Advanced Computer Architecture

CMPS 5133

Mark L. Short,

Kessia Eugene,

Kishore Lav Mutyala

Version 1

# Table of Contents

Main Page 2

Hierarchical Index 3

Class Index 4

File Index 5

Class Documentation 6

CDependencyGraph 6

CDirectedEdgeData 14

CGraphNode 17

CInstructionData 22

CNoopInstruction 25

CPipelineSim 26

File Documentation 32

PipelineProject/CommonDef.h 32

PipelineProject/DebugUtility.cpp 34

PipelineProject/DebugUtility.h 36

PipelineProject/DependencyGraph.cpp 38

PipelineProject/DependencyGraph.h 39

PipelineProject/PipelineProject.cpp 41

PipelineProject/PipelineSim.cpp 50

PipelineProject/PipelineSim.h 52

PipelineProject/stdafx.cpp 55

PipelineProject/stdafx.h 56

PipelineProject/targetver.h 58

Index 59

# 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:

CDependencyGraph

CDirectedEdgeData

CGraphNode

CInstructionData

CNoopInstruction

CPipelineSim

# Class Index

## Class List

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

[**CDependencyGraph**](#AAAAAAAAAL) **(A directed acyclic graph implementation )**

[**CDirectedEdgeData**](#AAAAAAAAAO) **(Maintains directed edge data properties )**

[**CGraphNode**](#AAAAAAAAAP) **(Directed graph node implementation )**

[**CInstructionData**](#AAAAAAAABO) **(Instruction data and state )**

[**CNoopInstruction**](#AAAAAAAABZ)

[**CPipelineSim**](#AAAAAAAABB) **(A 4-staged pipeline simulation class )**

# File Index

## File List

Here is a list of all files with brief descriptions:

**PipelineProject/**[**CommonDef.h**](#AAAAAAAAAA) **(Common type definitions )**

**PipelineProject/**[**DebugUtility.cpp**](#AAAAAAAAAE) **(Implementation of** [**DebugUtility.cpp**](#AAAAAAAAAE) **)**

**PipelineProject/**[**DebugUtility.h**](#AAAAAAAAAH) **(Debugging and utility method declarations )**

**PipelineProject/**[**DependencyGraph.cpp**](#AAAAAAAAAK) **(**[**CDependencyGraph**](#AAAAAAAAAL) **class implementation )**

**PipelineProject/**[**DependencyGraph.h**](#AAAAAAAAAN) **(**[**CDependencyGraph**](#AAAAAAAAAL) **class interface )**

**PipelineProject/**[**PipelineProject.cpp**](#AAAAAAAAAT) **(Main source file for implementation of pipeline project simulation )**

**PipelineProject/**[**PipelineSim.cpp**](#AAAAAAAABU) **(**[**CPipelineSim**](#AAAAAAAABB) **class implementation )**

**PipelineProject/**[**PipelineSim.h**](#AAAAAAAABY) **(**[**CPipelineSim**](#AAAAAAAABB) **class interface )**

**PipelineProject/**[**stdafx.cpp**](#AAAAAAAACN) **(Source file that includes just the standard includes )**

**PipelineProject/**[**stdafx.h**](#AAAAAAAACO) **(Application header file )**

**PipelineProject/**[**targetver.h**](#AAAAAAAACR) **(Windows OS platform header file )**

# Class Documentation

## CDependencyGraph Class Reference

A directed acyclic graph implementation.

#include <DependencyGraph.h>

Collaboration diagram for CDependencyGraph:

### Public Types

* typedef std::vector
* < [CGraphNode](#AAAAAAAAAP) >::[const\_iterator](#AAAAAAAABK) [const\_iterator](#AAAAAAAABK)

### Public Member Functions

* [CDependencyGraph](#AAAAAAAACS) ()

*Default Constructor.*

* [CDependencyGraph](#AAAAAAAACT) (size\_t nMaxNodes)

*Init Constructor.*

* bool [AddNode](#AAAAAAAABS) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idNode)

*This method adds a new node to the graph.*

* bool [AddEdge](#AAAAAAAABT) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idFromNode, const [NODE\_ID\_T](#AAAAAAAAAQ) &idToNode, int iWeight)

*Adds a directed edge between 2 existing nodes.*

* size\_t [GetNumNodes](#AAAAAAAABJ) (void) const

*Retrieves the current number of nodes in the graph.*

* size\_t [GetNumEdges](#AAAAAAAACU) (void) const

*Retrieves the current number of edges in the graph.*

* bool [HasNode](#AAAAAAAACV) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idNode) const

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

* [const\_iterator](#AAAAAAAABK) [begin](#AAAAAAAABL) (void) const

*Affords iteration functionality.*

* [const\_iterator](#AAAAAAAABK) [end](#AAAAAAAABM) (void) const

*Affords iteration functionality.*

* [~CDependencyGraph](#AAAAAAAACW) ()

*Default Destructor.*

### Private Member Functions

* bool [IsValidNodeID](#AAAAAAAACX) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idNode) const

*Performs basic validation of node ID.*

* bool [IsValidNodeIndex](#AAAAAAAACY) (size\_t nIndex) const

*Performs basic validation of a node index.*

* size\_t [GetNodeIndex](#AAAAAAAACZ) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idNode) const

*returns corresponding node index*

* [CDependencyGraph](#AAAAAAAADA) (const [CDependencyGraph](#AAAAAAAAAL) &o)

*copy constructor*

* [CDependencyGraph](#AAAAAAAAAL) & [operator=](#AAAAAAAADB) (const [CDependencyGraph](#AAAAAAAAAL) &rhs)

*assignment operator*

### Private Attributes

* size\_t [m\_nMaxNodes](#AAAAAAAADC)

*maintains an upper limit on nodes allowed in the graph*

* size\_t [m\_nNumNodes](#AAAAAAAADD)

*maintains a current number of nodes in the graph*

* std::vector< [CGraphNode](#AAAAAAAAAP) > [m\_vNodes](#AAAAAAAADE)

### Detailed Description

A directed acyclic graph implementation.

The [CDependencyGraph](#AAAAAAAAAL) 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](#AAAAAAAAAP)>::[const\_iterator](#AAAAAAAABK) [CDependencyGraph::const\_iterator](#AAAAAAAABK)

### Constructor & Destructor Documentation

#### CDependencyGraph::CDependencyGraph ()

Default Constructor.

40 : [m\_nMaxNodes](#AAAAAAAADC)([DEFAULT\_MAX\_NODES](#AAAAAAAAAM)),

41 [m\_nNumNodes](#AAAAAAAADD)(0),

42 [m\_vNodes](#AAAAAAAADE)([DEFAULT\_MAX\_NODES](#AAAAAAAAAM))

43 {

44 }

#### 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. |

47 : [m\_nMaxNodes](#AAAAAAAADC)(nMaxNodes),

48 [m\_nNumNodes](#AAAAAAAADD)(0),

49 [m\_vNodes](#AAAAAAAADE)(nMaxNodes)

50 {

51 }

#### CDependencyGraph::~CDependencyGraph ()

Default Destructor.

172 {

173 }

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

Copy constructor

### Member Function Documentation

#### bool CDependencyGraph::AddEdge (const [NODE\_ID\_T](#AAAAAAAAAQ) & *idFromNode*, const [NODE\_ID\_T](#AAAAAAAAAQ) & *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().

78 {

79 bool bReturn = false;

80

81 // lets validate the input data

82 if ( [IsValidNodeID](#AAAAAAAACX)(idFromNode) && [IsValidNodeID](#AAAAAAAACX)(idToNode) )

83 {

84 size\_t nIndex = [GetNodeIndex](#AAAAAAAACZ)(idFromNode);

85

86 if ( [IsValidNodeIndex](#AAAAAAAACY)(nIndex) )

87 bReturn = [m\_vNodes](#AAAAAAAADE)[nIndex].AddEdge (idToNode, iWeight);

88 }

89

90

91 return bReturn;

92 }

Here is the call graph for this function:

#### bool CDependencyGraph::AddNode (const [NODE\_ID\_T](#AAAAAAAAAQ) & *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().

