The Register Assignment Problem

Optimizing register placement in HLS designs.

Developed by the FPGA Compiler Team

**Contents**

[1 Overview 2](#_Toc459386105)

[2 The Register Assignment Problem: 3](#_Toc459386106)

[2.1 Overview 3](#_Toc459386107)

[2.1.1 Component Selection: 3](#_Toc459386108)

[2.1.2 Register Assignment: 3](#_Toc459386109)

[2.1.3 Extract Scheduling Constraints: 3](#_Toc459386110)

[2.2 Delay Graph 4](#_Toc459386111)

[2.3 CDFG to Delay Graph Translation 5](#_Toc459386112)

[2.4 Cycles in Delay Graph: 6](#_Toc459386113)

[2.4.1 Loop Carried Dependencies 7](#_Toc459386114)

[2.4.2 Algorithmic Feedback 7](#_Toc459386115)

[2.4.3 Memory constraints across Loop Iterations: 8](#_Toc459386116)

[3 Constraints 9](#_Toc459386117)

[3.1 Inherent Vertex Constraints: 9](#_Toc459386118)

[3.2 Correlated Inputs and Outputs Terminals: 9](#_Toc459386119)

[3.3 No Combinational Cycle: 10](#_Toc459386120)

[3.4 No Timing Path Violation: 10](#_Toc459386121)

[4 Optimization Objectives 10](#_Toc459386122)

[4.1 Throughput Cost 10](#_Toc459386123)

[4.2 Latency Cost 10](#_Toc459386124)

[4.3 Register Cost 11](#_Toc459386125)

[5 Testbench and Test Cases 11](#_Toc459386126)

[6 Register Assignment Algorithms: 12](#_Toc459386127)

[6.1 ASAP Algorithm 12](#_Toc459386128)

[6.2 Greedy Algorithm 12](#_Toc459386129)

[7 Lesson Plans: 13](#_Toc459386130)

[7.1 Topological Sort for DAG 13](#_Toc459386131)

[7.1.1 Recursive implementation 13](#_Toc459386132)

[7.1.2 Iterative implementation 13](#_Toc459386133)

[7.2 Cycle Detection using Tarjan’s Algorithm 13](#_Toc459386134)

[7.2.1 Recursive implementation 14](#_Toc459386135)

[7.2.2 Iterative implementation 14](#_Toc459386136)

[7.3 Find Longest Paths 14](#_Toc459386137)

[7.3.1 Topological Sort based implementation 14](#_Toc459386138)

[7.3.2 WavefrontDictionary 15](#_Toc459386139)

[7.4 Register Assignment 15](#_Toc459386140)

# Overview

This Register Assignment Problem Package contains following components:

1. Formulation of the Register Assignment Problem
2. Specification of the Delay Graph data structures
3. 2300+ test cases extracted from realistic applications
4. Testbench that runs multiple implementations and compares results
5. Lesson plans for programming exercises

The motivations for creating and distributing this package are:

1. Advance the state of the art in High Level Synthesis
2. Provide a rich and realistic set of example graphs and testbench
3. Educate theory and implementation of advanced graph algorithms

What is new and novel:

1. The register assignment problem formation is novel, lending itself to new and interesting solutions
2. The delay graph is cyclic, encouraging learning and development of cyclic graph algorithms
3. The delay graph can also be treated as directed acyclic graph, enabling acyclic graph algorithms
4. Large number of huge example graphs, and a testbench that runs different algorithms over this data set, allows easy compare of user algorithms
5. Lesson plans to introduce programming graph algorithms for large graphs.

Material:

1. C# Source code of Delay Graph, two Register Assignment algorithms, and Testbench that runs these algorithms over test set
2. Test set of 2300+ Delay Graph and Clock Period Constraints (both in GraphML xml format).
3. Documentation of the Delay Graph Data Structures, the Register Assignment Problem, and Suggested lesson plans

# ****The Register Assignment Problem****:

## Overview

In High Level Synthesis (HLS), a data and control flow graph (CDFG) representing an algorithm is compiled into a structural HDL netlist that implements the algorithm in hardware with a particular target clock rate. HLS compilation flow typically abstracts the picosecond timing among HDL netlist components away from clock cycle level scheduling problem.

