Pipeline Project

Advanced Computer Architecture
CMPS 5133

Mark L. Short,

Kessia Eugene,

Kishore Lav Mutyala

Version 1
11/25/2014 3:47:00 PM

Table of Contents

Main Page

Author:

Mark L. Short, Kessia Eugene, Kishore Lav Mutyala

Date:

Oct 31, 2014

Cite:

- Modern Processor Design, John Paul Shen, Mikko H. Lipasti, 2005
- Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis, Alle, Morvan, Derien, IRISA / University of Rennes
- Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance, Emma, Davidson, IEEE Transactions on Computer, 1987
- Instruction Scheduling, Cambridge University UK, 2005, http://www.cl.cam.ac.uk/teaching/2005/OptComp/slides/lecture14.pdf
- The Optimum Pipeline Depth for a Microprocessor, IBM, 2005
- Graph Partitioning Implementation Strategy, University of CA, Berkeley, http://parlab.eecs.berkeley.edu/wiki/_media/patterns/graph_partitioning.pdf
- Data Structures and Algorithms with Object-Oriented Design Patterns in C++, Preiss, 1997, http://www.brpreiss.com/books/opus4/html/page9.html
- Data Abstraction & Problem Solving with C++, Carrano, 2007, http://www.cs.rutgers.edu/~szhou/351/Graphs.pdf
- Technical Report - Polymorphic C++ Debugging for System Design, Doucet, Gupta, University of CA, Irvine, 2000, http://mesl.ucsd.edu/site/pubs/UCI-CECS-TR00-06.pdf

Course: CMPS 5133 Advanced Computer Architecture

Instructor: Dr. Nelson Passos

Assignment:

Your just got hired by a company that produces processors. Their main goal is to start using pipeline design in their processors, but they heard rumors that data dependence may negatively affect the performance of such processors. Your job is to verify that assertion and to show how a four stage pipeline (Fetch, Decode, Execute, Write-Back) works. Data fetching happens during the execution stage. No branch instructions are considered so the code runs straight from beginning to end according with the initial order of the instructions. Each instruction stage consumes one cycle of the processor. Resulting data is available only after the Write Back stage (no forward circuits or any other design optimization). In order to perform your task you receive a sequence of instructions (first line of data) and its perspective dependency graph. Your program should read the data and present the overlapped execution of those instructions. The program must be able to handle 25 instructions.

Hierarchical Index

Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

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:

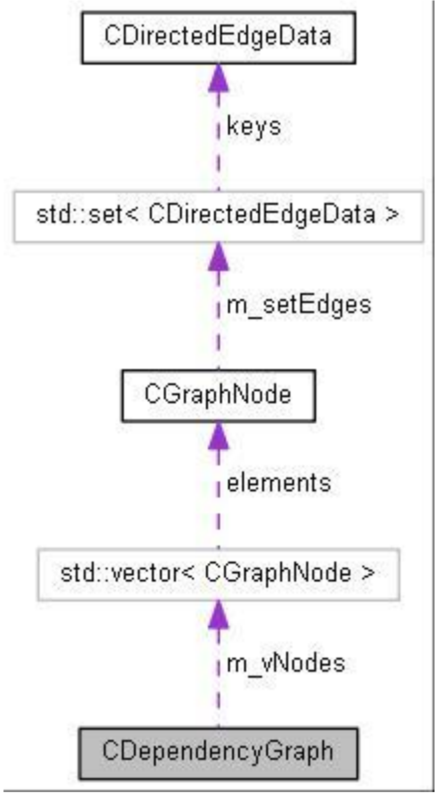
Class Documentation

CDependencyGraph Class Reference

A directed acyclic graph implementation.

`#include <DependencyGraph.h>`

Collaboration diagram for CDependencyGraph:



Public Types

- typedef std::vector
- < CGraphNode >::const_iterator const_iterator

Public Member Functions

- CDependencyGraph ()
  Default Constructor.
- CDependencyGraph (size_t nMaxNodes)
  Init Constructor.
- bool AddNode (const NODE_ID_T &idNode)
  This method adds a new node to the graph.
- bool AddEdge (const NODE_ID_T &idFromNode, const NODE_ID_T &idToNode, int iWeight)
  Adds a directed edge between 2 existing nodes.
- size_t GetNumNodes (void) const
  Retrieves the current number of nodes in the graph.

- size_t GetNumEdges (void) const
  Retrieves the current number of edges in the graph.
- bool HasNode (const NODE_ID_T &idNode) const
  Affords the ability to query for the existance of a particular graph node.
- const_iterator begin (void) const
  Affords iteration functionality.
- const_iterator end (void) const
  Affords iteration functionality.
- ~CDependencyGraph ()
  Default Destructor.

Private Member Functions

- bool IsValidNodeID (const NODE_ID_T &idNode) const
  Performs basic validation of node ID.
- bool IsValidNodeIndex (size_t nIndex) const
  Performs basic validation of a node index.
- size_t GetNodeIndex (const NODE_ID_T &idNode) const
  returns corresponding node index
- CDependencyGraph (const CDependencyGraph &o)
  copy constructor
- CDependencyGraph & operator= (const CDependencyGraph &rhs)
  assignment operator

Private Attributes

- size_t m_nMaxNodes
  maintains an upper limit on nodes allowed in the graph
- size_t m_nNumNodes
  maintains a current number of nodes in the graph
- std::vector< CGraphNode > m_vNodes

Detailed Description

A directed acyclic graph implementation.

The CDependencyGraph class uses a form of an "adjacency list" in order to model a DAG, with the following caveats:

- rather than being implemented as an array of "linked-lists", it is implemented as a vector of sets. A vector provides random access to the node data and a set is implemented as a balanced red-black tree and provides access to the edge end-point in O(log n) time complexity.

Member Typedef Documentation

typedef std::vector<CGraphNode>::const_iterator CDependencyGraph::const_iterator

Constructor & Destructor Documentation

CDependencyGraph::CDependencyGraph ()

Default Constructor.

```
    : m_nMaxNodes(DEFAULT_MAX_NODES),
      m_nNumNodes(0),
      m_vNodes(DEFAULT_MAX_NODES)
{
}
```

CDependencyGraph::CDependencyGraph (size_t *nMaxNodes*)

Init Constructor.

Optimized constructor to allow the pre-allocation of the underlying graph node vector.

Parameters:

in	*nMaxNodes*	The potential number of nodes to be stored in the graph. This value is used to preallocate enough space in the vector.

```
    : m_nMaxNodes(nMaxNodes),
      m_nNumNodes(0),
      m_vNodes(nMaxNodes)
{
}
```

CDependencyGraph::~CDependencyGraph ()

Default Destructor.

```
{
}
```

CDependencyGraph::CDependencyGraph (const CDependencyGraph & *o*) [private]

copy constructor

Member Function Documentation

bool CDependencyGraph::AddEdge (const NODE_ID_T & *idFromNode*, const NODE_ID_T & *idToNode*, int *iWeight*)

Adds a directed edge between 2 existing nodes.

Parameters:

in	*idFromNode*	value of the source node ID
in	*idToNode*	value of the destination node ID
in	*iWeight*	value of Edge weight

Return values:

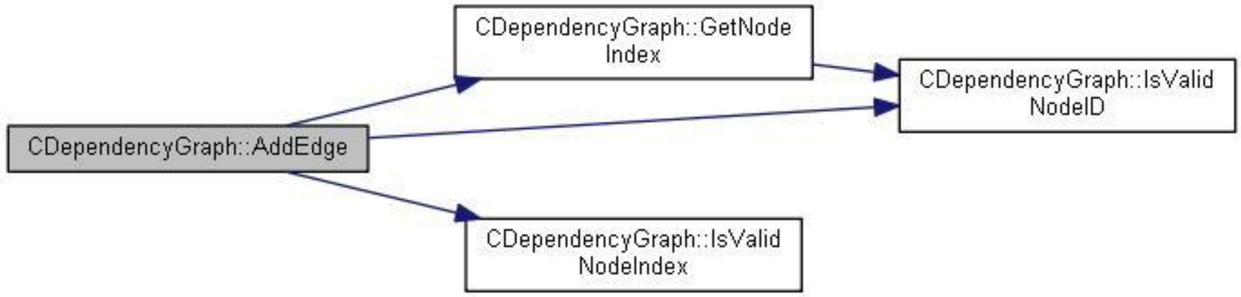
true	if successfully added
false	if already exists or error

References GetNodeIndex(), IsValidNodeID(), IsValidNodeIndex(), and m_vNodes.

Referenced by LoadData().

```
{
    bool bReturn = false;

    // lets validate the input data
    if ( IsValidNodeID(idFromNode) && IsValidNodeID(idToNode) )
    {
        size_t nIndex = GetNodeIndex(idFromNode);

        if ( IsValidNodeIndex(nIndex) )
            bReturn = m_vNodes[nIndex].AddEdge (idToNode, iWeight);
    }


    return bReturn;
}
```

Here is the call graph for this function:


CDependencyGraph::AddEdge
CDependencyGraph::GetNode Index
CDependencyGraph::IsValid NodeID
CDependencyGraph::IsValid NodeIndex

bool CDependencyGraph::AddNode (const NODE_ID_T & *idNode*)

This method adds a new node to the graph.

Parameters:

in	*idNode*	ID of the new node to be added

Return values:

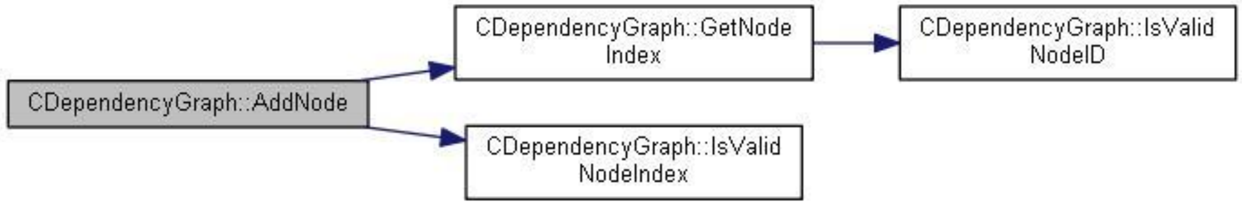
true	if successfully added
false	if already exists or error

References GetNodeIndex(), IsValidNodeIndex(), m_nNumNodes, and m_vNodes.

Referenced by LoadData().

```
{
    bool bReturn = false;

    size_t nNodeIndex = GetNodeIndex(idNode);

    // check to make sure index is not out of bounds of our vector
    if ( IsValidNodeIndex(nNodeIndex) )
    {
        // check to make sure we have not already added this node
        if ( m_vNodes[nNodeIndex].IsValid() == false )
        {
            // node has not been previously added, lets update
            // the node ID with a valid ID to mark it has been
            // added now.
            m_vNodes[nNodeIndex].SetNodeID(idNode);
            m_nNumNodes++;
            bReturn = true;
        }
    }

    return bReturn;
}
```

Here is the call graph for this function:


CDependencyGraph::AddNode
CDependencyGraph::GetNode Index
CDependencyGraph::IsValid NodeID
CDependencyGraph::IsValid NodeIndex

const_iterator CDependencyGraph::begin (void) const

Affords iteration functionality.

Method provides limited read-only access to iterate over the current Node set.

Return values:

const_iterator	iterator for beginning of nonmutable sequence

Referenced by CalculateNumberOfStallsRequired(), and ExecutePipelineSimulation().

```
    { return m_vNodes.begin ( ); };
```

const_iterator CDependencyGraph::end (void) const

Affords iteration functionality.

Method provides limited read-only access to iterate over the current Node set.

Return values:

const_iterator	iterator for end of nonmutable sequence

Referenced by CalculateNumberOfStallsRequired(), and ExecutePipelineSimulation().

```
    { return m_vNodes.end ( ); };
```

size_t CDependencyGraph::GetNodeIndex (const NODE_ID_T & *idNode*) const `[private]`

returns corresponding node index

Performs a basic hash-translation of the index of the node from its associated ID. The returned index corresponds to the nodes offset within the vector.

