Errors:" << iCacheErrors  << std::endl;

    }

    oflog << std::dec;
    oflog << "----------------------------------------" << std::endl;
    oflog << "Cache Misses:" << iCacheMisses << std::endl;
    oflog << "Cache Hits:  " << iCacheHits   << std::endl;
    oflog << "Cache Errors:" << iCacheErrors << std::endl;

    oflog.close();

    std::cout << std::endl;
    std::cout << "[Enter 'q' to exit program]" << std::endl;

    char c;
    std::cin >> c;

    return 0;
}

```

CacheMemory_Project/stdafx.cpp File Reference

`#include "stdafx.h"`

Include dependency graph for stdafx.cpp:



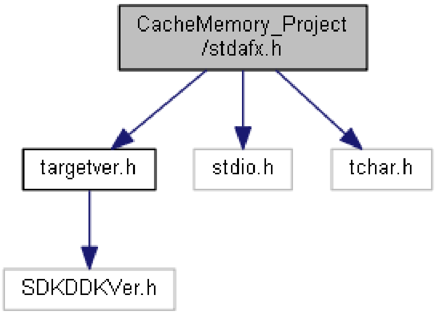
stdafx.cpp

```
// stdafx.cpp : source file that includes just the standard includes
// CacheMemory_Project.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

```

CacheMemory_Project/stdafx.h File Reference

```
#include "targetver.h"
#include <stdio.h>
#include <tchar.h>
```

Include dependency graph for stdafx.h:



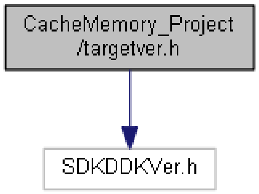
stdafx.h

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
```

CacheMemory_Project/targetver.h File Reference

```
#include <SDKDDKVer.h>
```

Include dependency graph for targetver.h:



targetver.h

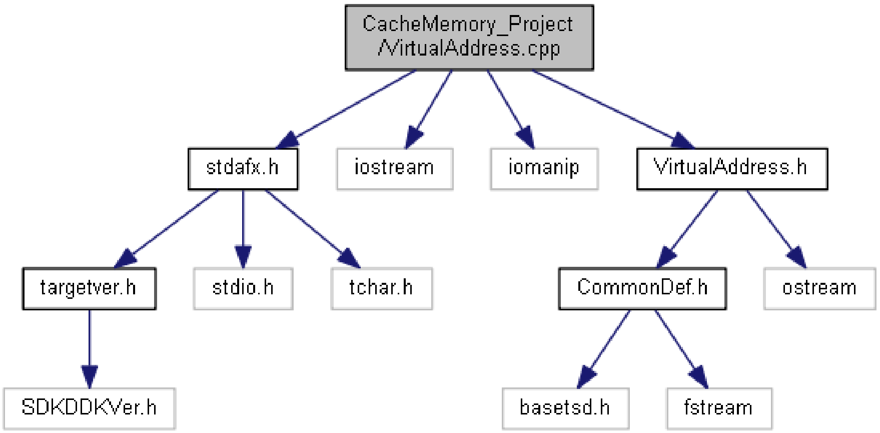
```
#pragma once

// Including SDKDDKVer.h defines the highest available Windows platform.

#include <SDKDDKVer.h>
```

CacheMemory_Project/VirtualAddress.cpp File Reference

CVirtualAddress class implementation.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "VirtualAddress.h"
```

Include dependency graph for VirtualAddress.cpp:



Functions

- std::ostream & operator<< (std::ostream &os, const CVirtualAddress &va)

Detailed Description

CVirtualAddress class implementation.

Author:

Mark L. Short

Definition in file VirtualAddress.cpp.

Function Documentation

std::ostream& operator<< (std::ostream & *os*, const CVirtualAddress & *va*)

Definition at line 78 of file VirtualAddress.cpp.

VirtualAddress.cpp

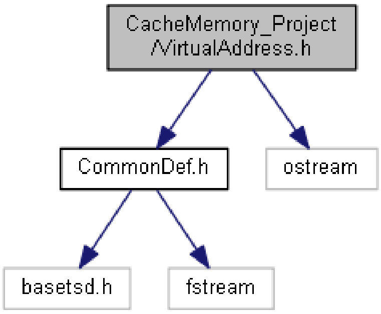
```

#include "stdafx.h"

#include <iostream>
#include <iomanip>

#include "VirtualAddress.h"

//           32 bit Address
//     Tag       (set) Index   (blocK) Offset
// |   bits[t]  |   bits[i]    |   bits[o]    |
// |    25      |     2        |      5       |
// |   0..24    |   25..26     |   27..31     |

//           64 bit Address
//     Tag       (set) Index   (blocK) Offset
// |   bits[t]  |   bits[i]    |   bits[o]    |
// |    57      |     2        |      5       |
// |   0..56    |   57..58     |   59..63     |