One way to achieve this abstraction may consist of following steps:

### Component Selection:

For each node in the CDFG, pick a HDL component from a component database that will run at the target clock rate. This component will have picosecond timing characteristics which can be back annotated onto the inputs and output terminals of the original node in the CDFG. For example, a multiply operation may be implemented by a specific multiply component that has specific pipelining and timing characteristics depending on target clock rate. The Latency of the HDL component is the number of pipeline register stages inside, which corresponds to the number of clock cycles from inputs to outputs.

### Register Assignment:

Assign additional registers in between CDFG nodes to ensure that all combinational timing path will have delays that are within the target clock period.  A convenient simplification is to assign registers at all inputs and/or all outputs of individual CDFG nodes. If we decide to assign registers at all inputs of a node, we set the node's PreLatency to 1; if we decide to assign registers at all outputs of a node, we set the node's PostLatency to 1. Otherwise each node's Pre and Post Latencies default to 0.

### Extract Scheduling Constraints:

After Component Selection and Register Assignment, every node in the CDFG has PreLatency, PostLatency and Latency integer numbers which ensures that, if we map every CDFG node to its selected HDL component and connect up their inputs and outputs according to their Pre and Post Latency numbers, then this instantiated HDL netlist would meet target clock rate. This abstracts away picosecond timing into cycle level constraints. The scheduling problem is to further optimize beyond this one-to-one mapping, for example by assigning nodes to scheduling cycles to allow time multiplexed sharing of HDL components. The scheduling constraints can be extracted from the resulting CDFG by considering each node to produce its outputs cycles after the cycle in which its latest inputs arrived, where . That is, if node A's output drives B's input, then:

In this flow, the solution to the register assignment problem directly determines an upper bound on the HLS quality of result. For example, if the register assignment solution inserted 2 registers in an algorithmic cycle, then the initiation interval (II) for the final solution is at least 2 cycles. If the register assignment solution inserted 3 registers in the path from input to output, then the algorithmic latency for the final solution is at least 3 cycles.

For the register assignment problem, we extract a Delay Graph from the CDFG and the results of component selection. The edges in this Delay Graph are timing edges, the vertices in this Delay Graph are points in the CDFG where we might insert registers. This delay graph captures picosecond timing edges in and between selected components in the CDFG, as well as throughput and latency cost functions for each candidate register assignments. The register assign problem is then to assign registers in this Delay Graph in order to minimize costs (namely, initiation interval, latency, and registers), subject to timing constraints (namely, no combinational cycles, and every combinational timing path is shorter than the specified clock period).

## ****Delay Graph****

The DelayGraph is a directed cyclic graph data structure consisting of vertices and edges, namely, DelayGraphVertex and DelayGraphEdge objects. Every vertex can have 0 or more incoming edges and 0 or more outgoing edges. Every edge has one and only one source vertex and one and only one target vertex. The direction of an edge is from Source vertex to Target vertex. By construction (when extracting from CDFG), we tag certain edges with property **IsFeedback** as true, such that, if we ignore all edges with **IsFeedback**=true (aka, feedback edges), we will end up with a Directed Acyclic Graph.

The DelayGraphSolution contains a DelayGraph and a HashSet of DelayGraphVertex objects which are to be registered. This implements cost functions that evaluate throughput, latency and register costs for a register assignment solution, as well as dot file print functions that allow visualization of the solution.

The package also includes a DelayGraphGraphMLSerializer class which implements a serialization of DelayGraph to GraphML (xml format) file, so that it can be stored on disk as a  **.**graphml (xml) file. There is also a corresponding deserialization function that converts from **.**graphml to DelayGraph in the same DelayGraphGraphMLSerializer class. Properties that need to be serialized/deserialized are marked with the tag [GraphmlAttribute()].

**DelayGraphVertex**:

A DelayGraphVertex object has following properties tagged with [GraphmlAttribute()]

* NodeUniqueId: A Dataflow node id, to correlate Delay Graph Vertices that originate from the same Dataflow node.
* ThroughputCostIfRegistered: An integer that represents the throughput cost of this vertex when it is registered.
* LatencyCostIfRegistered: An integer that represents the latency cost of this vertex when it is registered.
* RegisterCostIfRegistered: An integer that represents the register cost of this vertex when it is registered.
* IsRegistered: A boolean which indicates that this vertex is already registered.
* IsInputTerminal: A boolean which indicates that this vertex represents an input terminal.
* IsOutputTerminal: A boolean which indicates that this vertex represents an output terminal.
* DisallowRegister: A boolean which indicates that this terminal is not allowed to be registered

**DelayGraphEdge**:

A DelayGraphEdge object has following Properties tagged with [GraphmlAttribute()]:

* Source: A DelayGraphVertex which represents the source of the delay edge.
* Target: A DelayGraphVertex which represents the destination of the delay edge.
* Delay: An integer which represents the pico second timing from Source to Target.
* IsFeedback: A bool which, if true, means the edge represents a feedback edge in original Data Flow graph.