54 {

55 bool bReturn = false;

56

57 size\_t nNodeIndex = [GetNodeIndex](#AAAAAAAACZ)(idNode);

58

59 // check to make sure index is not out of bounds of our vector

60 if ( [IsValidNodeIndex](#AAAAAAAACY)(nNodeIndex) )

61 {

62 // check to make sure we have not already added this node

63 if ( [m\_vNodes](#AAAAAAAADE)[nNodeIndex].IsValid() == false )

64 {

65 // node has not been previously added, lets update

66 // the node ID with a valid ID to mark it has been

67 // added now.

68 [m\_vNodes](#AAAAAAAADE)[nNodeIndex].SetNodeID(idNode);

69 [m\_nNumNodes](#AAAAAAAADD)++;

70 bReturn = true;

71 }

72 }

73

74 return bReturn;

75 }

Here is the call graph for this function:

#### [const\_iterator](#AAAAAAAABK) 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().

380 { return [m\_vNodes](#AAAAAAAADE).begin ( ); };

#### [const\_iterator](#AAAAAAAABK) 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().

392 { return [m\_vNodes](#AAAAAAAADE).end ( ); };

#### size\_t CDependencyGraph::GetNodeIndex (const [NODE\_ID\_T](#AAAAAAAAAQ) & *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().

140 {

141 size\_t nReturn = [INVALID\_NODE\_INDEX](#AAAAAAAAAS);

142

143 if ( [IsValidNodeID](#AAAAAAAACX)(idNode) )

144 {

145 const std::locale& loc = std::locale(); // construct from the default locale

146 TCHAR cId = static\_cast<TCHAR>(idNode);

147 nReturn = std::tolower(cId, loc) - \_T('a');

148 }

149

150 return nReturn;

151 }

Here is the call graph for this function:

#### 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.

102 {

103 size\_t nNumEdges = 0;

104

105 // following static cast added to address compiler warning

106 for (int i = 0; i < static\_cast<int>([m\_nMaxNodes](#AAAAAAAADC)); i++)

107 {

108 nNumEdges += [m\_vNodes](#AAAAAAAADE)[i].GetNumEdges();

109 }

110

111 return nNumEdges;

112 }

#### 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().

96 {

97 return [m\_nNumNodes](#AAAAAAAADD);

98 }

#### bool CDependencyGraph::HasNode (const [NODE\_ID\_T](#AAAAAAAAAQ) & *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.

154 {

155 bool bReturn = false;

156

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

158

159 size\_t nIndex = [GetNodeIndex](#AAAAAAAACZ)(idNode);

160

161 if ( [IsValidNodeIndex](#AAAAAAAACY)(nIndex) )

162 {

163 // now determine if that node associated with the index has

164 // been added to the graph or not

165 bReturn = [m\_vNodes](#AAAAAAAADE)[nIndex].IsValid();

166 }

167

168 return bReturn;

169 };

Here is the call graph for this function:

#### bool CDependencyGraph::IsValidNodeID (const [NODE\_ID\_T](#AAAAAAAAAQ) & *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().

115 {

116 bool bReturn = false;

117 // we are only allowing up to 25 instructions, each represented by a char in

118 // in the range of [a..y]. lets allow some flexibility by being case-insensitive.

119 if ( (idNode >= 'a' && idNode <= 'y') || (idNode >= 'A' && idNode <= 'Y') )

120 bReturn = true;

121

122 return bReturn;

123 }

#### bool CDependencyGraph::IsValidNodeIndex (size\_t *nIndex*) const[private]

Performs basic validation of a node index.

The index 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().

126 {

127 bool bReturn = false;

128

129 if ( nIndex < [m\_nMaxNodes](#AAAAAAAADC) )

130 bReturn = true;

131

132 return bReturn;

133 }

#### [CDependencyGraph](#AAAAAAAAAL)& CDependencyGraph::operator= (const [CDependencyGraph](#AAAAAAAAAL) & *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](#AAAAAAAAAP)> 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](#AAAAAAAAAN)
* PipelineProject/[DependencyGraph.cpp](#AAAAAAAAAK)

## CDirectedEdgeData Class Reference

Maintains directed edge data properties.

#include <DependencyGraph.h>

### Public Member Functions

* [CDirectedEdgeData](#AAAAAAAADF) ()

*Default Constructor.*

* [CDirectedEdgeData](#AAAAAAAADG) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idToNode, int iWeight=0)

*Initialization Constructor.*

* void [SetNodeID](#AAAAAAAADH) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idSet)

*Sets the value or ID of the edge destination node.*

* [NODE\_ID\_T](#AAAAAAAAAQ) [GetDestNodeID](#AAAAAAAADI) (void) const

*Retrieves the value or ID of the edge destination node.*

* void [SetWeight](#AAAAAAAADJ) (int iSet)

*Sets the weight value associated with this edge.*

* int [GetWeight](#AAAAAAAADK) (void) const

*Retrieves the current weight value associated with this edge.*

* bool [operator<](#AAAAAAAADL) (const [CDirectedEdgeData](#AAAAAAAAAO) &rhs) const

*Comparison operation required by STL.*

* bool [operator==](#AAAAAAAADM) (const [CDirectedEdgeData](#AAAAAAAAAO) &rhs) const

*Comparison operation required by STL.*

* bool [operator>](#AAAAAAAADN) (const [CDirectedEdgeData](#AAAAAAAAAO) &rhs) const

*Comparison operation required by STL.*

### Private Attributes

* [NODE\_ID\_T](#AAAAAAAAAQ) [m\_idDestNode](#AAAAAAAADO)

*Maintains the destination node ID*

* int [m\_iWeight](#AAAAAAAADP)

*Weight value assigned to this edge*

### Detailed Description

Maintains directed edge data properties.

Additionally, [CDirectedEdgeData](#AAAAAAAAAO) 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.

64 : [m\_idDestNode](#AAAAAAAADO) ([INVALID\_NODE\_ID](#AAAAAAAAAR)), [m\_iWeight](#AAAAAAAADP)(0)

65 {};

#### CDirectedEdgeData::CDirectedEdgeData (const [NODE\_ID\_T](#AAAAAAAAAQ) & *idToNode*, int *iWeight* = 0)

Initialization Constructor.

69 : [m\_idDestNode](#AAAAAAAADO)(idToNode), [m\_iWeight](#AAAAAAAADP)(iWeight)

70 {};

### Member Function Documentation

#### [NODE\_ID\_T](#AAAAAAAAAQ) 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.

87 { return [m\_idDestNode](#AAAAAAAADO); };

#### int CDirectedEdgeData::GetWeight (void ) const

Retrieves the current weight value associated with this edge.

##### Return values:

|  |  |
| --- | --- |
| *int* | the current edge weight |

References m\_iWeight.

103 { return [m\_iWeight](#AAAAAAAADP); };

#### bool CDirectedEdgeData::operator< (const [CDirectedEdgeData](#AAAAAAAAAO) & *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.

114 { return [m\_idDestNode](#AAAAAAAADO) < rhs.[m\_idDestNode](#AAAAAAAADO); };

#### bool CDirectedEdgeData::operator== (const [CDirectedEdgeData](#AAAAAAAAAO) & *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.

125 { return [m\_idDestNode](#AAAAAAAADO) == rhs.[m\_idDestNode](#AAAAAAAADO); };

#### bool CDirectedEdgeData::operator> (const [CDirectedEdgeData](#AAAAAAAAAO) & *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.

135 { return [m\_idDestNode](#AAAAAAAADO) > rhs.[m\_idDestNode](#AAAAAAAADO); };

#### void CDirectedEdgeData::SetNodeID (const [NODE\_ID\_T](#AAAAAAAAAQ) & *idSet*)

Sets the value or ID of the edge destination node.

##### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *idSet* | the destination node ID to be set |

78 { [m\_idDestNode](#AAAAAAAADO) = idSet; };

#### void CDirectedEdgeData::SetWeight (int *iSet*)

Sets the weight value associated with this edge.

##### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *iSet* | new weight value to be set |

95 { [m\_iWeight](#AAAAAAAADP) = iSet; };

### Member Data Documentation

#### [NODE\_ID\_T](#AAAAAAAAAQ) 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](#AAAAAAAAAN)

## 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](#AAAAAAAADQ)

### Public Member Functions

* [CGraphNode](#AAAAAAAADR) ()

*Default Constructor.*

* [CGraphNode](#AAAAAAAADS) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idNode)

*Initialization Constructor.*

* void [SetNodeID](#AAAAAAAADT) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idSet)

*Sets this object's node ID.*

* [NODE\_ID\_T](#AAAAAAAAAQ) [GetNodeID](#AAAAAAAADU) (void) const

*Retrieves this object's node ID.*

* bool [IsValid](#AAAAAAAADV) (void) const

*Used to check to see if this node is active.*

* bool [AddEdge](#AAAAAAAADW) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idToNode, int iWeight)

*Adds a new edge to the graph*

* bool [AddEdge](#AAAAAAAADX) (const [CDirectedEdgeData](#AAAAAAAAAO) &edge)

*Adds a new edge the graph.*

* size\_t [GetNumEdges](#AAAAAAAADY) (void) const

*Retrieves number of edges.*

* [const\_iterator](#AAAAAAAADQ) [beginEdge](#AAAAAAAADZ) (void) const

*Affords iterator functionality.*

* [const\_iterator](#AAAAAAAADQ) [endEdge](#AAAAAAAAEA) (void) const

*Affords iterator functionality.*

* bool [HasEdge](#AAAAAAAAEB) (const [NODE\_ID\_T](#AAAAAAAAAQ) &idToNode) const

*Test if given edge exists.*

* [~CGraphNode](#AAAAAAAAEC) ()

*Default Destructor.*

### Private Types

* typedef std::set
* < [CDirectedEdgeData](#AAAAAAAAAO) > [EDGE\_SET\_T](#AAAAAAAAED)
* typedef EDGE\_SET\_T::\_Pairib [\_Pairib](#AAAAAAAAEE)

### Private Attributes

* [NODE\_ID\_T](#AAAAAAAAAQ) [m\_ID](#AAAAAAAAEF)

*this is the node value or ID*

* [EDGE\_SET\_T](#AAAAAAAAED) [m\_setEdges](#AAAAAAAAEG)

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

### Detailed Description

A directed graph node implementation.

The [CGraphNode](#AAAAAAAAAP) class maintains a node ID, as well as a set of [CDirectedEdgeData](#AAAAAAAAAO) 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](#AAAAAAAAEE)[private]

#### typedef EDGE\_SET\_T::const\_iterator [CGraphNode::const\_iterator](#AAAAAAAADQ)

#### typedef std::set<[CDirectedEdgeData](#AAAAAAAAAO)> [CGraphNode::EDGE\_SET\_T](#AAAAAAAAED)[private]

### Constructor & Destructor Documentation

#### CGraphNode::CGraphNode ()

Default Constructor.

159 : [m\_ID](#AAAAAAAAEF)([INVALID\_NODE\_ID](#AAAAAAAAAR)), [m\_setEdges](#AAAAAAAAEG)()

160 { };

#### CGraphNode::CGraphNode (const [NODE\_ID\_T](#AAAAAAAAAQ) & *idNode*)

Initialization Constructor.

164 : [m\_ID](#AAAAAAAAEF)(idNode), [m\_setEdges](#AAAAAAAAEG)()

165 { };

#### CGraphNode::~CGraphNode ()

Default Destructor.

283 { };

### Member Function Documentation

#### bool CGraphNode::AddEdge (const [NODE\_ID\_T](#AAAAAAAAAQ) & *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 |

16 {

17 return [AddEdge](#AAAAAAAADW)( [CDirectedEdgeData](#AAAAAAAAAO)(idToNode, iWeight) );

18 }

#### bool CGraphNode::AddEdge (const [CDirectedEdgeData](#AAAAAAAAAO) & *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.

21 {

22 bool bReturn = false;

23

24 if ( [IsValid](#AAAAAAAADV) ( ) ) // check to make sure we are a valid node before assigning any edges

25 {

26 [\_Pairib](#AAAAAAAAEE) Result = [m\_setEdges](#AAAAAAAAEG).insert ( edge );

27

28 if ( Result.second ) // was the insert successful ?

29 {

30 bReturn = true;

31 }

32 }

33

34 return bReturn;

35 }

Here is the call graph for this function:

#### [const\_iterator](#AAAAAAAADQ) 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 |

252 { return [m\_setEdges](#AAAAAAAAEG).begin(); };

#### [const\_iterator](#AAAAAAAADQ) 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 |

264 { return [m\_setEdges](#AAAAAAAAEG).end(); };

#### [NODE\_ID\_T](#AAAAAAAAAQ) 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.

182 { return [m\_ID](#AAAAAAAAEF); };

#### 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().

240 { return [IsValid](#AAAAAAAADV)() ? [m\_setEdges](#AAAAAAAAEG).size() : 0; };

Here is the call graph for this function:

#### bool CGraphNode::HasEdge (const [NODE\_ID\_T](#AAAAAAAAAQ) & *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 |

276 {

277 [const\_iterator](#AAAAAAAADQ) itr = [m\_setEdges](#AAAAAAAAEG).find( [CDirectedEdgeData](#AAAAAAAAAO)(idToNode) );

278 return (itr != [m\_setEdges](#AAAAAAAAEG).end() );

279 };

#### 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().

195 { return [m\_ID](#AAAAAAAAEF) != [INVALID\_NODE\_ID](#AAAAAAAAAR); };

#### void CGraphNode::SetNodeID (const [NODE\_ID\_T](#AAAAAAAAAQ) & *idSet*)

Sets this object's node ID.

##### Parameters:

|  |  |  |
| --- | --- | --- |
| in | *idSet* | new node ID to be set |

173 { [m\_ID](#AAAAAAAAEF) = idSet; };

### Member Data Documentation

#### [NODE\_ID\_T](#AAAAAAAAAQ) CGraphNode::m\_ID[private]

This is the node value or ID

Referenced by GetNodeID().

#### [EDGE\_SET\_T](#AAAAAAAAED) 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](#AAAAAAAAAN)
* PipelineProject/[DependencyGraph.cpp](#AAAAAAAAAK)

## CInstructionData Class Reference

Instruction data and state.

#include <PipelineSim.h>

Inheritance diagram for CInstructionData:

### Public Member Functions

* [CInstructionData](#AAAAAAAAEH) ()

*Default Constructor.*

* [CInstructionData](#AAAAAAAAEI) (const [INSTRUCTION\_T](#AAAAAAAACE) &instruction, bool bDataDependent)

*Initialization Constructor.*

* [CInstructionData](#AAAAAAAAEJ) (const [INSTRUCTION\_T](#AAAAAAAACE) &instruction, [PS\_PIPELINE\_STATE](#AAAAAAAACC) psState=[PS\_INVALID](#AAAAAAAACF), bool bDataDependent=false)

*Initialization Constructor.*

* [INSTRUCTION\_T](#AAAAAAAACE) [GetInstruction](#AAAAAAAAEK) (void) const

*Retrieves the instruction.*

* [PS\_PIPELINE\_STATE](#AAAAAAAACC) [GetState](#AAAAAAAAEL) (void) const

*Gets the instruction pipeline state.*

* void [SetState](#AAAAAAAAEM) ([PS\_PIPELINE\_STATE](#AAAAAAAACC) psSet)

*Set the instruction pipeline state.*

* bool [IsDataDependent](#AAAAAAAAEN) (void) const
* void [SetDataDependent](#AAAAAAAAEO) (bool bSet=true)
* bool [IsNOOP](#AAAAAAAAEP) (void) const
* [~CInstructionData](#AAAAAAAAEQ) ()

*Destructor.*

### Private Attributes

* [INSTRUCTION\_T](#AAAAAAAACE) [m\_Instruction](#AAAAAAAAER)
* [PS\_PIPELINE\_STATE](#AAAAAAAACC) [m\_psState](#AAAAAAAAES)
* bool [m\_bDataDependent](#AAAAAAAAET)

### Detailed Description

Instruction data and state.

### Constructor & Destructor Documentation

#### CInstructionData::CInstructionData ()

Default Constructor.

78 : [m\_Instruction](#AAAAAAAAER)([INVALID\_INSTRUCTION](#AAAAAAAACL)),

79 [m\_psState](#AAAAAAAAES)([PS\_INVALID](#AAAAAAAACF)),

80 [m\_bDataDependent](#AAAAAAAAET)(false)

81 {};

#### CInstructionData::CInstructionData (const [INSTRUCTION\_T](#AAAAAAAACE) & *instruction*, bool *bDataDependent*)

Initialization Constructor.