Parameters:

in	*idNode*	node ID

Return values:

size_t	vector index for idNode
INVALID_NODE_INDEX	if no valid index exists

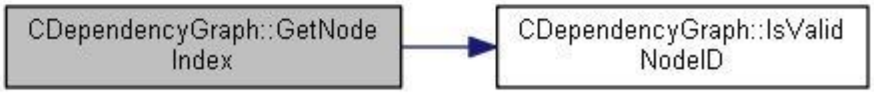
Following method performs a pseudo-hashing of the node ID to determine the proper location of the Node in the vector

References INVALID_NODE_INDEX, and IsValidNodeID().

Referenced by AddEdge(), AddNode(), and HasNode().

```
{
    size_t nReturn = INVALID_NODE_INDEX;

    if ( IsValidNodeID(idNode) )
    {
        const std::locale& loc = std::locale();  // construct from the default locale
        TCHAR cId = static_cast<TCHAR>(idNode);
        nReturn   = std::tolower(cId, loc) - _T('a');
    }

    return nReturn;
}
```

Here is the call graph for this function:


CDependencyGraph::GetNode Index
CDependencyGraph::IsValid NodeID

size_t CDependencyGraph::GetNumEdges (void) const

Retrieves the current number of edges in the graph.

Return values:

size_t	the number of edges (or arcs) in the graph

References m_nMaxNodes, and m_vNodes.

```
{
    size_t nNumEdges = 0;

    // following static cast added to address compiler warning
    for (int i = 0; i < static_cast<int>(m_nMaxNodes); i++)
    {
        nNumEdges += m_vNodes[i].GetNumEdges();
    }

    return nNumEdges;
}
```

size_t CDependencyGraph::GetNumNodes (void) const

Retrieves the current number of nodes in the graph.

Return values:

size_t	the number of nodes (or vertices) in the graph

References m_nNumNodes.

Referenced by CalculateCompleteOverlappedExecutionCycles(), CalculatePartialOverlappedExecutionCycles(), and CalculateSequentialExecutionCycles().

```
{
    return m_nNumNodes;
}
```

bool CDependencyGraph::HasNode (const NODE_ID_T & *idNode*) const

Affords the ability to query for the existance of a particular graph node.

Parameters:

in	*idNode*	target node ID

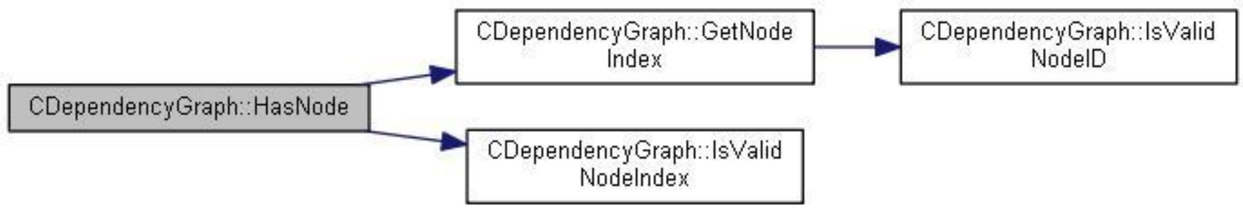
Return values:

true	if idNode is found in the graph

References GetNodeIndex(), IsValidNodeIndex(), and m_vNodes.

```
{
    bool bReturn = false;

    // first determine if ID will even convert to an actual Node index;

    size_t nIndex = GetNodeIndex(idNode);

    if ( IsValidNodeIndex(nIndex) )
    {
        // now determine if that node associated with the index has
        // been added to the graph or not
        bReturn = m_vNodes[nIndex].IsValid();
    }

    return bReturn;
};
```

Here is the call graph for this function:


CDependencyGraph::HasNode
CDependencyGraph::GetNode Index
CDependencyGraph::IsValid NodeID
CDependencyGraph::IsValid NodeIndex

bool CDependencyGraph::IsValidNodeID (const NODE_ID_T & *idNode*) const`[private]`

Performs basic validation of node ID.

Parameters:

in	*idNode*	value to be verified

Return values:

true	if idNode is a valid ID

Referenced by AddEdge(), and GetNodeIndex().

```
{
    bool bReturn = false;
// we are only allowing up to 25 instructions, each represented by a char in
// in the range of [a..y].  lets allow some flexibility by being case-insensitive.
    if ( (idNode >= 'a' && idNode <= 'y') || (idNode >= 'A' && idNode <= 'Y') )
        bReturn = true;

    return bReturn;
}
```

bool CDependencyGraph::IsValidNodeIndex (size_t *nIndex*) const`[private]`

Performs basic validation of a node index.

The nIndex is probed to see if it falls within the underlying vector boundaries.

Parameters:

in	*nIndex*	index to be verified

Return values:

true	if nIndex falls within the current vector range
false	if nIndex is found out-of-bounds

References m_nMaxNodes.

Referenced by AddEdge(), AddNode(), and HasNode().

```
{
    bool bReturn = false;

    if ( nIndex < m_nMaxNodes )
        bReturn = true;

    return bReturn;
}
```

CDependencyGraph& CDependencyGraph::operator= (const CDependencyGraph & *rhs*) [private]

assignment operator

Member Data Documentation

size_t CDependencyGraph::m_nMaxNodes [private]

maintains an upper limit on nodes allowed in the graph

Referenced by GetNumEdges(), and IsValidNodeIndex().

size_t CDependencyGraph::m_nNumNodes [private]

maintains a current number of nodes in the graph

Referenced by AddNode(), and GetNumNodes().

std::vector<CGraphNode> CDependencyGraph::m_vNodes [private]

Referenced by AddEdge(), AddNode(), GetNumEdges(), and HasNode().

The documentation for this class was generated from the following files:

- PipelineProject/DependencyGraph.h
- PipelineProject/DependencyGraph.cpp

CDirectedEdgeData Class Reference

Maintains directed edge data properties.

```
#include <DependencyGraph.h>
```

Public Member Functions

- CDirectedEdgeData ()
  Default Constructor.
- CDirectedEdgeData (const NODE_ID_T &idToNode, int iWeight=0)
  Initialization Constructor.
- void SetNodeID (const NODE_ID_T &idSet)
  Sets the value or ID of the edge destination node.
- NODE_ID_T GetDestNodeID (void) const
  Retrieves the value or ID of the edge destination node.
- void SetWeight (int iSet)
  Sets the weight value associated with this edge.
- int GetWeight (void) const
  Retrieves the current weight value associated with this edge.
- bool operator< (const CDirectedEdgeData &rhs) const
  Comparison operation required by STL.
- bool operator== (const CDirectedEdgeData &rhs) const
  Comparison operation required by STL.
- bool operator> (const CDirectedEdgeData &rhs) const
  Comparison operation required by STL.

Private Attributes

- NODE_ID_T m_idDestNode
  maintains the destination node ID
- int m_iWeight
  weight value assigned to this edge

Detailed Description

Maintains directed edge data properties.

Additionally, CDirectedEdgeData overrides the default behavior of the comparison operators such that it is ordered and identified only by its m_idDestNode data member, and thus able to be stored in an STL collection using its node ID as the key.

Constructor & Destructor Documentation

CDirectedEdgeData::CDirectedEdgeData ()

Default Constructor.

```
           : m_idDestNode (INVALID_NODE_ID), m_iWeight(0)
```

```
    {};
```

CDirectedEdgeData::CDirectedEdgeData (const NODE_ID_T & *idToNode*, int *iWeight* = 0)

Initialization Constructor.

```
        : m_idDestNode(idToNode), m_iWeight(iWeight)
    {};
```

Member Function Documentation

NODE_ID_T CDirectedEdgeData::GetDestNodeID (void) const

Retrieves the value or ID of the edge destination node.

Return values:

NODE_ID_T	the edge's destination node ID
INVALID_NODE_ID	on error

References m_idDestNode.

```
    { return m_idDestNode; };
```

int CDirectedEdgeData::GetWeight (void) const

Retrieves the current weight value associated with this edge.

Return values:

int	the current edge weight

References m_iWeight.

```
    { return m_iWeight; };
```

bool CDirectedEdgeData::operator< (const CDirectedEdgeData & *rhs*) const

Comparison operation required by STL.

Performs comparison evaluation of this class using the m_idDestNode value only.

Return values:

bool	less than '<' evaluation

References m_idDestNode.

```
    { return m_idDestNode < rhs.m_idDestNode; };
```

bool CDirectedEdgeData::operator== (const CDirectedEdgeData & *rhs*) const

Comparison operation required by STL.

Performs comparison evaluation of this class using the m_idDestNode value only.

Return values:

bool	equal '==' evaluation

References m_idDestNode.

```
    { return m_idDestNode == rhs.m_idDestNode; };
```

bool CDirectedEdgeData::operator> (const CDirectedEdgeData & *rhs*) const

Comparison operation required by STL.

Performs comparison evaluation of this class using the m_idDestNode value only.

Return values:

bool	greater than '>' evaluation

References m_idDestNode.

```
    { return m_idDestNode > rhs.m_idDestNode; };
```

void CDirectedEdgeData::SetNodeID (const NODE_ID_T & *idSet*)

Sets the value or ID of the edge destination node.

Parameters:

in	*idSet*	the destination node ID to be set

```
    { m_idDestNode = idSet; };
```

void CDirectedEdgeData::SetWeight (int *iSet*)

Sets the weight value associated with this edge.

Parameters:

in	*iSet*	new weight value to be set

```
    { m_iWeight = iSet; };
```

Member Data Documentation

NODE_ID_T CDirectedEdgeData::m_idDestNode `[private]`

maintains the destination node ID

Referenced by GetDestNodeID(), operator<(), operator==(), and operator>().

int CDirectedEdgeData::m_iWeight `[private]`

weight value assigned to this edge

Referenced by GetWeight().

The documentation for this class was generated from the following file:

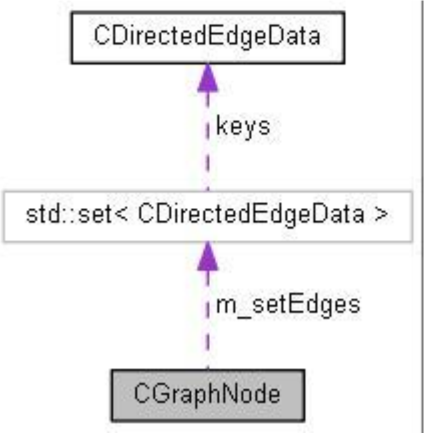
- PipelineProject/DependencyGraph.h

CGraphNode Class Reference

a directed graph node implementation.

`#include <DependencyGraph.h>`

Collaboration diagram for CGraphNode:



Public Types

- typedef EDGE_SET_T::const_iterator const_iterator

Public Member Functions

- CGraphNode ()
  Default Constructor.
- CGraphNode (const NODE_ID_T &idNode)
  Initialization Constructor.
- void SetNodeID (const NODE_ID_T &idSet)
  Sets this object's node ID.
- NODE_ID_T GetNodeID (void) const
  Retrieves this object's node ID.
- bool IsValid (void) const
  Used to check to see if this node is active.
- bool AddEdge (const NODE_ID_T &idToNode, int iWeight)
  adds a new edge to the graph
- bool AddEdge (const CDirectedEdgeData &edge)
  Adds a new edge the graph.
- size_t GetNumEdges (void) const
  Retrieves number of edges.
- const_iterator beginEdge (void) const
  Affords iterator functionality.
- const_iterator endEdge (void) const
  Affords iterator functionality.
- bool HasEdge (const NODE_ID_T &idToNode) const
  Test if given edge exists.

- ~CGraphNode ()
  Default Destructor.

Private Types

- typedef std::set
- < CDirectedEdgeData > EDGE_SET_T
- typedef EDGE_SET_T::_Pairib _Pairib

Private Attributes

- NODE_ID_T m_ID
  this is the node value or ID
- EDGE_SET_T m_setEdges
  this is a set of directed 'out' edges from this node

Detailed Description

a directed graph node implementation.

The CGraphNode class maintains a node ID, as well as a set of CDirectedEdgeData elements representing the set of 'out' edges from this graph node, as a form of an adjacency list.