**DelayGraph Solution:**

A DelayGraph Solution consists of a DelayGraph object and a HashSet containing DelayGraph Vertices that are to be registered.

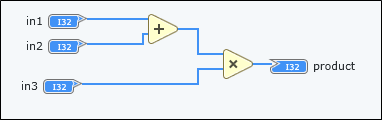
* Graph: A DelayGraph.
* RegisteredTerminals: A HashSet of DelayGraphVertex objects, which indicate which Delay Graph vertices are to be registered.

## CDFG to Delay Graph Translation

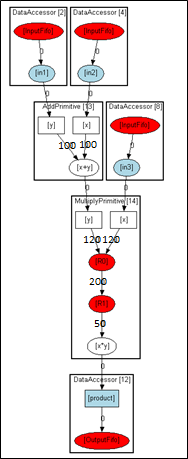
Given the pseudo code:

product := (in1 + in2) \* in3;

The CDFG may look like this:



And the DelayGraph may look like this:



The Add operation has been converted to the subgraph (in the box denoted as AddPrimitive[13]) containing two input vertices (drawn as boxes) and one output vertex (the oval) with the delay edges of 100 from each input to the output vertex. And the Multiply operation has been converted to a subgraph (in the box denoted as MultiplyPrimitive[14]) containing two input vertices, one output vertex, and two internal register vertices (red ovals). The timing edges from inputs to the first internal register has delays of 120, the timing edge in between the two internal registers has a delay of 200, and the delay from second register to output of the multiply is 50. The blue colored vertices denote the vertices with DisallowRegister property being true. In this example, the primary input and outputs of the CDFG are from and to FIFO registers, which are displayed as DataAccessor boxes. Note that the boxes, colors and shapes, are drawn by PrintDotString() routines which generates dot files from DelayGraphSolution, the vertices are grouped into boxes by their matching NodeUniqueId values.

## Cycles in Delay Graph:

Cycles in Delay Graph comes from following sources:

### Loop Carried Dependencies

A variable that is assigned in one loop iteration and then used in later loop iteration, creates a loop carried dependency that shows up as a cycle in the Delay Graph.

For example:

int x = 0;

for (int idx = 0; idx < 8; idx++)

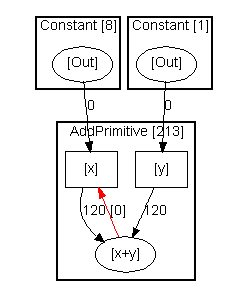
{

x = x + 1;

}

This has a loop carried dependency in x. The output of the addition operation for “x + 1” feeds into the input for the same addition operation in the next loop iteration.

This is the corresponding DelayGraph:



Where the red edge denotes the feedback edge from output of x+1 to input x of x + 1.

### Algorithmic Feedback

Even without Loops, there may be algorithmic feedback that prevents consecutive calls to a function until the feedback has been computed. A simple example is with static variable that is updated in one call of a function and used by the next call of the same function.

For example:

function int macc (int x, int y)

{

static int sum = 0;

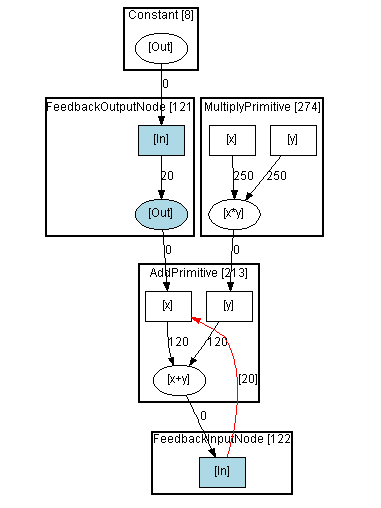
sum += x \* y;

return sum;

}

The feedback variable, “sum”, creates a cycle in the Delay Graph to capture the feedback timing constraint for the throughput of this function.