84 : [m\_Instruction](#AAAAAAAAER) ( instruction ),

85 [m\_psState](#AAAAAAAAES) ( [PS\_INVALID](#AAAAAAAACF) ),

86 [m\_bDataDependent](#AAAAAAAAET) ( bDataDependent )

87 {};

#### CInstructionData::CInstructionData (const [INSTRUCTION\_T](#AAAAAAAACE) & *instruction*, [PS\_PIPELINE\_STATE](#AAAAAAAACC) *psState* = [PS\_INVALID](#AAAAAAAACF), bool *bDataDependent* = false)

Initialization Constructor.

91 : [m\_Instruction](#AAAAAAAAER) ( instruction ),

92 [m\_psState](#AAAAAAAAES) ( psState ),

93 [m\_bDataDependent](#AAAAAAAAET)(bDataDependent)

94 {};

#### CInstructionData::~CInstructionData ()

Destructor.

132 {};

### Member Function Documentation

#### [INSTRUCTION\_T](#AAAAAAAACE) CInstructionData::GetInstruction (void ) const

Retrieves the instruction.

##### Return values:

|  |  |
| --- | --- |
| *INSTRUCTION\_T* |  |

References m\_Instruction.

102 { return [m\_Instruction](#AAAAAAAAER); };

#### [PS\_PIPELINE\_STATE](#AAAAAAAACC) 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.

115 { return [m\_psState](#AAAAAAAAES); };

#### bool CInstructionData::IsDataDependent (void ) const

References m\_bDataDependent.

123 { return [m\_bDataDependent](#AAAAAAAAET); };

#### bool CInstructionData::IsNOOP (void ) const

References NOOP\_INSTRUCTION.

129 { return [m\_Instruction](#AAAAAAAAER) == [NOOP\_INSTRUCTION](#AAAAAAAACM); };

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

126 { [m\_bDataDependent](#AAAAAAAAET) = bSet; };

#### void CInstructionData::SetState ([PS\_PIPELINE\_STATE](#AAAAAAAACC) *psSet*)

Set the instruction pipeline state.

120 { [m\_psState](#AAAAAAAAES) = psSet; };

### Member Data Documentation

#### bool CInstructionData::m\_bDataDependent[private]

Referenced by IsDataDependent().

#### [INSTRUCTION\_T](#AAAAAAAACE) CInstructionData::m\_Instruction[private]

Referenced by GetInstruction().

#### [PS\_PIPELINE\_STATE](#AAAAAAAACC) CInstructionData::m\_psState[private]

Referenced by GetState().

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

* PipelineProject/[PipelineSim.h](#AAAAAAAABY)

## CNoopInstruction Class Reference

#include <PipelineSim.h>

Inheritance diagram for CNoopInstruction:

Collaboration diagram for CNoopInstruction:

### Public Member Functions

* [CNoopInstruction](#AAAAAAAAEU) ()
* [CNoopInstruction](#AAAAAAAAEV) ([PS\_PIPELINE\_STATE](#AAAAAAAACC) psState)

### Constructor & Destructor Documentation

#### CNoopInstruction::CNoopInstruction ()

139 : [CInstructionData](#AAAAAAAAEH)([NOOP\_INSTRUCTION](#AAAAAAAACM))

140 { };

#### CNoopInstruction::CNoopInstruction ([PS\_PIPELINE\_STATE](#AAAAAAAACC) *psState*)

143 : [CInstructionData](#AAAAAAAAEH)([NOOP\_INSTRUCTION](#AAAAAAAACM), psState)

144 { };

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

* PipelineProject/[PipelineSim.h](#AAAAAAAABY)

## CPipelineSim Class Reference

A 4-staged pipeline simulation class.

#include <PipelineSim.h>

Collaboration diagram for CPipelineSim:

### Public Member Functions

* [CPipelineSim](#AAAAAAAAEW) ()

*Default Constructor.*

* [DWORD](#AAAAAAAAAD) [GetCycle](#AAAAAAAAEX) (void) const

*Retrieves the current number of cycles executed.*

* [DWORD](#AAAAAAAAAD) [GetStallCount](#AAAAAAAAEY) (void) const

*Retrieves the count of stalls introduced into the pipeline.*

* [DWORD](#AAAAAAAAAD) [GetCompletionCount](#AAAAAAAAEZ) (void) const

*Retrieves the current count of completed instructions.*

* bool [ProcessNextCycle](#AAAAAAAABQ) (void)

*Process next pipeline instruction cycle.*

* size\_t [InsertInstruction](#AAAAAAAABN) (const [CInstructionData](#AAAAAAAABO) &instruction)

*Adds the instruction to the instruction queue.*

* [tostream](#AAAAAAAACB) & [OutputCurrentInstructionCycle](#AAAAAAAABR) ([tostream](#AAAAAAAACB) &os)

*formats and outputs current pipelined instructions to the provided stream*

* [~CPipelineSim](#AAAAAAAAFA) ()

*Destructor.*

### Private Attributes

* [DWORD](#AAAAAAAAAD) [m\_dwCycle](#AAAAAAAAFB)

*Maintains current pipeline cycle*

* [DWORD](#AAAAAAAAAD) [m\_dwStallCtr](#AAAAAAAAFC)

*A count of the stalls introduced*

* [DWORD](#AAAAAAAAAD) [m\_dwCompletedCtr](#AAAAAAAAFD)

*A count of the instructions completed execution*

* [DWORD](#AAAAAAAAAD) [m\_dwMaxPipelineDepth](#AAAAAAAAFE)

*Limit on instructions in the pipeline*

* std::list< [CInstructionData](#AAAAAAAABO) > [m\_lstInstructionPipeline](#AAAAAAAAFF)

*Our instruction pipeline*

* std::queue< [CInstructionData](#AAAAAAAABO) > [m\_queInstructions](#AAAAAAAAFG)

*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.

24 : [m\_dwCycle](#AAAAAAAAFB)(0),

25 [m\_dwStallCtr](#AAAAAAAAFC)(0),

26 [m\_dwCompletedCtr](#AAAAAAAAFD)(0),

27 [m\_dwMaxPipelineDepth](#AAAAAAAAFE) ( [CONCURRENT\_INSTRUCTION\_LIMIT](#AAAAAAAABX) ),

28 [m\_lstInstructionPipeline](#AAAAAAAAFF)(),

29 [m\_queInstructions](#AAAAAAAAFG)()

30 {

31 }

#### CPipelineSim::~CPipelineSim ()

Destructor.

171 {

172 }

### Member Function Documentation

#### [DWORD](#AAAAAAAAAD) CPipelineSim::GetCompletionCount (void ) const

Retrieves the current count of completed instructions.

##### Return values:

|  |  |
| --- | --- |
| *DWORD* | count of completed instructions |

References m\_dwCompletedCtr.

198 { return [m\_dwCompletedCtr](#AAAAAAAAFD); };

#### [DWORD](#AAAAAAAAAD) CPipelineSim::GetCycle (void ) const

Retrieves the current number of cycles executed.

##### Return values:

|  |  |
| --- | --- |
| *DWORD* | current cycle |

References m\_dwCycle.

182 { return [m\_dwCycle](#AAAAAAAAFB); };

#### [DWORD](#AAAAAAAAAD) CPipelineSim::GetStallCount (void ) const

Retrieves the count of stalls introduced into the pipeline.

##### Return values:

|  |  |
| --- | --- |
| *DWORD* | count of stalls |

References m\_dwStallCtr.

190 { return [m\_dwStallCtr](#AAAAAAAAFC); };

#### size\_t CPipelineSim::InsertInstruction (const [CInstructionData](#AAAAAAAABO) & *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().

142 {

143 [m\_queInstructions](#AAAAAAAAFG).push(instruction);

144

145 return [m\_queInstructions](#AAAAAAAAFG).size();

146 };

#### [tostream](#AAAAAAAACB) & CPipelineSim::OutputCurrentInstructionCycle ([tostream](#AAAAAAAACB) & *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().