Member Typedef Documentation

typedef EDGE_SET_T::_Pairib CGraphNode::_Pairib [private]

typedef EDGE_SET_T::const_iterator CGraphNode::const_iterator

typedef std::set<CDirectedEdgeData> CGraphNode::EDGE_SET_T [private]

Constructor & Destructor Documentation

CGraphNode::CGraphNode ()

Default Constructor.

```
        : m_ID(INVALID_NODE_ID), m_setEdges()
    { };
```

CGraphNode::CGraphNode (const NODE_ID_T & *idNode*)

Initialization Constructor.

```
        : m_ID(idNode), m_setEdges()
    { };
```

CGraphNode::~CGraphNode ()

Default Destructor.

```
    { };
```

Member Function Documentation

bool CGraphNode::AddEdge (const NODE_ID_T & *idToNode*, int *iWeight*)

adds a new edge to the graph

This method adds a new edge, originating from this node, to the associated edge set. The idToNode is presumed to be a valid destination node in the underlying graph

Parameters:

in	*idToNode*	ID of the destination node
in	*iWeight*	Edge's weight value

Return values:

true	if edge successfully added
false	on error

```
{
    return AddEdge( CDirectedEdgeData(idToNode, iWeight) );
}
```

bool CGraphNode::AddEdge (const CDirectedEdgeData & *edge*)

Adds a new edge the graph.

This method adds a new edge, originating from this node, to the associated edge set.

Parameters:

in	*edge*	data object containing associated node destination and edge weight information

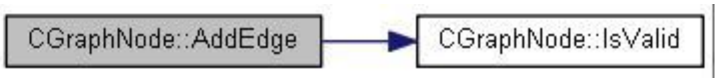
Return values:

true	if edge successfully added
false	on error

References IsValid(), and m_setEdges.

```
{
    bool bReturn = false;

    if ( IsValid ( ) ) // check to make sure we are a valid node before assigning any edges
    {
         Pairib Result = m_setEdges.insert ( edge );

        if ( Result.second )  // was the insert successful ?
        {
            bReturn = true;
        }
    }

    return bReturn;
}
```

Here is the call graph for this function:



const_iterator CGraphNode::beginEdge (void) const

Affords iterator functionality.

Method provides limited read-only access to iterate over the current edge set.

Return values:

const_iterator	iterator for beginning of nonmutable edge sequence

```
    { return m_setEdges.begin(); };
```

const_iterator CGraphNode::endEdge (void) const

Affords iterator functionality.

Method provides limited read-only access to iterate over the current edge set.

Return values:

const_iterator	iterator for end of nonmutable edge sequence

```
    { return m_setEdges.end(); };
```

NODE_ID_T CGraphNode::GetNodeID (void) const

Retrieves this object's node ID.

Return values:

NODE_ID_T	the current node ID
INVALID_NODE_ID	if node is vacant or has not been assigned a value

References m_ID.

```
    { return m_ID; };
```

size_t CGraphNode::GetNumEdges (void) const

Retrieves number of edges.

This method retrieves the current number of 'out' edges originating from this node.

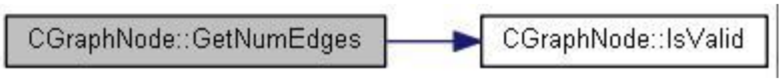
Return values:

size_t	current number of nodes in the edge set
0	if this node is not a valid node

References IsValid().

```
    { return IsValid() ? m_setEdges.size() : 0; };
```

Here is the call graph for this function:



bool CGraphNode::HasEdge (const NODE_ID_T & *idToNode*) const

Test if given edge exists.

Parameters:

in	*idToNode*	target node

Return values:

true	if there exists an edge from this node to target node
false	if target node or edge is not found

```
    {
        const_iterator itr = m_setEdges.find( CDirectedEdgeData(idToNode) );
        return (itr != m_setEdges.end() );
    };
```

bool CGraphNode::IsValid (void) const

Used to check to see if this node is active.

This method checks to see if this node is active and assigned to a graph.

Return values:

true	if current node ID is valid, denoting it has been added to the graph
false	if m_ID == INVALID_NODE_ID

References INVALID_NODE_ID.

Referenced by AddEdge(), and GetNumEdges().

```
    { return m_ID != INVALID_NODE_ID; };
```

void CGraphNode::SetNodeID (const NODE_ID_T & *idSet*)

Sets this object's node ID.

Parameters:

in	*idSet*	new node ID to be set

```
    { m_ID = idSet; };
```

Member Data Documentation

NODE_ID_T CGraphNode::m_ID [private]

this is the node value or ID

Referenced by GetNodeID().

EDGE_SET_T CGraphNode::m_setEdges [private]

this is a set of directed 'out' edges from this node

Referenced by AddEdge().

The documentation for this class was generated from the following files:

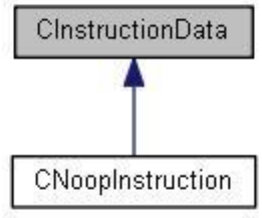
- PipelineProject/DependencyGraph.h
- PipelineProject/DependencyGraph.cpp

CInstructionData Class Reference

Instruction data and state.

`#include <PipelineSim.h>`

Inheritance diagram for CInstructionData:



Public Member Functions

- CInstructionData ()
  Default Constructor.
- CInstructionData (const INSTRUCTION_T &instruction, bool bDataDependent)
  Initialization Constructor.
- CInstructionData (const INSTRUCTION_T &instruction, PS_PIPELINE_STATE psState=PS_INVALID, bool bDataDependent=false)
  Initialization Constructor.
- INSTRUCTION_T GetInstruction (void) const
  Retrieves the instruction.
- PS_PIPELINE_STATE GetState (void) const
  Gets the instruction pipeline state.
- void SetState (PS_PIPELINE_STATE psSet)
  Set the instruction pipeline state.
- bool IsDataDependent (void) const
- void SetDataDependent (bool bSet=true)
- bool IsNOOP (void) const
- ~CInstructionData ()
  Destructor.

Private Attributes

- INSTRUCTION_T m_Instruction
- PS_PIPELINE_STATE m_psState
- bool m_bDataDependent

Detailed Description

Instruction data and state.

Constructor & Destructor Documentation

CInstructionData::CInstructionData ()

Default Constructor.

```
        : m_Instruction(INVALID_INSTRUCTION),
          m_psState(PS_INVALID),
          m_bDataDependent(false)
    {};
```

CInstructionData::CInstructionData (const INSTRUCTION_T & *instruction*, bool *bDataDependent*)

Initialization Constructor.

```
        : m_Instruction ( instruction ),
          m_psState ( PS_INVALID ),
          m_bDataDependent ( bDataDependent )
    {};
```

CInstructionData::CInstructionData (const INSTRUCTION_T & *instruction*, PS_PIPELINE_STATE *psState* = `PS_INVALID`, bool *bDataDependent* = `false`)

Initialization Constructor.

```
        : m_Instruction ( instruction ),
          m_psState ( psState ),
          m_bDataDependent(bDataDependent)
    {};
```

CInstructionData::~CInstructionData ()

Destructor.

```
{};
```

Member Function Documentation

INSTRUCTION_T CInstructionData::GetInstruction (void) const

Retrieves the instruction.

Return values:

INSTRUCTION_T	

References m_Instruction.

```
    { return m_Instruction; };
```

PS_PIPELINE_STATE CInstructionData::GetState (void) const

Gets the instruction pipeline state.

Return values:

PS_INVALID	initial default state
PS_IF	Instruction Fetch

PS_ID	Instruction Decode
PS_EX	Execute
PS_WB	Write Back
PS_COMPLETED	instruction processing completed

References m_psState.

```
      { return m_psState; };
```

bool CInstructionData::IsDataDependent (void) const

References m_bDataDependent.

```
      { return m_bDataDependent; };
```

bool CInstructionData::IsNOOP (void) const

References NOOP_INSTRUCTION.

```
      { return m_Instruction == NOOP_INSTRUCTION; };
```

void CInstructionData::SetDataDependent (bool *bSet* = true)

```
      { m_bDataDependent = bSet; };
```

void CInstructionData::SetState (PS_PIPELINE_STATE *psSet*)

Set the instruction pipeline state.

```
      { m_psState = psSet; };
```

Member Data Documentation

bool CInstructionData::m_bDataDependent [private]

Referenced by IsDataDependent().

INSTRUCTION_T CInstructionData::m_Instruction [private]

Referenced by GetInstruction().

PS_PIPELINE_STATE CInstructionData::m_psState [private]

Referenced by GetState().

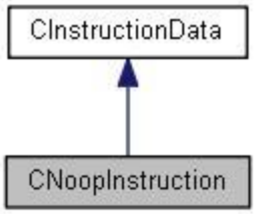
The documentation for this class was generated from the following file:

- PipelineProject/PipelineSim.h

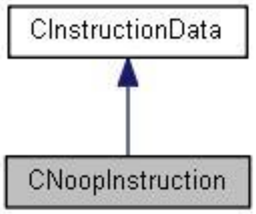
CNoopInstruction Class Reference

```
#include <PipelineSim.h>
```

Inheritance diagram for CNoopInstruction:



Collaboration diagram for CNoopInstruction:



Public Member Functions

- CNoopInstruction ()
- CNoopInstruction (PS_PIPELINE_STATE psState)

Constructor & Destructor Documentation

CNoopInstruction::CNoopInstruction ()

```
        : CInstructionData(NOOP_INSTRUCTION)
    { };
```

CNoopInstruction::CNoopInstruction (PS_PIPELINE_STATE *psState*)

```
        : CInstructionData(NOOP_INSTRUCTION, psState)
    { };
```

The documentation for this class was generated from the following file:

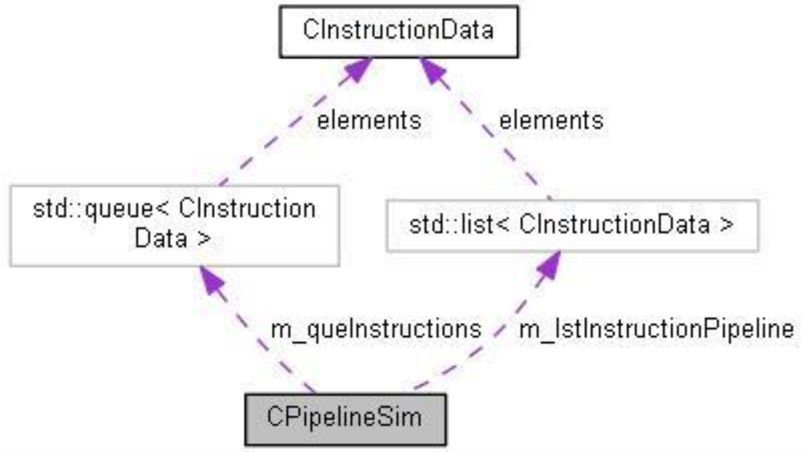
- PipelineProject/PipelineSim.h

CPipelineSim Class Reference

A 4-staged pipeline simulation class.

```
#include <PipelineSim.h>
```

Collaboration diagram for CPipelineSim:



Public Member Functions

- CPipelineSim ()
  Default Constructor.
- DWORD GetCycle (void) const
  Retrieves the current number of cycles executed.
- DWORD GetStallCount (void) const
  Retrieves the count of stalls introduced into the pipeline.
- DWORD GetCompletionCount (void) const
  Retrieves the current count of completed instructions.
- bool ProcessNextCycle (void)
  Process next pipeline instruction cycle.
- size_t InsertInstruction (const CInstructionData &instruction)
  Adds the instruction to the instruction queue.
- tostream & OutputCurrentInstructionCycle (tostream &os)
  formats and outputs current pipelined instructions to the provided stream
- ~CPipelineSim ()
  Destructor.

Private Attributes

- DWORD m_dwCycle
  maintains current pipeline cycle
- DWORD m_dwStallCtr
  a count of the stalls introduced
- DWORD m_dwCompletedCtr
  a count of the instructions completed execution

- DWORD m_dwMaxPipelineDepth
  limit on instructions in the pipeline
- std::list< CInstructionData > m_lstInstructionPipeline
  our instruction pipeline
- std::queue< CInstructionData > m_queInstructions
  our instruction queue

Detailed Description

A 4-staged pipeline simulation class.