The DelayGraph might look like this:



Where the blue colors mark vertices with “DisallowRegister=true” properties. In this case, there is a mux delay (shown arbitrarily as 20 in the diagram) to select between the initial value of “sum” and subsequent feedback value, so there is a feedback edge of delay=20 (shown as red edge with label “[20]” in the dotty diagram).

### Memory constraints across Loop Iterations:

Array read and write operations may be mapped to memory read and write operations that share the same memory port. These operations impose scheduling constraints which may generate feedback edges in the Delay Graph.

For example:

int array[1024];

for (int idx=0; idx<8; idx++)

{

a = array[x];

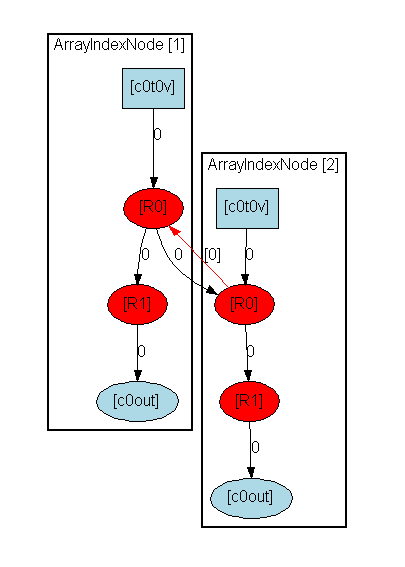
b = array[x+1];

…

}

If “array” is mapped to a memory, and if this memory only has a single read port, then these two array read operations inside the loop has to be serialized, not just within the same loop iteration, but across loop iterations as well.

The Delay Graph may look like this:



The array read operations are mapped to memory read components with pipelined, 2 cycle latency. The red vertices denote vertices with property IsRegistered=true and the blue vertices denote vertices with DisallowRegister=true. Each memory read component contains a blue box, representing the address input terminal, two red ovals representing the internal pipeline registers, and a blue oval representing the data output terminal. There are two edges between the first registered vertices of the two read components, one forward edge and one feedback edge. The forward edge captures the scheduling constraint from the first read to the second read in same loop iteration. The red feedback edge captures the scheduling constraint from the second read in one loop iteration to the first read in the next loop iteration. In this case, there are two registers in the cycle formed by the forward and feedback edges, therefore the minimum initiation interval for this for loop is 2.

# Constraints

The following constraints apply for a Register Assignment to be a valid solution:

## Inherent Vertex Constraints:

* A vertex with property IsRegistered=true represents internal pipeline register inside a HDL component, and cannot be de-registered.
* A vertex with property DisallowRegister=true represents internal signals inside a HDL component, and cannot be registered.

## Correlated Inputs and Outputs Terminals:

* All input terminal vertices (i.e., with property IsInputTerminal=true) with the same NodeUniqueId must be either all registered, or all un-registered.
* This allows back annotating the register assignment to CDFG nodes as PreLatency of 0 or 1.
* All output terminal vertices (i.e., with property IsOutputTerminal=true) with the same NodeUniqueId must be either all registered, or all un-registered. This allows back annotating the register assignment to CDFG nodes as PostLatency of 0 or 1.

## No Combinational Cycle:

A combinational cycle happens when there is a cycle in the delay graph in which none of the vertices are registered. This is not allowed in a solution because it would lead to a combinational cycle in the HDL Components. A valid solution must break all cycles by inserting at least one register in every cycle path.

## No Timing Path Violation:

Given a target clock period, a valid register assignment solution must ensure that every combinational path (i.e., a path consisting of unregistered vertices) must have cumulative delays that are less than or equal to the target clock period.

# ****Optimization Objectives****

There are three optimization objectives in the Register Assignment Problem, namely, to minimize (1) throughput cost, (2) latency cost, and (3) registers cost. The solution comparison function in the DelayGraphSolution implements a priority comparison function, giving highest priority to initiation interval and lowest priority to registers.

## ****Throughput Cost****

Each vertex inside a FOR loop with loop iteration counts of N is assigned a ThroughputCostIfRegistered of N cycles. If this vertex is inside a nest FOR loop with loop iteration counts of N1 and N2, then it is assigned a ThroughputCostIfRegistered of N1xN2 cycles. Given a DelayGraphSolution, we compute the overall throughput cost of the solution as the maximum throughput cost of all cycle in the DelayGraph, where the throughput cost of a cycle is calculated by finding all registered vertices in the cycle, and summing the ThroughputCostIfRegistered for these registered vertices.

This calculates the best case throughput for the CDFG given the register assignment specified by the solution.