149 {

150 for ( [LstIterator](#AAAAAAAABV) it = [m\_lstInstructionPipeline](#AAAAAAAAFF).begin ( ); it != [m\_lstInstructionPipeline](#AAAAAAAAFF).end ( ); ++it )

151 {

152 switch (it->GetState())

153 {

154 case [PS\_IF](#AAAAAAAACG):

155 case [PS\_ID](#AAAAAAAACH):

156 case [PS\_EX](#AAAAAAAACI):

157 case [PS\_WB](#AAAAAAAACJ):

158 os << it->GetInstruction() << \_T(" ");

159 break;

160 default:

161 break;

162 }

163 }

164

165 os << std::endl;

166

167 return os;

168 }

#### 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().

34 {

35 bool bReturn = false;

36 // increment the cycle counter

37 [m\_dwCycle](#AAAAAAAAFB)++;

38

39 // Begin processing our instruction que

40 // check our current instruction pipeline size and see if we have room

41 if ( [m\_lstInstructionPipeline](#AAAAAAAAFF).size ( ) <= [m\_dwMaxPipelineDepth](#AAAAAAAAFE) )

42 {

43 // check our instruction queue and see if we have anything left to execute

44

45 if ([m\_queInstructions](#AAAAAAAAFG).size() != 0)

46 {

47 [CInstructionData](#AAAAAAAABO) instruction = [m\_queInstructions](#AAAAAAAAFG).front();

48

49 [m\_queInstructions](#AAAAAAAAFG).pop();

50

51 // insert the instruction at the beginning of our pipeline

52 [m\_lstInstructionPipeline](#AAAAAAAAFF).push\_front ( instruction );

53

54 bReturn = true;

55 }

56 else

57 {

58 // nothing left in the instruction que,

59 // so we insert NOOPS until everything

60 // clears the pipeline

61 [CNoopInstruction](#AAAAAAAABZ) NOOP;

62

63 [m\_lstInstructionPipeline](#AAAAAAAAFF).push\_front ( NOOP );

64 }

65 }

66

67 bool bStalled = false;

68 // reverse iterate over the instruction currently in the pipeline

69 for ([rLstIterator](#AAAAAAAABW) itr = [m\_lstInstructionPipeline](#AAAAAAAAFF).rbegin(); itr != [m\_lstInstructionPipeline](#AAAAAAAAFF).rend() && (bStalled == false); ++itr)

70 {

71 [PS\_PIPELINE\_STATE](#AAAAAAAACC) stInstruction = itr->GetState();

72

73 switch (stInstruction)

74 {

75

76 case [PS\_INVALID](#AAAAAAAACF): // initial default state

77 itr->SetState([PS\_IF](#AAAAAAAACG));

78 if ( itr->IsNOOP ( ) == false )

79 bReturn = true;

80 break;

81

82 case [PS\_IF](#AAAAAAAACG): // Instruction Fetch state

83 itr->SetState([PS\_ID](#AAAAAAAACH));

84 if ( itr->IsNOOP ( ) == false )

85 bReturn = true;

86 break;

87

88 case [PS\_ID](#AAAAAAAACH): // Instruction Decode state

89 // need to verify if a dependency exists between this instruction

90 // and the immediately previous instruction

91

92 if (itr->IsDataDependent())

93 {

94 // we have to introduce a stall here

95 itr->SetDataDependent(false);

96

97 [CNoopInstruction](#AAAAAAAABZ) NOOP([PS\_EX](#AAAAAAAACI));

98

99 [m\_lstInstructionPipeline](#AAAAAAAAFF).insert(itr.base(), NOOP);

100

101 bStalled = true;

102 [m\_dwStallCtr](#AAAAAAAAFC)++;

103 }

104 else

105 {

106 itr->SetState([PS\_EX](#AAAAAAAACI));

107 }

108

109 if (itr->IsNOOP() == false)

110 bReturn = true;

111 break;

112

113 case [PS\_EX](#AAAAAAAACI): // Instruction Execute state

114 itr->SetState([PS\_WB](#AAAAAAAACJ));

115 if (itr->IsNOOP() == false)

116 bReturn = true;

117 break;

118

119 case [PS\_WB](#AAAAAAAACJ): // Instruction Write Back state

120 itr->SetState ([PS\_COMPLETED](#AAAAAAAACK)); // mark this for removal later

121

122 if (itr->IsNOOP() == false)

123 [m\_dwCompletedCtr](#AAAAAAAAFD)++;

124

125 break;

126

127 case [PS\_COMPLETED](#AAAAAAAACK):

128 default:

129

130 break;

131 }

132 }

133

134 // check to see if we have a completed instruction for removal from the pipeline

135 if ([m\_lstInstructionPipeline](#AAAAAAAAFF).back().GetState() == [PS\_COMPLETED](#AAAAAAAACK))

136 [m\_lstInstructionPipeline](#AAAAAAAAFF).pop\_back();

137

138 return bReturn;

139 };

### Member Data Documentation

#### [DWORD](#AAAAAAAAAD) CPipelineSim::m\_dwCompletedCtr[private]

A count of the instructions completed execution

Referenced by GetCompletionCount(), and ProcessNextCycle().

#### [DWORD](#AAAAAAAAAD) CPipelineSim::m\_dwCycle[private]

Maintains current pipeline cycle

Referenced by GetCycle(), and ProcessNextCycle().

#### [DWORD](#AAAAAAAAAD) CPipelineSim::m\_dwMaxPipelineDepth[private]

Limit on instructions in the pipeline

Referenced by ProcessNextCycle().

#### [DWORD](#AAAAAAAAAD) CPipelineSim::m\_dwStallCtr[private]

A count of the stalls introduced

Referenced by GetStallCount(), and ProcessNextCycle().

#### std::list<[CInstructionData](#AAAAAAAABO)> CPipelineSim::m\_lstInstructionPipeline[private]

Our instruction pipeline

Referenced by OutputCurrentInstructionCycle(), and ProcessNextCycle().

#### std::queue<[CInstructionData](#AAAAAAAABO)> 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](#AAAAAAAABY)
* PipelineProject/[PipelineSim.cpp](#AAAAAAAABU)

# File Documentation

## PipelineProject/CommonDef.h File Reference

Common type definitions.

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

### Macros

* #define [\_COMMON\_DEF\_H\_\_](#AAAAAAAAAB)

### Typedefs

* typedef unsigned \_\_int8 [BYTE](#AAAAAAAAAC)

*8-bit unsigned type*

* typedef unsigned \_\_int32 [DWORD](#AAAAAAAAAD)

*32-bit unsigned type*

### Detailed Description

Common type definitions.

##### Author:

Mark L. Short

### Typedef Documentation

#### typedef unsigned \_\_int8 [BYTE](#AAAAAAAAAC)

8-bit unsigned type

#### typedef unsigned \_\_int32 [DWORD](#AAAAAAAAAD)

32-bit unsigned type

## PipelineProject/DebugUtility.cpp File Reference

Implementation of [DebugUtility.cpp](#AAAAAAAAAE).

#include "stdafx.h"

#include "stdlib.h"

#include "stdarg.h"

#include "Windows.h"

#include "DebugUtility.h"

Include dependency graph for DebugUtility.cpp:

### Functions

* int [DebugTrace](#AAAAAAAAAF) (const TCHAR \*szFmt,...)

*Directs output to the IDE output window.*

* TCHAR \* [GetModulePath](#AAAAAAAAAG) (TCHAR \*szModulePath, size\_t cchLen)

*Retrieves the current executable directory.*

### Detailed Description

Implementation of [DebugUtility.cpp](#AAAAAAAAAE).

##### 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. |

27 {

28 TCHAR szDebugMsg[512] = { 0 };

29

30 va\_list vaArgs;

31 va\_start (vaArgs, szFmt);

32

33 // use the format string and arguments to construct the debug output string

34 int iReturnVal = \_vsntprintf (szDebugMsg, \_countof (szDebugMsg) - 1, szFmt, vaArgs);

35 va\_end (vaArgs);

36

37 ::OutputDebugString (szDebugMsg);

38 return iReturnVal;

39

40 }

#### 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 |

43 {

44 // Get the executable file path

45 TCHAR szModuleFileName[\_MAX\_PATH] = { 0 };

46

47 // Note, if HANDLE is NULL, GetModuleFileName is supposed to return the file path to the

48 // current executable, but it appears that it is inconsistently returning filename as

49 // well....

50 [DWORD](#AAAAAAAAAD) dwStrLen = ::GetModuleFileName (NULL, szModuleFileName, \_countof(szModuleFileName) );

51

52 TCHAR szDir[\_MAX\_PATH] = {0};

53

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

55

56 return \_tcsncpy(szModulePath, szDir, cchLen);

57 }

## 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:

### Functions

* int [DebugTrace](#AAAAAAAAAI) (const TCHAR \*szFmt,...)