The following class attempts to simulate the processing of instructions in a four-staged pipeline. In a four-stage pipeline it is possible to execute 'sub-instructions' of four separate instructions at the same time, with each one having a pipeline 'state' denoting what stage of the execution process it is in.

Additionally, no two instructions may share the same 'state' concurrently.

This class uses simulates a pipeline in the form of a linked-list to model the concurrent instruction processing.

Constructor & Destructor Documentation

CPipelineSim::CPipelineSim ()

Default Constructor.

```
    : m_dwCycle(0),
      m_dwStallCtr(0),
      m_dwCompletedCtr(0),
      m_dwMaxPipelineDepth ( CONCURRENT_INSTRUCTION_LIMIT ),
      m_lstInstructionPipeline(),
      m_queInstructions()
{
}
```

CPipelineSim::~CPipelineSim ()

Destructor.

```
{
}
```

Member Function Documentation

DWORD CPipelineSim::GetCompletionCount (void) const

Retrieves the current count of completed instructions.

Return values:

DWORD	count of completed instructions

References m_dwCompletedCtr.

```
    { return m_dwCompletedCtr; };
```

DWORD CPipelineSim::GetCycle (void) const

Retrieves the current number of cycles executed.

Return values:

DWORD	current cycle

References m_dwCycle.

```
    { return m_dwCycle; };
```

DWORD CPipelineSim::GetStallCount (void) const

Retrieves the count of stalls introduced into the pipeline.

Return values:

DWORD	count of stalls

References m_dwStallCtr.

```
    { return m_dwStallCtr; };
```

size_t CPipelineSim::InsertInstruction (const CInstructionData & *instruction*)

Adds the instruction to the instruction queue.

Queued instructions are popped off the queue and inserted into the pipeline during the ProcessNextCycle method call. Instruction state is update accordingly to denote the current pipeline stage it is in.

Parameters:

in	*instruction*	instruction data to add to the queue for further insertion and processing in the pipeline.

Return values:

size_t	number of instructions in the queue

References m_queInstructions.

Referenced by ExecutePipelineSimulation().

```
{
    m_queInstructions.push(instruction);

    return m_queInstructions.size();
};
```

tostream & CPipelineSim::OutputCurrentInstructionCycle (tostream & *os*)

formats and outputs current pipelined instructions to the provided stream

Parameters:

in,out	*os*	destination output stream

References m_lstInstructionPipeline, PS_EX, PS_ID, PS_IF, and PS_WB.

Referenced by ExecutePipelineSimulation().

```
{
    for ( LstIterator it = m_lstInstructionPipeline.begin ( ); it !=
m_lstInstructionPipeline.end ( ); ++it )
    {
        switch (it->GetState())
        {
            case PS_IF:
            case PS_ID:
            case PS_EX:
            case PS_WB:
                os << it->GetInstruction() << _T(" ");
                break;
            default:
                break;
        }
    }

    os << std::endl;

    return os;
}
```

bool CPipelineSim::ProcessNextCycle (void)

Process next pipeline instruction cycle.

Increments cycle counter and continues processing of the currently que'ed instructions, advancing each one to the next pipeline state accordingly.

Return values:

true	if there are subsequent instructions to be executed.
false	if there are no more instructions to be executed.

References m_dwCompletedCtr, m_dwCycle, m_dwMaxPipelineDepth, m_dwStallCtr, m_lstInstructionPipeline, m_queInstructions, PS_COMPLETED, PS_EX, PS_ID, PS_IF, PS_INVALID, and PS_WB.

Referenced by ExecutePipelineSimulation().

```
{
    bool bReturn = false;
    // increment the cycle counter
    m_dwCycle++;

    // Begin processing our instruction que
    // check our current instruction pipeline size and see if we have room
    if ( m_lstInstructionPipeline.size ( ) <= m_dwMaxPipelineDepth )
    {
        // check our instruction queue and see if we have anything left to execute

        if (m_queInstructions.size() != 0)
        {
            CInstructionData instruction = m_queInstructions.front();

            m_queInstructions.pop();

        // insert the instruction at the beginning of our pipeline
            m_lstInstructionPipeline.push_front ( instruction );

            bReturn = true;
        }
        else
        {
            // nothing left in the instruction que,
            // so we insert NOOPS until everything
            // clears the pipeline
            CNoopInstruction NOOP;

            m_lstInstructionPipeline.push_front ( NOOP );
        }
    }

    bool bStalled = false;
    // reverse iterate over the instruction currently in the pipeline
```

```
   for (rLstIterator itr = m_lstInstructionPipeline.rbegin(); itr !=
m_lstInstructionPipeline.rend() && (bStalled == false); ++itr)
   {
       PS_PIPELINE_STATE stInstruction = itr->GetState();

       switch (stInstruction)
       {

           case PS_INVALID:   // initial default state
               itr->SetState(PS_IF);
               if ( itr->IsNOOP ( ) == false )
                   bReturn = true;
               break;

           case PS_IF:        // Instruction Fetch state
               itr->SetState(PS_ID);
               if ( itr->IsNOOP ( ) == false )
                   bReturn = true;
               break;

           case PS_ID:        // Instruction Decode state
               // need to verify if a dependency exists between this instruction
               // and the immediately previous instruction

               if (itr->IsDataDependent())
               {
                   // we have to introduce a stall here
                   itr->SetDataDependent(false);

                   CNoopInstruction NOOP(PS_EX);

                   m_lstInstructionPipeline.insert(itr.base(), NOOP);

                    bStalled = true;
                    m_dwStallCtr++;
                }
                else
                {
                    itr->SetState(PS_EX);
                }

                if (itr->IsNOOP() == false)
                    bReturn = true;
                break;

            case PS_EX:        // Instruction Execute state
                itr->SetState(PS_WB);
                if (itr->IsNOOP() == false)
                    bReturn = true;
                break;

            case PS_WB:         // Instruction Write Back state
                itr->SetState (PS_COMPLETED); // mark this for removal later

                if (itr->IsNOOP() == false)
                    m_dwCompletedCtr++;

                break;

            case PS_COMPLETED:
            default:

                break;
        }
    }

// check to see if we have a completed instruction for removal from the pipeline
    if (m_lstInstructionPipeline.back().GetState() == PS_COMPLETED)
        m_lstInstructionPipeline.pop_back();

    return bReturn;
```

```
};
```

Member Data Documentation

DWORD CPipelineSim::m_dwCompletedCtr [private]

a count of the instructions completed execution

Referenced by GetCompletionCount(), and ProcessNextCycle().

DWORD CPipelineSim::m_dwCycle [private]

maintains current pipeline cycle

Referenced by GetCycle(), and ProcessNextCycle().

DWORD CPipelineSim::m_dwMaxPipelineDepth [private]

limit on instructions in the pipeline

Referenced by ProcessNextCycle().

DWORD CPipelineSim::m_dwStallCtr [private]

a count of the stalls introduced

Referenced by GetStallCount(), and ProcessNextCycle().

std::list<CInstructionData> CPipelineSim::m_lstInstructionPipeline [private]

our instruction pipeline

Referenced by OutputCurrentInstructionCycle(), and ProcessNextCycle().

std::queue<CInstructionData> CPipelineSim::m_queInstructions [private]

our instruction queue

Referenced by InsertInstruction(), and ProcessNextCycle().

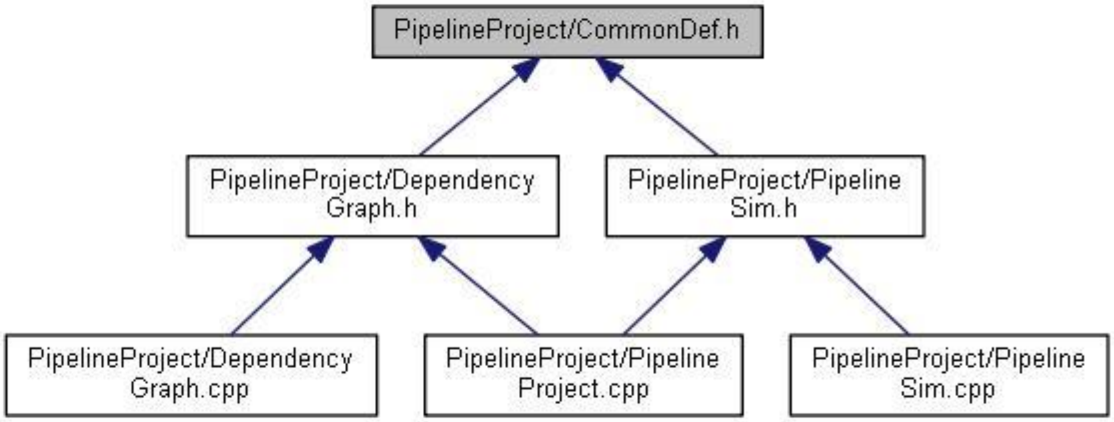
The documentation for this class was generated from the following files:

- PipelineProject/PipelineSim.h
- PipelineProject/PipelineSim.cpp

File Documentation

PipelineProject/CommonDef.h File Reference

Common type definitions.
This graph shows which files directly or indirectly include this file:


PipelineProject/CommonDef.h
PipelineProject/Dependency Graph.h
PipelineProject/Pipeline Sim.h
PipelineProject/Dependency Graph.cpp
PipelineProject/Pipeline Project.cpp
PipelineProject/Pipeline Sim.cpp

Macros

- #define _COMMON_DEF_H__

Typedefs

- typedef unsigned __int8 BYTE
  8-bit unsigned type
- typedef unsigned __int32 DWORD
  32-bit unsigned type

Detailed Description

Common type definitions.

Author:
Mark L. Short

$Date:$ $Revision:$

Macro Definition Documentation

#define _COMMON_DEF_H__

Typedef Documentation

typedef unsigned __int8 BYTE

8-bit unsigned type

typedef unsigned __int32 DWORD

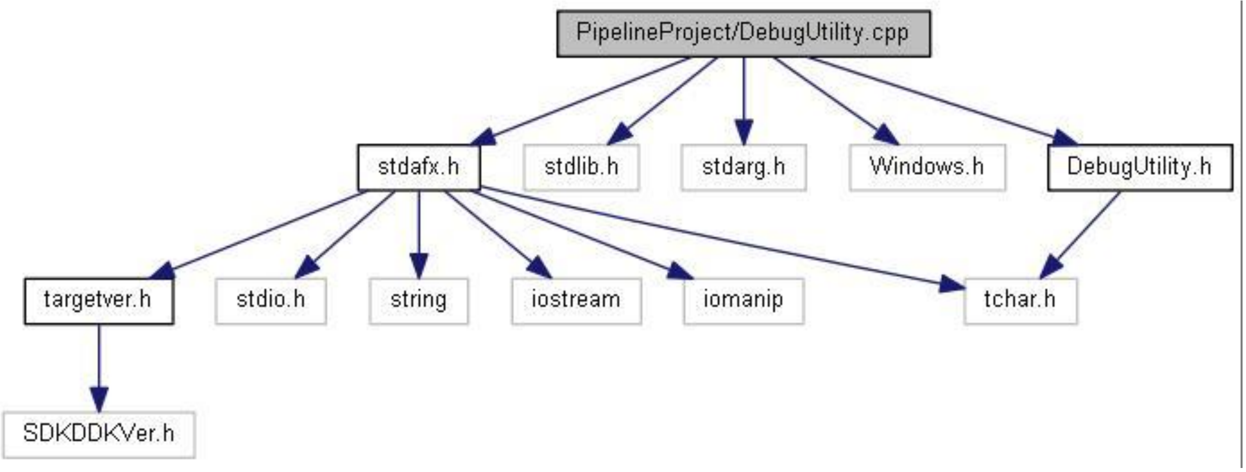
32-bit unsigned type

PipelineProject/DebugUtility.cpp File Reference

Implementation of DebugUtility.cpp.

```
#include "stdafx.h"
#include "stdlib.h"
#include "stdarg.h"
#include "Windows.h"
#include "DebugUtility.h"
```

Include dependency graph for DebugUtility.cpp:



Functions

- int DebugTrace (const TCHAR *szFmt,...)
  Directs output to the IDE output window.
- TCHAR * GetModulePath (TCHAR *szModulePath, size_t cchLen)
  Retrieves the current executable directory.