## ****Latency Cost****

Each vertex inside a FOR loop with loop iteration counts of N is also assigned a LatencyCostIfRegistered of N cycles. If this is inside a nest FOR loop with loop iteration counts of N1 and N2, then it is assigned a LatencyCostIfRegistered of N1xN2 cycles. Given a DelayGraphSolution, we compute the overall latency cost of the solution as the maximum latency cost of all paths from any input to any output, where the latency cost of a path is calculated by finding all registered vertices in the path, and summing the LatencyCostIfRegistered for these registered vertices. This calculates the best case Latency for the CDFG given the register assignment specified by the solution. For this purpose, the inputs of the DelayGraph are vertices which have no incoming edges, and outputs are vertices which have no outgoing edges.

## ****Register Cost****

Each vertex in the DelayGraph is assigned a RegisterCostIfRegistered which is basically the number of register bits required to insert a register at the corresponding point in the CDFG. Given a DelayGraphSolution, we compute the overall register cost by finding all registered vertices, and summing their RegisterCostIfRegistered costs.

# ****Testbench and Test Cases****

The DataSets folder contains >2300 test cases numbered 0, 1, 2, etc. Each test case N consists of a DelayGraph\_<N>.graphml file which contains the serialized Delay Graph, and OriginalGoals\_<N>. xml file that contains the target clock period for this test. For example, for test case 0, DelayGraph\_0.graphml contains its delay graph, and OriginalGoals\_0.xml contains the target clock period in picosecond for this test case.

The Program.cs implements a testbench that takes twos arguments. First argument is the path to the DataSets folder, and the second argument is the path to Reports folder where that program’s output is written to.

The program finds all tests in the DataSets folder, and for each test case, it de-serializes the graphml for this case, and the xml target clock period, and runs the list of register assignment algorithms in the list “algorithms”:

var algorithms = new List<Tuple<string, ILatencyAssignment>>

{

new Tuple<string, ILatencyAssignment>("asap", new LatencyAssignmentAsap()),

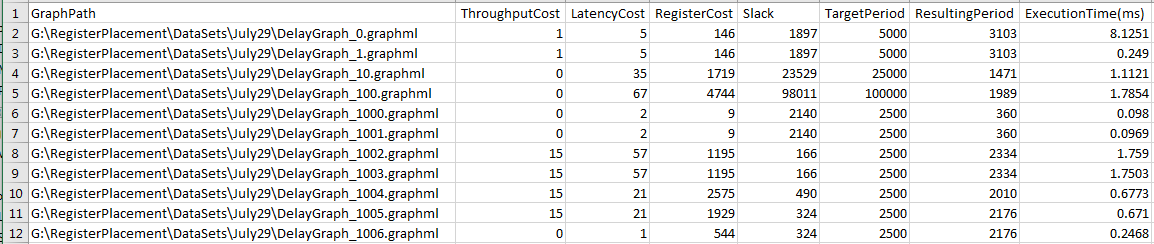
new Tuple<string, ILatencyAssignment>("greedy", new LatencyAssignmentGreedy()),

// add your own latency assigner here

};

**Additional algorithms may be added to this list and it will be run over all test cases and solutions compared with those generated by the other algorithms.**

**The program generates an excel (.csv) file for each algorithm, tabulating the costs and runtime for each test case. For example, the asap\_ScoreCard.csv is generated for the “asap” algorithm:**



**It also generates a summary report for all algorithms run over all test cases, in Summary.csv:**



**The DelayGraphSolution construction method checks for and repairs (1) correlated input and output terminals and (2) combinational cycles, before computing the costs for each solution.**

**Finally, it also generates a dot file for every solution produced by each algorithm, in a DotFiles folder next to where the executable is, using the test name and algorithm name. For example, DelayGraph\_0.asap.dot is the dot file for solution generated by “asap” algorithm, for test case DelayGraph\_0.**

# ****Register Assignment Algorithms:****

Two algorithms are included in this package. The “asap” and “greedy” algorithms.

## ASAP Algorithm

This algorithm does a simple traversal of all vertices in the delay graph, and for each vertex, it computes the maximum upstream delays, and maximum downstream delays, and see if the sum of these two is greater than the target clock period. If it is, then a register is assigned to this vertex. The upstream delays are cumulative, whereas the downstream delays are simply the delays on immediate outgoing edges. It also preemptively assigns register to a vertex if the downstream register costs are higher. This traversal is performed twice, to try to include more accurate upstream delays due to feedback edges.