*Directs output to the IDE output window.*

* TCHAR \* [GetModulePath](#AAAAAAAAAJ) (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. |

27 {

28 TCHAR szDebugMsg[512] = { 0 };

29

30 va\_list vaArgs;

31 va\_start (vaArgs, szFmt);

32

33 // use the format string and arguments to construct the debug output string

34 int iReturnVal = \_vsntprintf (szDebugMsg, \_countof (szDebugMsg) - 1, szFmt, vaArgs);

35 va\_end (vaArgs);

36

37 ::OutputDebugString (szDebugMsg);

38 return iReturnVal;

39

40 }

#### 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 |

43 {

44 // Get the executable file path

45 TCHAR szModuleFileName[\_MAX\_PATH] = { 0 };

46

47 // Note, if HANDLE is NULL, GetModuleFileName is supposed to return the file path to the

48 // current executable, but it appears that it is inconsistently returning filename as

49 // well....

50 [DWORD](#AAAAAAAAAD) dwStrLen = ::GetModuleFileName (NULL, szModuleFileName, \_countof(szModuleFileName) );

51

52 TCHAR szDir[\_MAX\_PATH] = {0};

53

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

55

56 return \_tcsncpy(szModulePath, szDir, cchLen);

57 }

## PipelineProject/DependencyGraph.cpp File Reference

[CDependencyGraph](#AAAAAAAAAL) class implementation.

#include "stdafx.h"

#include "DependencyGraph.h"

#include <locale>

Include dependency graph for DependencyGraph.cpp:

### Variables

* const size\_t [DEFAULT\_MAX\_NODES](#AAAAAAAAAM) = 10

### Detailed Description

[CDependencyGraph](#AAAAAAAAAL) 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](#AAAAAAAAAL) 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:

### Classes

* class [CDirectedEdgeData](#AAAAAAAAAO)
* *Maintains directed edge data properties.* class [CGraphNode](#AAAAAAAAAP)
* *a directed graph node implementation.* class [CDependencyGraph](#AAAAAAAAAL)

### *A directed acyclic graph implementation.* Typedefs

* typedef TCHAR [NODE\_ID\_T](#AAAAAAAAAQ)

### Variables

* const [NODE\_ID\_T](#AAAAAAAAAQ) [INVALID\_NODE\_ID](#AAAAAAAAAR) = 0

*Used to identify an active node.*

* const size\_t [INVALID\_NODE\_INDEX](#AAAAAAAAAS) = static\_cast<size\_t>(-1)

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

### Detailed Description

[CDependencyGraph](#AAAAAAAAAL) 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](#AAAAAAAAAQ)

### Variable Documentation

#### const [NODE\_ID\_T](#AAAAAAAAAQ) 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:

### Functions

* int [CalculateSequentialExecutionCycles](#AAAAAAAAAW) (const [CDependencyGraph](#AAAAAAAAAL) &dag)

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

* int [CalculateCompleteOverlappedExecutionCycles](#AAAAAAAAAX) (const [CDependencyGraph](#AAAAAAAAAL) &dag)

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

* int [CalculatePartialOverlappedExecutionCycles](#AAAAAAAAAY) (const [CDependencyGraph](#AAAAAAAAAL) &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](#AAAAAAAAAZ) (const [CDependencyGraph](#AAAAAAAAAL) &dag)

*CalculateNumberOfStallsRequired calculates data-dependent pipeline stalls.*

* bool [ExecutePipelineSimulation](#AAAAAAAABA) ([CPipelineSim](#AAAAAAAABB) &sim, const [CDependencyGraph](#AAAAAAAAAL) &dag)

*Performs basic pipeline process simulation.*

* size\_t [LoadData](#AAAAAAAABC) (const TCHAR \*szFileName, [CDependencyGraph](#AAAAAAAAAL) &dag)

*LoadData performs basic file level data input.*

* int [\_tmain](#AAAAAAAABD) (int argc, \_TCHAR \*argv[])

### Variables

* const int [MAX\_INSTRUCTIONS](#AAAAAAAABE) = 25

*Maximum instructions specified*

* const int [BASE\_CYCLES\_PER\_INSTUCTION](#AAAAAAAABF) = 4

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

* const TCHAR [g\_szFileName](#AAAAAAAABG) [] = \_T("InstructionInputData.txt")

*File used to read in test case data.*

* [CDependencyGraph](#AAAAAAAAAL) [g\_DAG](#AAAAAAAABH) ([MAX\_INSTRUCTIONS](#AAAAAAAABE))

*Global directed acyclic graph object.*

* [CPipelineSim](#AAAAAAAABB) [g\_PipelineSim](#AAAAAAAABI)

*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().

220 {

221 [LoadData](#AAAAAAAABC)([g\_szFileName](#AAAAAAAABG), [g\_DAG](#AAAAAAAABH));

222

223 [ExecutePipelineSimulation](#AAAAAAAABA)([g\_PipelineSim](#AAAAAAAABI), [g\_DAG](#AAAAAAAABH));

224

225 return 0;

226 }

Here is the call graph for this function:

#### int CalculateCompleteOverlappedExecutionCycles (const [CDependencyGraph](#AAAAAAAAAL) & *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().

294 {

295 return dag.[GetNumNodes](#AAAAAAAABJ) ( ) + 3;

296 }

Here is the call graph for this function:

#### int CalculateNumberOfStallsRequired (const [CDependencyGraph](#AAAAAAAAAL) & *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().

304 {

305 int iNumStalls = 0;

306

307 [CDependencyGraph::const\_iterator](#AAAAAAAABK) it;

308

309 for (it = dag.[begin](#AAAAAAAABL)(); it != dag.[end](#AAAAAAAABM)(); ++ it )

310 {

311 if (it->IsValid())

312 {

313 [NODE\_ID\_T](#AAAAAAAAAQ) idNode = it->GetNodeID();

314 // following will determine if an instruction is dependent on

315 // an immediately previous one, (i.e B->A), which is the only

316 // case that any stall is required to be introduced.

317 if (it->HasEdge(idNode - 1))

318 iNumStalls++;

319 }

320 }

321

322 return iNumStalls;

323 }

Here is the call graph for this function:

#### int CalculatePartialOverlappedExecutionCycles (const [CDependencyGraph](#AAAAAAAAAL) & *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().

299 {

300 return dag.[GetNumNodes](#AAAAAAAABJ)() + [CalculateNumberOfStallsRequired](#AAAAAAAAAZ)(dag) + 3;

301 }

Here is the call graph for this function:

#### int CalculateSequentialExecutionCycles (const [CDependencyGraph](#AAAAAAAAAL) & *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().

289 {

290 return dag.[GetNumNodes](#AAAAAAAABJ) ( ) \* [BASE\_CYCLES\_PER\_INSTUCTION](#AAAAAAAABF);

291 }

Here is the call graph for this function:

#### bool ExecutePipelineSimulation ([CPipelineSim](#AAAAAAAABB) & *sim*, const [CDependencyGraph](#AAAAAAAAAL) & *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().

326 {

327 bool bReturn = false;

328

329 // add the loaded instructions to the pipeline simulator

330 [CDependencyGraph::const\_iterator](#AAAAAAAABK) it;

331

332 for ( it = dag.[begin](#AAAAAAAABL) ( ); it != dag.[end](#AAAAAAAABM) ( ); ++it )

333 {

334 if ( it->IsValid ( ) )

335 {

336 bool bDataDependent = false;

337 [NODE\_ID\_T](#AAAAAAAAAQ) idNode = it->GetNodeID ( );

338

339 if ( it->HasEdge ( idNode - 1 ) )

340 bDataDependent = true;

341

342 sim.[InsertInstruction](#AAAAAAAABN)([CInstructionData](#AAAAAAAABO)(idNode, bDataDependent) );

343 }

344 }

345

346

347 [tcout](#AAAAAAAABP) << \_T ( "Total time for sequential (non overlapped) execution: " )

348 << [CalculateSequentialExecutionCycles](#AAAAAAAAAW) ( dag ) << \_T ( " cycles" ) << std::endl;

349 [tcout](#AAAAAAAABP) << \_T ("------------------------------------------------------------------")

350 << std::endl;

351 [tcout](#AAAAAAAABP) << \_T ( "Overlapped execution:" ) << std::endl;

352

353 bool bMoreInstructions = sim.[ProcessNextCycle](#AAAAAAAABQ)();

354

355 while (bMoreInstructions)

356 {

357 sim.[OutputCurrentInstructionCycle](#AAAAAAAABR)([tcout](#AAAAAAAABP));

358

359 bMoreInstructions = sim.[ProcessNextCycle](#AAAAAAAABQ)();

360 }

361

362 [tcout](#AAAAAAAABP) << \_T ( "------------------------------------------------------------------")

363 << std::endl;

364 [tcout](#AAAAAAAABP) << \_T ( "Total time for pipelined (overlapped) execution: " )

365 << [CalculatePartialOverlappedExecutionCycles](#AAAAAAAAAY) ( dag ) << \_T ( " cycles" ) << std::endl;

366

367 return bReturn;

Here is the call graph for this function:

#### size\_t LoadData (const TCHAR \* *szFileName*, [CDependencyGraph](#AAAAAAAAAL) & *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().