Detailed Description

Implementation of DebugUtility.cpp.

Author:

Mark L. Short

Date:

November 24, 2014

Function Documentation

int DebugTrace (const TCHAR * *szFmt*, ...)

Directs output to the IDE output window.

Parameters:

in	*szFmt*	printf-styled format string

Return values:

int	the number of characters written if the number of characters to write is less than or equal to count; if the number of characters to write is greater than count, these functions return -1 indicating that output has been truncated. The return value does not include the terminating null, if one is written.

```
{
    TCHAR szDebugMsg[512] = { 0 };

    va_list vaArgs;
    va_start (vaArgs, szFmt);

    // use the format string and arguments to construct the debug output string
    int iReturnVal = _vsntprintf (szDebugMsg, _countof (szDebugMsg) - 1, szFmt, vaArgs);
    va_end (vaArgs);

    ::OutputDebugString (szDebugMsg);
    return iReturnVal;

}
```

TCHAR* GetModulePath (TCHAR * *szModulePath*, size_t *cchLen*)

Retrieves the current executable directory.

Parameters:

out	*szModulePath*	destination memory address used to write application's directory path
in	*cchLen*	count of charecters in available to be written in destination buffer

Return values:

*TCHAR**	destination address
NULL	on error

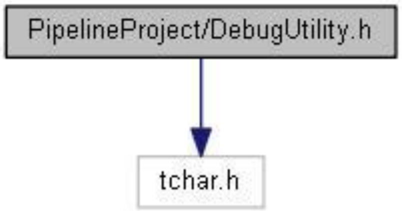
```
{
    // Get the executable file path
    TCHAR szModuleFileName[_MAX_PATH] = { 0 };

    // Note, if HANDLE is NULL, GetModuleFileName is supposed to return the file path to the
    // current executable, but it appears that it is inconsistently returning filename as
    // well....
    DWORD dwStrLen = ::GetModuleFileName (NULL, szModuleFileName, _countof(szModuleFileName) );

    TCHAR szDir[_MAX_PATH] = {0};

    _tsplitpath(szModuleFileName, szDir, &szDir[2], NULL, NULL);

    return _tcsncpy(szModulePath, szDir, cchLen);
}
```

PipelineProject/DebugUtility.h File Reference

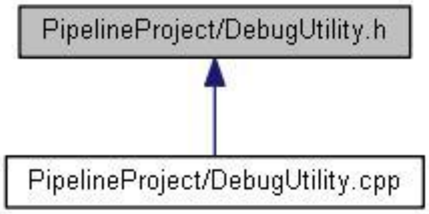
Debugging and utility method declarations.

```
#include "tchar.h"
```

Include dependency graph for DebugUtility.h:



This graph shows which files directly or indirectly include this file:


PipelineProject/DebugUtility.h
PipelineProject/DebugUtility.cpp

Functions

- int DebugTrace (const TCHAR *szFmt,...)
  Directs output to the IDE output window.
- TCHAR * GetModulePath (TCHAR *szModulePath, size_t cchLen)
  Retrieves the current executable directory.

Detailed Description

Debugging and utility method declarations.

Author:

Mark L. Short

Date:

November 24, 2014

Function Documentation

int DebugTrace (const TCHAR * *szFmt*, ...)

Directs output to the IDE output window.

Parameters:

in	*szFmt*	printf-styled format string

Return values:

int	the number of characters written if the number of characters to write is less than or equal to count; if the number of characters to write is greater than count, these functions return -1 indicating that output has been truncated. The return value does not include the terminating null, if one is written.

```
{
    TCHAR szDebugMsg[512] = { 0 };

    va_list vaArgs;
    va_start (vaArgs, szFmt);

    // use the format string and arguments to construct the debug output string
    int iReturnVal = _vsntprintf (szDebugMsg, _countof (szDebugMsg) - 1, szFmt, vaArgs);
    va_end (vaArgs);

    ::OutputDebugString (szDebugMsg);
    return iReturnVal;

}
```

TCHAR* GetModulePath (TCHAR * *szModulePath*, size_t *cchLen*)

Retrieves the current executable directory.

Parameters:

out	*szModulePath*	destination memory address used to write application's directory path
in	*cchLen*	count of charecters in available to be written in destination buffer

Return values:

*TCHAR**	destination address
NULL	on error

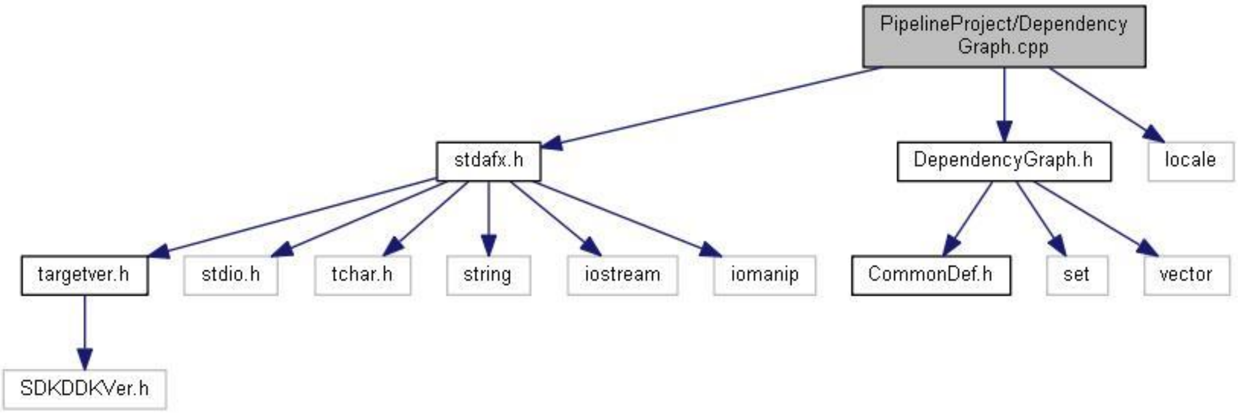
```
{
    // Get the executable file path
    TCHAR szModuleFileName[_MAX_PATH] = { 0 };

    // Note, if HANDLE is NULL, GetModuleFileName is supposed to return the file path to the
    // current executable, but it appears that it is inconsistently returning filename as
    // well....
    DWORD dwStrLen = ::GetModuleFileName (NULL, szModuleFileName, _countof(szModuleFileName) );

    TCHAR szDir[_MAX_PATH] = {0};

    _tsplitpath(szModuleFileName, szDir, &szDir[2], NULL, NULL);

    return _tcsncpy(szModulePath, szDir, cchLen);
}
```

PipelineProject/DependencyGraph.cpp File Reference

CDependencyGraph class implementation.

```
#include "stdafx.h"
#include "DependencyGraph.h"
#include <locale>
```

Include dependency graph for DependencyGraph.cpp:



Variables

- const size_t DEFAULT_MAX_NODES = 10

Detailed Description

CDependencyGraph class implementation.

Author:

Mark L. Short

Date:

November 23, 2014

Variable Documentation

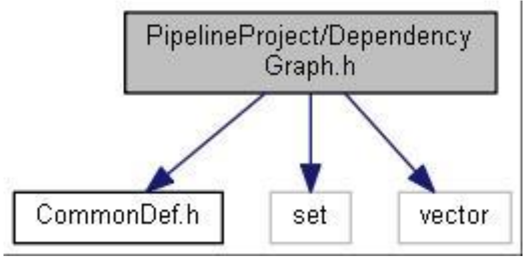
const size_t DEFAULT_MAX_NODES = 10

PipelineProject/DependencyGraph.h File Reference

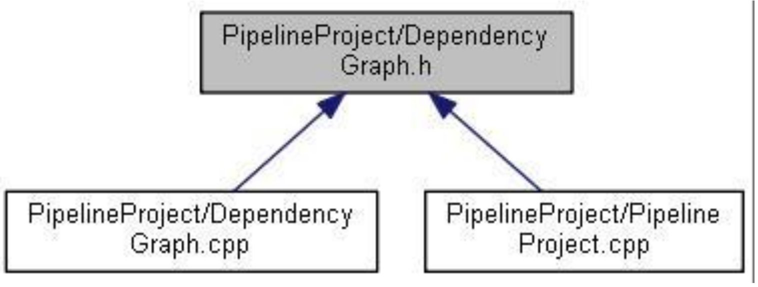
CDependencyGraph class interface.

```
#include "CommonDef.h"
#include <set>
#include <vector>
```

Include dependency graph for DependencyGraph.h:



This graph shows which files directly or indirectly include this file:


PipelineProject/Dependency Graph.h
PipelineProject/Dependency Graph.cpp
PipelineProject/Pipeline Project.cpp

Classes

- class CDirectedEdgeData
- *Maintains directed edge data properties.* class CGraphNode
- *a directed graph node implementation.* class CDependencyGraph

A directed acyclic graph implementation. Typedefs

- typedef TCHAR NODE_ID_T

Variables

- const NODE_ID_T INVALID_NODE_ID = 0
  Used to identify an active node.
- const size_t INVALID_NODE_INDEX = static_cast<size_t>(-1)
  used to provide a consistent index out-of-range result

Detailed Description

CDependencyGraph class interface.

Author:

Mark L. Short

Date:

November 23, 2014

Generally, a graph consists of:

- a set of nodes (or vertices)
- a set of edges (or arc)

Directed Acyclic Graphs (DAG) - http://en.wikipedia.org/wiki/Directed_acyclic_graph

Like most scheduling problems, instruction scheduling is usually modelled as a directed acyclic graph (DAG) evaluation problem. Each node in the data dependency graph represents a single machine instruction, and each arc represents a dependency with a weight corresponding to the latency of the relevant instruction.

See also:

http://www.lighterra.com/papers/basicinstructionscheduling/

In order to construct a DAG to represent the dependencies between instructions:

- For each instruction, create a corresponding vertex in the graph
- For each dependency between two instructions, create a corresponding edge in the graph
- This edge is directed : it goes from the earlier instruction to the later one

Typedef Documentation

typedef TCHAR NODE_ID_T

Variable Documentation

const NODE_ID_T INVALID_NODE_ID = 0

Used to identify an active node.

Referenced by CGraphNode::IsValid().

const size_t INVALID_NODE_INDEX = static_cast<size_t>(-1)

used to provide a consistent index out-of-range result

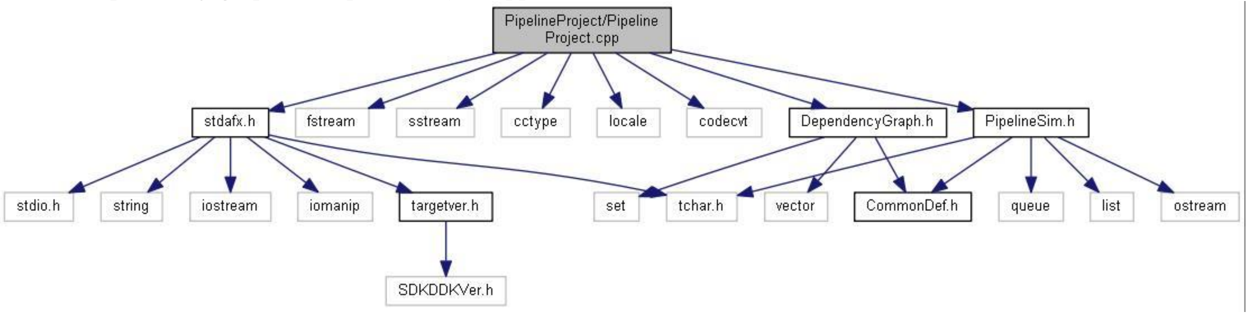
Referenced by CDependencyGraph::GetNodeIndex().