DWORD_PTR CVirtualAddress::DecodeTag (void) const
{
    if ( m_pAddress )
    {
        DWORD_PTR dwReturn = reinterpret_cast<DWORD_PTR>(m_pAddress);

        return (dwReturn >> (INDEX_BITS + OFFSET_BITS));
    }
    else
        return DECODE_ERROR;
}

DWORD_PTR CVirtualAddress::DecodeOffset (void) const
{
    if ( m_pAddress )
    {
        DWORD_PTR dwReturn = reinterpret_cast<DWORD_PTR>(m_pAddress);

        return (dwReturn & 0x001F);
    }
    else
        return DECODE_ERROR;
}

DWORD_PTR CVirtualAddress::DecodeIndex (void) const
{
    if ( m_pAddress )
    {
        DWORD_PTR dwReturn = reinterpret_cast<DWORD_PTR>(m_pAddress);

        dwReturn = dwReturn >> OFFSET_BITS;

        return (dwReturn & 0x0003);
    }
    else
        return DECODE_ERROR;
}

std::ostream& CVirtualAddress::operator << (std::ostream& os) const
{
    os  << "Address[0x"  << std::hex << std::setw (2 * sizeof (DWORD_PTR))
        << std::setfill ('0') << DecodeAddress () << "] "
        << std::dec
        << "Tag["        << DecodeTag ( )       << "] "
        << "Index["      << DecodeIndex ( )     << "] "
        << "Offset["     << DecodeOffset ( )    << "] ";

    return os;

};

std::ostream& operator<< (std::ostream& os, const CVirtualAddress& va)
{
    return va.operator<< (os);
}
```

CacheMemory_Project/VirtualAddress.h File Reference

CVirtualAddress class interface.

```
#include "CommonDef.h"
#include <ostream>
```

Include dependency graph for VirtualAddress.h:



Classes

- class CVirtualAddress

Functions

- std::ostream & operator<< (std::ostream &os, const CVirtualAddress &va)

Variables

- const DWORD_PTR DECODE_ERROR = (DWORD_PTR) -1
  Error Code returned on decoding failure.

Detailed Description

CVirtualAddress class interface.

Author:

Mark L. Short

Definition in file VirtualAddress.h.

Function Documentation

std::ostream& operator<< (std::ostream & *os*, const CVirtualAddress & *va*)

Definition at line 78 of file VirtualAddress.cpp.

Variable Documentation

const DWORD_PTR DECODE_ERROR = (DWORD_PTR) -1

Error Code returned on decoding failure.

Definition at line 33 of file VirtualAddress.h.

VirtualAddress.h

```
 1
 9 #if !defined(_VIRTUAL_ADDRESS_H__)
10 #define _VIRTUAL_ADDRESS_H__
11
12 #if !defined(_COMMON_DEF_H__)
13     #include "CommonDef.h"
14 #endif
15
16 #ifndef _OSTREAM_
17     #include <ostream>
18 #endif
19
20 //           32 bit Address
21 //     Tag       (set) Index   (block) Offset
22 // |   bits[t]  |   bits[i]    |   bits[o]    |
23 // |    25      |     2        |      5       |
24 // |   0..24    |   25..26     |   27..31     |
25
26 //           64 bit Address
27 //     Tag       (set) Index   (block) Offset
28 // |   bits[t]  |   bits[i]    |   bits[o]    |
29 // |    57      |     2        |      5       |
30 // |   0..56    |   57..58     |   59..63     |
31
33 const DWORD_PTR DECODE_ERROR = (DWORD_PTR) -1;
34
50 class CVirtualAddress
51 {
53     const size_t   OFFSET_BITS = 5;
55     const size_t   INDEX_BITS  = 2;
56     const size_t   TAG_BITS    = sizeof (DWORD_PTR) - INDEX_BITS - OFFSET_BITS;
57
58     const void* m_pAddress;
59
60 public:
61
63     CVirtualAddress (const void* pAddress)
64         : m_pAddress (pAddress)
65     { };
66
73     DWORD_PTR DecodeTag       (void) const;
74
81     DWORD_PTR DecodeOffset    (void) const;
82
```

```
    DWORD_PTR DecodeIndex     (void) const;

    const DWORD_PTR DecodeAddress    (void) const
    { return reinterpret_cast<const DWORD_PTR>(m_pAddress); };

    std::ostream& operator << (std::ostream& os) const;

private:
    // We really do not want this class to be instantiated in this manner,
    // so going to make private

    CVirtualAddress ( )
        : m_pAddress (nullptr)
    { };

};

std::ostream& operator<< (std::ostream& os, const CVirtualAddress& va);

#endif
```

Index