229 {

230 size\_t nReturn = 0;

231

232 [tifstream](#AAAAAAAAAU) infile;

233

234 #if defined(UNICODE) || defined(\_UNICODE)

235 std::locale utf8\_locale ( std::locale ( infile.getloc ( ) ), new std::codecvt\_utf8\_utf16<wchar\_t> );

236 infile.imbue ( utf8\_locale );

237 #endif

238

239 infile.open ( szFileName );

240

241 if ( infile.bad ( ) )

242 {

243 [tcout](#AAAAAAAABP) << \_T ( "Error opening data file:" ) << szFileName << std::endl;

244 }

245 else

246 {

247 TCHAR szLineBuffer[128] = { 0 };

248

249 // read in the 1st line of input, this will contain the list of instructions

250 infile.getline ( szLineBuffer, \_countof ( szLineBuffer ) - 1 );

251

252 // parse the instruction list, removing trailing punctuation.

253 [tstringstream](#AAAAAAAAAV) strStream;

254

255 strStream << szLineBuffer;

256

257 while ( strStream.getline ( szLineBuffer, 5, \_T ( ' ' ) ) && ( nReturn < [MAX\_INSTRUCTIONS](#AAAAAAAABE) ) )

258 {

259 dag.[AddNode](#AAAAAAAABS) ( szLineBuffer[0] );

260

261 nReturn++;

262 }

263

264 // now parse the instruction dependencies

265 // this will be in the format of:

266 // B<space>A<NL>

267 // where "B A" means that B depends on the result of A

268

269 TCHAR idSrcNode = 0;

270 TCHAR idDestNode = 0;

271

272 while ( infile >> idSrcNode >> idDestNode )

273 {

274 // in estimating an edge weight, lets use the time delta or "dependency

275 // distance" between when the 2 instructions are scheduled to begin execution.

276

277 int iWeight = idSrcNode - idDestNode;

278

279 dag.[AddEdge](#AAAAAAAABT) ( idSrcNode, idDestNode, iWeight );

280 }

281

282 infile.close ( );

283 }

284

285 return nReturn;

286 }

Here is the call graph for this function:

### 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](#AAAAAAAAAL) g\_DAG([MAX\_INSTRUCTIONS](#AAAAAAAABE))

Global directed acyclic graph object.

Referenced by \_tmain().

#### [CPipelineSim](#AAAAAAAABB) 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](#AAAAAAAABB) class implementation.

#include "stdafx.h"

#include "PipelineSim.h"

Include dependency graph for PipelineSim.cpp:

### Typedefs

* typedef std::list< [CInstructionData](#AAAAAAAABO) >::iterator [LstIterator](#AAAAAAAABV)
* typedef std::list< [CInstructionData](#AAAAAAAABO) >::reverse\_iterator [rLstIterator](#AAAAAAAABW)

### Variables

* const int [CONCURRENT\_INSTRUCTION\_LIMIT](#AAAAAAAABX) = 4

### Detailed Description

[CPipelineSim](#AAAAAAAABB) class implementation.

##### Author:

Mark L. Short

##### Date:

November 23, 2014

### Typedef Documentation

#### typedef std::list<[CInstructionData](#AAAAAAAABO)>::iterator [LstIterator](#AAAAAAAABV)

#### typedef std::list<[CInstructionData](#AAAAAAAABO)>::reverse\_iterator [rLstIterator](#AAAAAAAABW)

### 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](#AAAAAAAABB) 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:

### Classes

* class [CInstructionData](#AAAAAAAABO)

*Instruction data and state.*

* class [CNoopInstruction](#AAAAAAAABZ)
* class [CPipelineSim](#AAAAAAAABB)

### *A 4-staged pipeline simulation class.*

### Typedefs

* typedef enum [PS\_PIPELINE\_STATE](#AAAAAAAACC) [PS\_PIPELINE\_STATE\_T](#AAAAAAAACD)

*Pipeline Instruction State.*

* typedef TCHAR [INSTRUCTION\_T](#AAAAAAAACE)

### Enumerations

* enum [PS\_PIPELINE\_STATE](#AAAAAAAACC) { [PS\_INVALID](#AAAAAAAACF), [PS\_IF](#AAAAAAAACG), [PS\_ID](#AAAAAAAACH), [PS\_EX](#AAAAAAAACI), [PS\_WB](#AAAAAAAACJ), [PS\_COMPLETED](#AAAAAAAACK) }

### *Pipeline Instruction State.* Variables

* const [INSTRUCTION\_T](#AAAAAAAACE) [INVALID\_INSTRUCTION](#AAAAAAAACL) = 0

*Used to denote an uninitialized instruction*

* const [INSTRUCTION\_T](#AAAAAAAACE) [NOOP\_INSTRUCTION](#AAAAAAAACM) = '-'

### Detailed Description

[CPipelineSim](#AAAAAAAABB) class interface.

##### Author:

Mark L. Short

##### Date:

November 24, 2014

### Typedef Documentation

#### typedef TCHAR [INSTRUCTION\_T](#AAAAAAAACE)

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](#AAAAAAAACC) [PS\_PIPELINE\_STATE\_T](#AAAAAAAACD)

Pipeline Instruction State.

### Enumeration Type Documentation

#### enum [PS\_PIPELINE\_STATE](#AAAAAAAACC)

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

45 {

46 [PS\_INVALID](#AAAAAAAACF),

47 [PS\_IF](#AAAAAAAACG),

48 [PS\_ID](#AAAAAAAACH),

49 [PS\_EX](#AAAAAAAACI),

50 [PS\_WB](#AAAAAAAACJ),

51 [PS\_COMPLETED](#AAAAAAAACK)

52 } [PS\_PIPELINE\_STATE\_T](#AAAAAAAACD);

### Variable Documentation

#### const [INSTRUCTION\_T](#AAAAAAAACE) INVALID\_INSTRUCTION = 0

Used to denote an uninitialized instruction

#### const [INSTRUCTION\_T](#AAAAAAAACE) 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:

### 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

#### 

## 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:

### Detailed Description

Windows OS platform header file.