PipelineProject/PipelineProject.cpp File Reference

Main source file for implementation of pipeline project simulation.

```
#include "stdafx.h"
#include <fstream>
#include <sstream>
#include <cctype>
#include <locale>
#include <codecvt>
#include "DependencyGraph.h"
#include "PipelineSim.h"
```

Include dependency graph for PipelineProject.cpp:



Macros

- #define tifstream   std::ifstream
- #define tstringstream   std::stringstream

Functions

- int CalculateSequentialExecutionCycles (const CDependencyGraph &dag)
  CalculateSequentialExecutionCycles calculates the number of cycles required to "sequentially" execute a set of instructions.
- int CalculateCompleteOverlappedExecutionCycles (const CDependencyGraph &dag)
  CalculateCompleteOverlappedExecutionCycles calculates the best case execution scenario in terms of minimum number of cycles required to execute the set of instructions.
- int CalculatePartialOverlappedExecutionCycles (const CDependencyGraph &dag)
  CalculatePartialOverlappedExecutionCycles computes the number of cycles required to execute a set of instruction using a 4-staged pipeline and factoring in delays introduced to address instruction-level data dependencies.
- int CalculateNumberOfStallsRequired (const CDependencyGraph &dag)
  CalculateNumberOfStallsRequired calculates data-dependent pipeline stalls.
- bool ExecutePipelineSimulation (CPipelineSim &sim, const CDependencyGraph &dag)
  Performs basic pipeline process simulation.
- size_t LoadData (const TCHAR *szFileName, CDependencyGraph &dag)
  LoadData performs basic file level data input.
- int _tmain (int argc, _TCHAR *argv[])

Variables

- const int MAX_INSTRUCTIONS = 25
  maximum instructions specified
- const int BASE_CYCLES_PER_INSTUCTION = 4
  non-overlapped cycles required to execute 1 instruction in a 4 staged-pipeline
- const TCHAR g_szFileName [] = _T("InstructionInputData.txt")
  File used to read in test case data.
- CDependencyGraph g_DAG (MAX_INSTRUCTIONS)
  Global directed acyclic graph object.
- CPipelineSim g_PipelineSim
  Global pipeline simulation object.

Detailed Description

Main source file for implementation of pipeline project simulation.

Macro Definition Documentation

#define tifstream   std::ifstream

Referenced by LoadData().

#define tstringstream   std::stringstream

Referenced by LoadData().

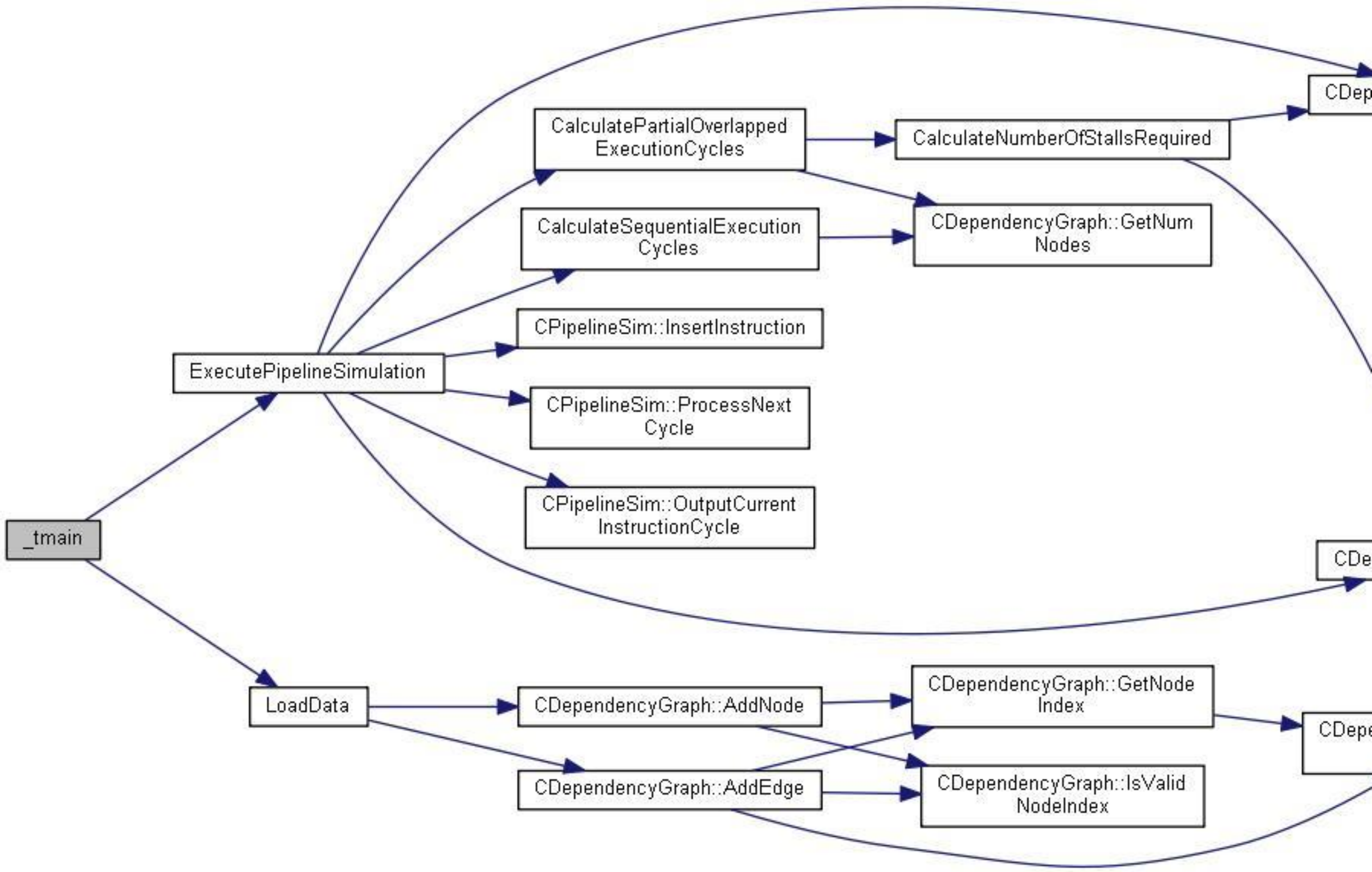
Function Documentation

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

References ExecutePipelineSimulation(), g_DAG, g_szFileName, and LoadData().

```
{
    LoadData(g_szFileName, g_DAG);

    ExecutePipelineSimulation(g_PipelineSim, g_DAG);

    return 0;
}
```

Here is the call graph for this function:


_tmain
ExecutePipelineSimulation
LoadData
CalculatePartialOverlapped ExecutionCycles
CalculateSequentialExecution Cycles
CPipelineSim::InsertInstruction
CPipelineSim::ProcessNext Cycle
CPipelineSim::OutputCurrent InstructionCycle
CalculateNumberOfStallsRequired
CDependencyGraph::GetNum Nodes
CDep
CDe
CDependencyGraph::AddNode
CDependencyGraph::AddEdge
CDependencyGraph::GetNode Index
CDependencyGraph::IsValid NodeIndex
CDepe

int CalculateCompleteOverlappedExecutionCycles (const CDependencyGraph & *dag*)

CalculateCompleteOverlappedExecutionCycles calculates the best case execution scenario in terms of minimum number of cycles required to execute the set of instructions.

The basic formula to calculate the execution cycles required to run N instructions in a 4 staged pipeline is something like:

1. Calculate number of cycles to execute 1st instruction, in this case it is 4 cycles.
2. Then, based on the fact that a subsequent instruction will complete every cycle from cycle 4 on, for N number of instructions, therefore it will take **N + 3 cycles** to run the entire set of N instructions.

Parameters:

in	*dag*	DAG object containing a list of instructions

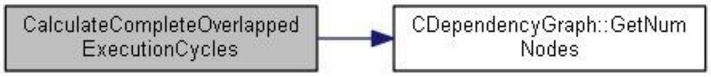
Return values:

int	the number of overlapped cycles (with no delays) required to run the instructions contained in DAG

References CDependencyGraph::GetNumNodes().

```
{
    return dag.GetNumNodes ( ) + 3;
}
```

Here is the call graph for this function:


CalculateCompleteOverlapped ExecutionCycles
CDependencyGraph::GetNum Nodes

int CalculateNumberOfStallsRequired (const CDependencyGraph & *dag*)

CalculateNumberOfStallsRequired calculates data-dependent pipeline stalls.

This function identifies and calculates the number Pipeline stalls introduced into the pipeline to avoid data-dependency hazards in the pipelined execution of the set of instructions.

Parameters:

in	*dag*	DAG object containing a list of instructions

Return values:

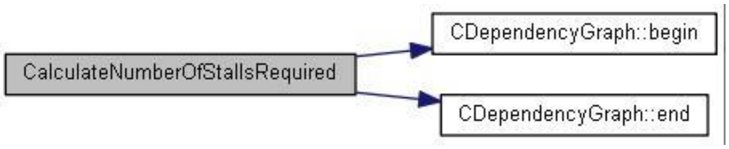
int	the number pipeline stalls required to address instruction data dependencies identified in a the DAGi

References CDependencyGraph::begin(), and CDependencyGraph::end().

Referenced by CalculatePartialOverlappedExecutionCycles().

```
{
    int iNumStalls = 0;

    CDependencyGraph::const_iterator it;

    for (it = dag.begin(); it != dag.end(); ++ it )
    {
        if (it->IsValid())
        {
            NODE_ID_T idNode = it->GetNodeID();
            // following will determine if an instruction is dependent on
            // an immediately previous one, (i.e B->A), which is the only
            // case that any stall is required to be introduced.
            if (it->HasEdge(idNode - 1))
                iNumStalls++;
        }
    }

    return iNumStalls;
}
```

Here is the call graph for this function:



int CalculatePartialOverlappedExecutionCycles (const CDependencyGraph & *dag*)

CalculatePartialOverlappedExecutionCycles computes the number of cycles required to execute a set of instruction using a 4-staged pipeline and factoring in delays introduced to address instruction-level data dependencies.

The basic formula needed to calculate the execution cycles required to run N instructions in a 4 staged pipeline is :

1. Use the calculation from the formula above to determine the minimum number of cycles required. In this case it is: **N + 3 cycles** .
2. Then add 1 cycle for each delay introduced.

Considering the initial program data provided in this assignment :

1. 6 instructions to be executed.Minimum execution time is N + 3 cycles, or 9 cycles in this case.
2. 2 bubbles or stalls were introduced due to data - dependencies.
3. 9 cycles + 2 cycles (for the stalls) = 11 cycles. This is the same result as reported on the assignment.
4. So the resultant formula to calculate the number of cycles for a partial overlapped pipelined execution of N instructions given M stalls introduced is : **N + 3 + M cycles**

So, how do we find the number of stalls required to be introduced due to data dependencies? In this scenario given only a 4 cycle "data hazard" window of opportunity, and considering data reads occur in the 3rd stage (EX), while data writes are only accessible after WB (to be interpreted as the 5th stage), that further narrows the "data hazard window" down to 2 cycles. The only way a data hazard could occur is if there is a data dependency between two immediately sequential instructions.

It is deduced that the very worst possible case of extreme data dependency requiring a stall for every instruction would only require at most:

N + 3 cycles for the instructions, plus another **N - 1 cycles** for adding 1 bubble / stall cycle for every instruction after the 1st.

Therefore **(2 * N) + 2 cycles** would be the worst possible number of cycles required to run any set of instructions overlapped.

Parameters:

in	*dag*	DAG object containing a list of instructions

Return values:

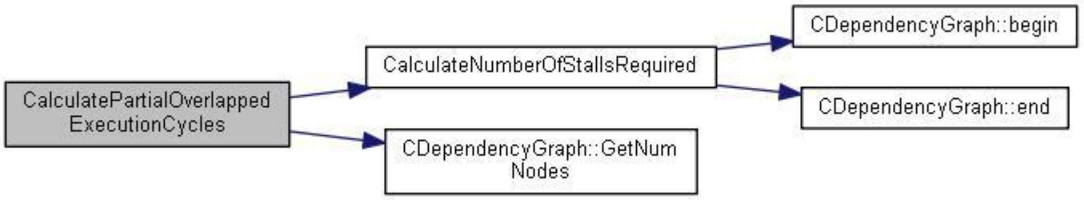
int	the number of overlapped cycles (with delays) required to run the instructions contained in DAG

References CalculateNumberOfStallsRequired(), and CDependencyGraph::GetNumNodes().