##### Author:

Mark L. Short

##### Date:

November 25, 2014

# Index

\_COMMON\_DEF\_H\_\_

CommonDef.h, 34

\_CRT\_SECURE\_NO\_WARNINGS

stdafx.h, 59

\_Pairib

CGraphNode, 19

\_PIPELINE\_SIM\_H\_\_

PipelineSim.h, 55

\_tmain

PipelineProject.cpp, 44

~CDependencyGraph

CDependencyGraph, 8

~CGraphNode

CGraphNode, 19

~CInstructionData

CInstructionData, 25

~CPipelineSim

CPipelineSim, 29

AddEdge

CDependencyGraph, 8

CGraphNode, 20

AddNode

CDependencyGraph, 9

BASE\_CYCLES\_PER\_INSTUCTION

PipelineProject.cpp, 50

begin

CDependencyGraph, 10

beginEdge

CGraphNode, 21

BYTE

CommonDef.h, 35

CalculateCompleteOverlappedExecutionCycles

PipelineProject.cpp, 45

CalculateNumberOfStallsRequired

PipelineProject.cpp, 45

CalculatePartialOverlappedExecutionCycles

PipelineProject.cpp, 46

CalculateSequentialExecutionCycles

PipelineProject.cpp, 47

CDependencyGraph, 6

~CDependencyGraph, 8

AddEdge, 8

AddNode, 9

begin, 10

CDependencyGraph, 8

const\_iterator, 7

end, 10

GetNodeIndex, 10

GetNumEdges, 11

GetNumNodes, 11

HasNode, 11

IsValidNodeID, 12

IsValidNodeIndex, 12

m\_nMaxNodes, 13

m\_nNumNodes, 13

m\_vNodes, 13

operator=, 13

CDirectedEdgeData, 14

CDirectedEdgeData, 14, 15

GetDestNodeID, 15

GetWeight, 15

m\_idDestNode, 16

m\_iWeight, 16

operator<, 15

operator==, 15

operator>, 16

SetNodeID, 16

SetWeight, 16

CGraphNode, 18

\_Pairib, 19

~CGraphNode, 19

AddEdge, 20

beginEdge, 21

CGraphNode, 19

const\_iterator, 19

EDGE\_SET\_T, 19

endEdge, 21

GetNodeID, 21

GetNumEdges, 21

HasEdge, 21

IsValid, 22

m\_ID, 22

m\_setEdges, 22

SetNodeID, 22

CInstructionData, 24

~CInstructionData, 25

CInstructionData, 25

GetInstruction, 25

GetState, 25

IsDataDependent, 26

IsNOOP, 26

m\_bDataDependent, 26

m\_Instruction, 26

m\_psState, 26

SetDataDependent, 26

SetState, 26

CNoopInstruction, 27

CNoopInstruction, 27

CommonDef.h

\_COMMON\_DEF\_H\_\_, 34

BYTE, 35

DWORD, 35

CONCURRENT\_INSTRUCTION\_LIMIT

PipelineSim.cpp, 53

const\_iterator

CDependencyGraph, 7

CGraphNode, 19

CPipelineSim, 28

~CPipelineSim, 29

CPipelineSim, 29

GetCompletionCount, 29

GetCycle, 30

GetStallCount, 30

InsertInstruction, 30

m\_dwCompletedCtr, 33

m\_dwCycle, 33

m\_dwMaxPipelineDepth, 33

m\_dwStallCtr, 33

m\_lstInstructionPipeline, 33

m\_queInstructions, 33

OutputCurrentInstructionCycle, 30

ProcessNextCycle, 31

DebugTrace

DebugUtility.cpp, 36

DebugUtility.h, 38

DebugUtility.cpp

DebugTrace, 36

GetModulePath, 37

DebugUtility.h

DebugTrace, 38

GetModulePath, 39

DEFAULT\_MAX\_NODES

DependencyGraph.cpp, 40

DependencyGraph.cpp

DEFAULT\_MAX\_NODES, 40

DependencyGraph.h

INVALID\_NODE\_ID, 42

INVALID\_NODE\_INDEX, 42

NODE\_ID\_T, 42

DWORD

CommonDef.h, 35

EDGE\_SET\_T

CGraphNode, 19

end

CDependencyGraph, 10

endEdge

CGraphNode, 21

ExecutePipelineSimulation

PipelineProject.cpp, 48

g\_DAG

PipelineProject.cpp, 50

g\_PipelineSim

PipelineProject.cpp, 51

g\_szFileName

PipelineProject.cpp, 51

GetCompletionCount

CPipelineSim, 29

GetCycle

CPipelineSim, 30

GetDestNodeID

CDirectedEdgeData, 15

GetInstruction

CInstructionData, 25

GetModulePath

DebugUtility.cpp, 37

DebugUtility.h, 39

GetNodeID

CGraphNode, 21

GetNodeIndex

CDependencyGraph, 10

GetNumEdges

CDependencyGraph, 11

CGraphNode, 21

GetNumNodes

CDependencyGraph, 11

GetStallCount

CPipelineSim, 30

GetState

CInstructionData, 25

GetWeight

CDirectedEdgeData, 15

HasEdge

CGraphNode, 21

HasNode

CDependencyGraph, 11

InsertInstruction

CPipelineSim, 30

INSTRUCTION\_T

PipelineSim.h, 55

INVALID\_INSTRUCTION

PipelineSim.h, 56

INVALID\_NODE\_ID

DependencyGraph.h, 42

INVALID\_NODE\_INDEX

DependencyGraph.h, 42

IsDataDependent

CInstructionData, 26

IsNOOP

CInstructionData, 26

IsValid

CGraphNode, 22

IsValidNodeID

CDependencyGraph, 12

IsValidNodeIndex

CDependencyGraph, 12

LoadData

PipelineProject.cpp, 49

LstIterator

PipelineSim.cpp, 52

m\_bDataDependent

CInstructionData, 26

m\_dwCompletedCtr

CPipelineSim, 33

m\_dwCycle

CPipelineSim, 33

m\_dwMaxPipelineDepth

CPipelineSim, 33

m\_dwStallCtr

CPipelineSim, 33

m\_ID

CGraphNode, 22

m\_idDestNode

CDirectedEdgeData, 16

m\_Instruction

CInstructionData, 26

m\_iWeight

CDirectedEdgeData, 16

m\_lstInstructionPipeline

CPipelineSim, 33

m\_nMaxNodes

CDependencyGraph, 13

m\_nNumNodes

CDependencyGraph, 13

m\_psState

CInstructionData, 26

m\_queInstructions

CPipelineSim, 33

m\_setEdges

CGraphNode, 22

m\_vNodes

CDependencyGraph, 13

MAX\_INSTRUCTIONS

PipelineProject.cpp, 51

NODE\_ID\_T

DependencyGraph.h, 42

NOOP\_INSTRUCTION

PipelineSim.h, 56

operator<

CDirectedEdgeData, 15

operator=

CDependencyGraph, 13

operator==

CDirectedEdgeData, 15

operator>

CDirectedEdgeData, 16

OutputCurrentInstructionCycle

CPipelineSim, 30

PipelineProject.cpp

\_tmain, 44

BASE\_CYCLES\_PER\_INSTUCTION, 50

CalculateCompleteOverlappedExecutionCycles, 45

CalculateNumberOfStallsRequired, 45

CalculatePartialOverlappedExecutionCycles, 46

CalculateSequentialExecutionCycles, 47

ExecutePipelineSimulation, 48

g\_DAG, 50

g\_PipelineSim, 51

g\_szFileName, 51

LoadData, 49

MAX\_INSTRUCTIONS, 51

tifstream, 44

tstringstream, 44

PipelineProject/CommonDef.h, 34

PipelineProject/DebugUtility.cpp, 36

PipelineProject/DebugUtility.h, 38

PipelineProject/DependencyGraph.cpp, 40

PipelineProject/DependencyGraph.h, 41

PipelineProject/PipelineProject.cpp, 43

PipelineProject/PipelineSim.cpp, 52

PipelineProject/PipelineSim.h, 54

PipelineProject/stdafx.cpp, 57

PipelineProject/stdafx.h, 58

PipelineProject/targetver.h, 60

PipelineSim.cpp

CONCURRENT\_INSTRUCTION\_LIMIT, 53

LstIterator, 52

rLstIterator, 52

PipelineSim.h

\_PIPELINE\_SIM\_H\_\_, 55

INSTRUCTION\_T, 55

INVALID\_INSTRUCTION, 56

NOOP\_INSTRUCTION, 56

PS\_COMPLETED, 55

PS\_EX, 55

PS\_ID, 55

PS\_IF, 55

PS\_INVALID, 55

PS\_PIPELINE\_STATE, 55

PS\_PIPELINE\_STATE\_T, 55

PS\_WB, 55

tostream, 55

ProcessNextCycle

CPipelineSim, 31

PS\_COMPLETED

PipelineSim.h, 55

PS\_EX

PipelineSim.h, 55

PS\_ID

PipelineSim.h, 55

PS\_IF

PipelineSim.h, 55

PS\_INVALID

PipelineSim.h, 55

PS\_PIPELINE\_STATE

PipelineSim.h, 55

PS\_PIPELINE\_STATE\_T

PipelineSim.h, 55

PS\_WB

PipelineSim.h, 55

rLstIterator

PipelineSim.cpp, 52

SetDataDependent

CInstructionData, 26

SetNodeID

CDirectedEdgeData, 16

CGraphNode, 22

SetState

CInstructionData, 26

SetWeight

CDirectedEdgeData, 16

stdafx.h

\_CRT\_SECURE\_NO\_WARNINGS, 59

tcout, 59

tstring, 59

tcout

stdafx.h, 59

tifstream

PipelineProject.cpp, 44

tostream

PipelineSim.h, 55

tstring

stdafx.h, 59

tstringstream

PipelineProject.cpp, 44