Referenced by ExecutePipelineSimulation().

```
{
    return dag.GetNumNodes() + CalculateNumberOfStallsRequired(dag) + 3;
}
```

Here is the call graph for this function:


CalculatePartialOverlapped ExecutionCycles
CalculateNumberOfStallsRequired
CDependencyGraph::begin
CDependencyGraph::end
CDependencyGraph::GetNum Nodes

int CalculateSequentialExecutionCycles (const CDependencyGraph & *dag*)

CalculateSequentialExecutionCycles calculates the number of cycles required to "sequentially" execute a set of instructions.

The basic formula to calculate the execution cycles required to run N instructions sequentially (non-overlapped) in this scenario is: **N * 4 cycles**

Parameters:

in	*dag*	DAG object containing a list of instructions

Return values:

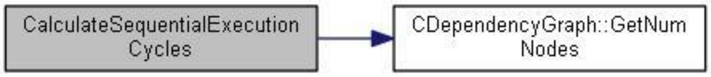
int	the number sequential cycles required to run the instructions contained in DAG

References BASE_CYCLES_PER_INSTUCTION, and CDependencyGraph::GetNumNodes().

Referenced by ExecutePipelineSimulation().

```
{
    return dag.GetNumNodes ( ) * BASE_CYCLES_PER_INSTUCTION;
}
```

Here is the call graph for this function:


CalculateSequentialExecution Cycles
CDependencyGraph::GetNum Nodes

bool ExecutePipelineSimulation (CPipelineSim & *sim*, const CDependencyGraph & *dag*)

Performs basic pipeline process simulation.

ExecutePipelineSimulation takes instruction data contain in an DAG and feeds it to the simulation object for running of the instruction pipeline simulation

Parameters:

in,out	*sim*	Simulation object
in	*dag*	DAG object containing a list of instructions

Return values:

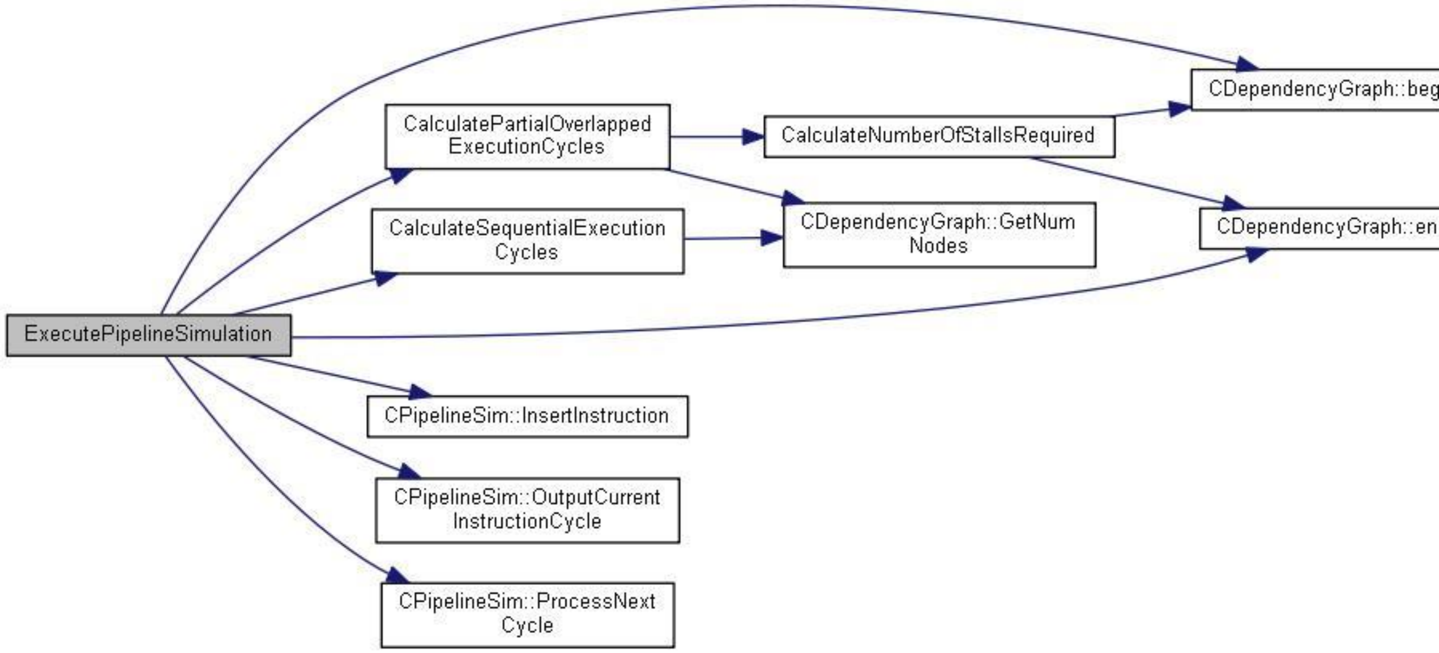
true	on success
false	on error

References CDependencyGraph::begin(), CalculatePartialOverlappedExecutionCycles(), CalculateSequentialExecutionCycles(), CDependencyGraph::end(), CPipelineSim::InsertInstruction(), CPipelineSim::OutputCurrentInstructionCycle(), CPipelineSim::ProcessNextCycle(), and tcout.

Referenced by _tmain().

```
{
    bool bReturn = false;

    // add the loaded instructions to the pipeline simulator
    CDependencyGraph::const_iterator it;

    for ( it = dag.begin ( ); it != dag.end ( ); ++it )
    {
        if ( it->IsValid ( ) )
        {
            bool bDataDependent = false;
            NODE_ID_T idNode = it->GetNodeID ( );

            if ( it->HasEdge ( idNode - 1 ) )
                bDataDependent = true;

            sim.InsertInstruction(CInstructionData(idNode, bDataDependent) );
        }
    }


    tcout << _T ( "Total time for sequential (non overlapped) execution: " )
          << CalculateSequentialExecutionCycles ( dag ) << _T ( " cycles" ) << std::endl;
    tcout << _T ("------------------------------------------------------------------")
          << std::endl;
    tcout << _T ( "Overlapped execution:" ) << std::endl;

    bool bMoreInstructions = sim.ProcessNextCycle();

    while (bMoreInstructions)
    {
        sim.OutputCurrentInstructionCycle(tcout);

        bMoreInstructions = sim.ProcessNextCycle();
    }

    tcout << _T ( "------------------------------------------------------------------")
          << std::endl;
    tcout << _T ( "Total time for pipelined (overlapped) execution: " )
          << CalculatePartialOverlappedExecutionCycles ( dag ) << _T ( " cycles" ) <<
std::endl;

    return bReturn;
}
```

Here is the call graph for this function:


ExecutePipelineSimulation
CalculatePartialOverlapped ExecutionCycles
CalculateSequentialExecution Cycles
CalculateNumberOfStallsRequired
CDependencyGraph::GetNum Nodes
CDependencyGraph::beg
CDependencyGraph::en
CPipelineSim::InsertInstruction
CPipelineSim::OutputCurrent InstructionCycle
CPipelineSim::ProcessNext Cycle

size_t LoadData (const TCHAR * *szFileName*, CDependencyGraph & *dag*)

LoadData performs basic file level data input.

This method reads input data from text file and returns contents in a directed acyclic graph

Parameters:

in	*szFileName*	name of the data file to be loaded
out	*dag*	reference to a dag object

Return values:

size_t	the number of item nodes read into the graph

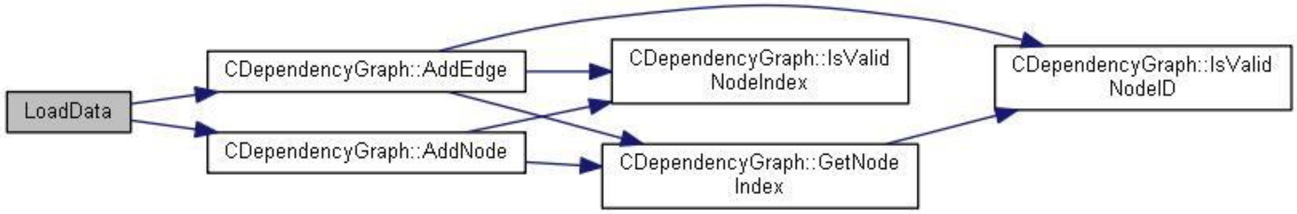
References CDependencyGraph::AddEdge(), CDependencyGraph::AddNode(), MAX_INSTRUCTIONS, tcout, tifstream, and tstringstream.

Referenced by _tmain().

```
{
    size_t nReturn = 0;

    tifstream infile;

#if defined(UNICODE) || defined(_UNICODE)
    std::locale utf8_locale ( std::locale ( infile.getloc ( ) ), new
std::codecvt_utf8_utf16<wchar_t> );
    infile.imbue ( utf8_locale );
#endif

```

```
    infile.open ( szFileName );

    if ( infile.bad ( ) )
    {
        tcout << _T ( "Error opening data file:" ) << szFileName << std::endl;
    }
    else
    {
        TCHAR szLineBuffer[128] = { 0 };

        // read in the 1st line of input, this will contain the list of instructions
        infile.getline ( szLineBuffer, _countof ( szLineBuffer ) - 1 );

        // parse the instruction list, removing trailing punctuation.
        tstringstream strStream;

        strStream << szLineBuffer;

        while ( strStream.getline ( szLineBuffer, 5, _T ( ' ' ) ) && ( nReturn < MAX_INSTRUCTIONS ) )
        {
            dag.AddNode ( szLineBuffer[0] );

            nReturn++;
        }

        // now parse the instruction dependencies
        // this will be in the format of:
        //      B<space>A<NL>
        // where "B A" means that B depends on the result of A

        TCHAR idSrcNode = 0;
        TCHAR idDestNode = 0;

        while ( infile >> idSrcNode >> idDestNode )
        {
            // in estimating an edge weight, lets use the time delta or "dependency
            // distance" between when the 2 instructions are scheduled to begin execution.

            int iWeight = idSrcNode - idDestNode;

            dag.AddEdge ( idSrcNode, idDestNode, iWeight );
        }

        infile.close ( );
    }

    return nReturn;
}
```

Here is the call graph for this function:


LoadData
CDependencyGraph::AddEdge
CDependencyGraph::AddNode
CDependencyGraph::IsValid NodeIndex
CDependencyGraph::GetNode Index
CDependencyGraph::IsValid NodeID

Variable Documentation

const int BASE_CYCLES_PER_INSTUCTION = 4

non-overlapped cycles required to execute 1 instruction in a 4 staged-pipeline

Referenced by CalculateSequentialExecutionCycles().

CDependencyGraph g_DAG(MAX_INSTRUCTIONS)

Global directed acyclic graph object.

Referenced by _tmain().

CPipelineSim g_PipelineSim

Global pipeline simulation object.

const TCHAR g_szFileName[] = _T("InstructionInputData.txt")

File used to read in test case data.

Referenced by _tmain().

const int MAX_INSTRUCTIONS = 25

maximum instructions specified

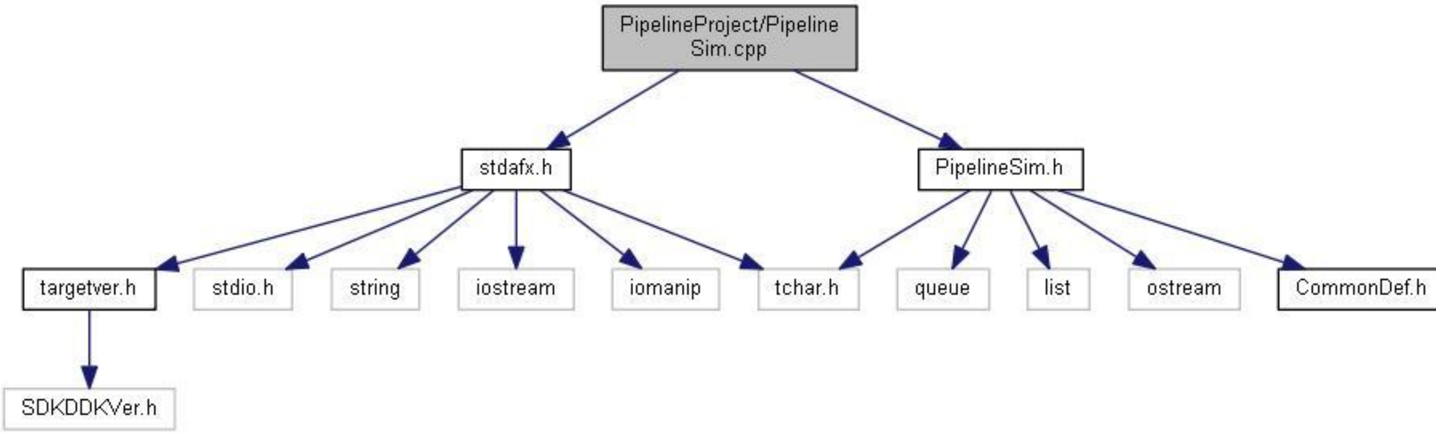
Referenced by LoadData().

PipelineProject/PipelineSim.cpp File Reference

CPipelineSim class implementation.

```
#include "stdafx.h"
#include "PipelineSim.h"
```

Include dependency graph for PipelineSim.cpp:



Typedefs

- typedef std::list< CInstructionData >::iterator LstIterator
- typedef std::list< CInstructionData >::reverse_iterator rLstIterator

Variables

- const int CONCURRENT_INSTRUCTION_LIMIT = 4

Detailed Description

CPipelineSim class implementation.

Author:

Mark L. Short

Date:

November 23, 2014

Typedef Documentation

typedef std::list<CInstructionData>::iterator LstIterator

typedef std::list<CInstructionData>::reverse_iterator rLstIterator

Variable Documentation

const int CONCURRENT_INSTRUCTION_LIMIT = 4

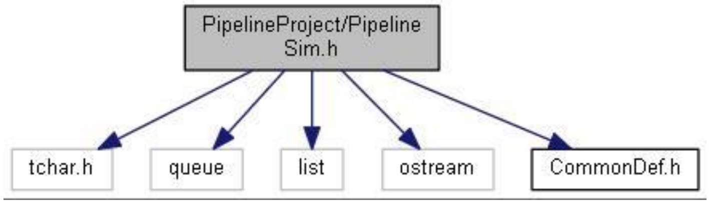
four-stage pipeline only allows concurrent processing of four instructions at a time.

PipelineProject/PipelineSim.h File Reference

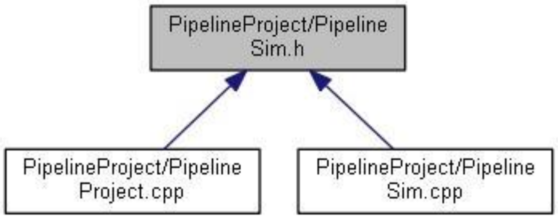
CPipelineSim class interface.

```
#include <tchar.h>
#include <queue>
#include <list>
#include <ostream>
#include "CommonDef.h"
```

Include dependency graph for PipelineSim.h:



This graph shows which files directly or indirectly include this file:


PipelineProject/Pipeline Sim.h
PipelineProject/Pipeline Project.cpp
PipelineProject/Pipeline Sim.cpp

Classes

- class CInstructionData
- *Instruction data and state.* class CNoopInstruction
- class CPipelineSim

A 4-staged pipeline simulation class. Macros

- #define __PIPELINE_SIM_H__
- #define tostream   std::ostream

Typedefs

- typedef enum PS_PIPELINE_STATE PS_PIPELINE_STATE_T
  Pipeline Instruction State.
- typedef TCHAR INSTRUCTION_T

Enumerations

- enum PS_PIPELINE_STATE { PS_INVALID, PS_IF, PS_ID, PS_EX, PS_WB, PS_COMPLETED }

Pipeline Instruction State. Variables

- const INSTRUCTION_T INVALID_INSTRUCTION = 0
  used to denote an uninitialized instruction

- const INSTRUCTION_T NOOP_INSTRUCTION = '-'

Detailed Description

CPipelineSim class interface.

Author:

Mark L. Short

Date:

November 24, 2014

Macro Definition Documentation

#define _PIPELINE_SIM_H__

#define tostream   std::ostream

Typedef Documentation

typedef TCHAR INSTRUCTION_T

In a more sophisticated simulation, the following would contain the actual instruction to be processed (either as a string or binary opcode); however, in this instance it is only a single letter ('a'..'y').

typedef enum PS_PIPELINE_STATE   PS_PIPELINE_STATE_T

Pipeline Instruction State.

Enumeration Type Documentation

enum PS_PIPELINE_STATE

Pipeline Instruction State.

Enumerator

PS_INVALID initial default state

PS_IF Instruction Fetch state

PS_ID Instruction Decode state

PS_EX Execute state

PS_WB Write Back state

PS_COMPLETED instruction processing completed

```
{
    PS_INVALID,
    PS_IF,
    PS_ID,
    PS_EX,
    PS_WB,
    PS_COMPLETED
} PS_PIPELINE_STATE_T;
```

Variable Documentation

const INSTRUCTION_T INVALID_INSTRUCTION = 0

used to denote an uninitialized instruction

const INSTRUCTION_T NOOP_INSTRUCTION = '-'

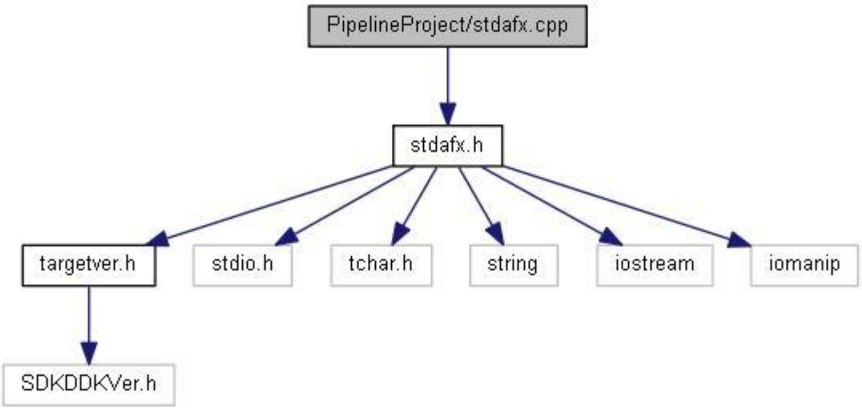
Referenced by CInstructionData::IsNOOP().

PipelineProject/stdafx.cpp File Reference

Source file that includes just the standard includes.

```
#include "stdafx.h"
```

Include dependency graph for stdafx.cpp:



Detailed Description

Source file that includes just the standard includes.

PipelineProject.pch will be the pre-compiled header stdafx.obj will contain the pre-compiled type information

Author:

Mark L. Short

Date:

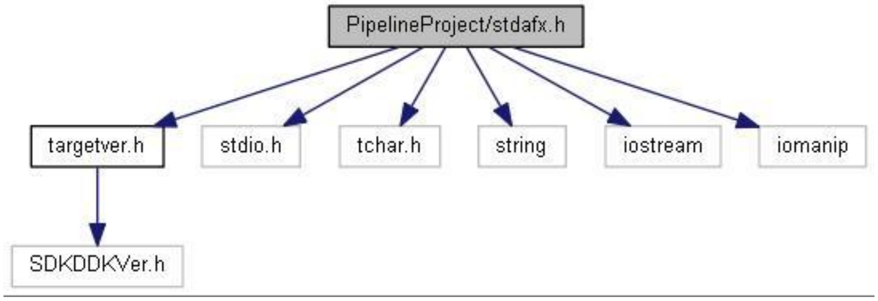
November 25, 2014

PipelineProject/stdafx.h File Reference

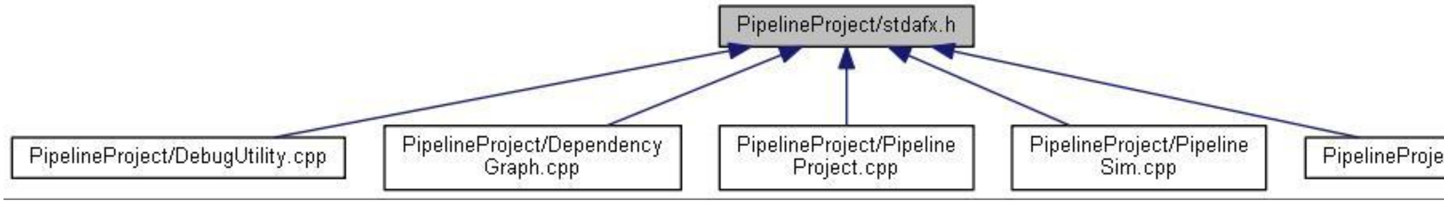
Application header file.

```
#include "targetver.h"
#include <stdio.h>
#include <tchar.h>
#include <string>
#include <iostream>
#include <iomanip>
```

Include dependency graph for stdafx.h:



This graph shows which files directly or indirectly include this file:


PipelineProject/stdafx.h
PipelineProject/DebugUtility.cpp
PipelineProject/Dependency Graph.cpp
PipelineProject/Pipeline Project.cpp
PipelineProject/Pipeline Sim.cpp
PipelineProje

Macros

- #define _CRT_SECURE_NO_WARNINGS
- #define tcout   std::cout
- #define tstring   std::string

Detailed Description

Application header file.

Include file for standard system include header files, or project specific include files that are used frequently, but are changed infrequently

Author:

Mark L. Short

Date:

Oct 30, 2014

Macro Definition Documentation

#define _CRT_SECURE_NO_WARNINGS

#define tcout   std::cout

Referenced by ExecutePipelineSimulation(), and LoadData().

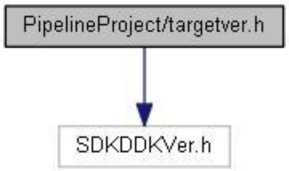
#define tstring   std::string

PipelineProject/targetver.h File Reference

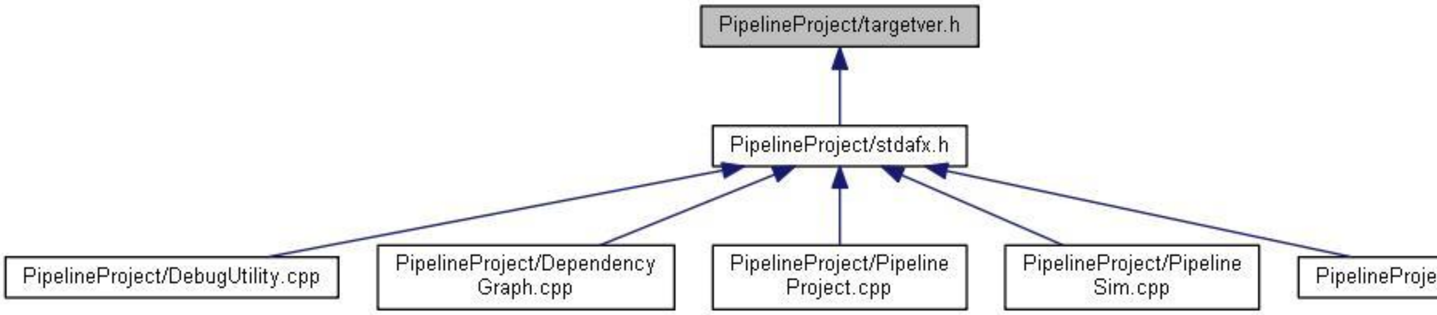
Windows OS platform header file.

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

Include dependency graph for targetver.h:



This graph shows which files directly or indirectly include this file:


PipelineProject/targetver.h
PipelineProject/stdafx.h
PipelineProject/DebugUtility.cpp
PipelineProject/Dependency Graph.cpp
PipelineProject/Pipeline Project.cpp
PipelineProject/Pipeline Sim.cpp
PipelineProje

Detailed Description

Windows OS platform header file.

Author:

Mark L. Short

Date:

November 25, 2014

Index