Known issues with this algorithm:

* It does not respect correlation of input and output terminals, nor DisallowRegister property of vertices.
* It does not traverse in topological order, so cumulative upstream delay calculation is haphazard.
* It has a simple, localized heuristic to bias towards lowering the objective costs.

## Greedy Algorithm

This algorithm initializes the solution to have registers everywhere, and then takes a vertex group in decreasing cost, and see if this vertex group can be de-registered (i.e., changed to be not registered) without violating timing or combinational cycle constraints. A vertex group is correlated group of input or output terminal vertices. A map of upstream and downstream delays is maintained for each vertex. This is initialized to simply the incoming and outgoing edge delays. To check if a vertex can be de-registered, we just have to check that (1) its upstream vertices do not intersect with its downstream vertices, and (2) the maximum upstream delay plus the maximum downstream delay add up to less than the target clock period. A vertex group can be de-registered if and only if all vertices in the group can be de-registered. As a vertex is de-registered, all of its upstream vertices are added to the upstream vertices of all of its downstream vertices, and all of its downstream vertices are added to the downstream vertices of all of its upstream vertices.

Known issues with this algorithm:

* The vertex groups are processed in a static order, and the next vertex group to work on may be off the critical paths of the current solution and therefore not the most profitable vertex group to try next.
* Greedy nature of this algorithm means it can get trapped in local minima.

# ****Lesson Plans:****

These are suggested topics in a graph algorithms class:

## ****Topological Sort for DAG****

The delay graph can be seen as a Directed Acyclic Graph by ignoring the feedback edges. The following API’s enumerate only the forward outgoing and incoming edges for a vertex v:

internal IEnumerable<DelayGraphEdge> GetForwardOutEdges(DelayGraphVertex v);

internal IEnumerable<DelayGraphEdge> GetForwardInEdges(DelayGraphVertex v);

### ****Recursive implementation****

**The DelayGraphAlgorithmcs.cs contains the core recursive topological sort function:**

private static bool TopoRecursive(DelayGraphVertex v, LinkedList<DelayGraphVertex> sorted, HashSet<DelayGraphVertex> visited, DelayGraph graph);

The issue with recursive implementation is that it will cause stack overflow when run on some of the bigger delay graph test cases.

### ****Iterative implementation****

The DelayGraphAlgorithms,cs contains the iterative implementation for the topological sort function:

internal static IEnumerable<DelayGraphVertex> TopologicalSort(DelayGraph graph);

private static bool TopoIterative(DelayGraphVertex v, LinkedList<DelayGraphVertex> sorted, Dictionary<DelayGraphVertex, VisitState> visited, DelayGraph graph);

## ****Cycle Detection using Tarjan’s Algorithm****

When extracting a delay graph from CDFG, by default all vertices have ThroughputCostIfRegistered equal to its LatencyCostIfRegistered. However, if a vertex does not lie in a cycle in the delay graph, then its ThroughputCostIfRegistered is cleared to 0. To do this, we need an efficient way to determine whether a vertex is in any cycle, for all vertices in the graph.

Tarjan’s algorithm for finding strongly connected components in a graph may be used to find all vertices that are not in any loops.

This is implemented in function:

internal static List<List<DelayGraphVertex>> DetectCycles(DelayGraph graph);

### ****Recursive implementation****

The recursive implementation for the core Tarjan cycle detection is in function:

internal static List<List<T>> TarjanCycleDetectRecursive<T>(IEnumerable<T> nodes, Func<T, IEnumerable<T>> getFanouts) ;

Again, this recursive function will cause stack overflow when run on some of the larger delay graphs.

### ****Iterative implementation****

The iterative implementation for the Tarjan cycle detection is in function:

private static void StronglyConnectIterative<T>(ref int index, T node, Dictionary<T, SccInfo> infoMap, Stack<T> stack, List<List<T>> sccs, Func<T, IEnumerable<T>> getFanouts)

The iterative implementation basically mimics what the recursion stack would have done in the recursion implementation, so that the vertices and their fanout vertices are processed in identical order as that by the recursion implementation.

## ****Find Longest Paths****

In order to calculate the Latency cost of a register assignment solution, we need to calculate the maximum path length in the solution. Here, the length of a path is defined as the sum of vertex latency costs for all vertices along a path. The latency cost of a vertex is 0 if it is not registered, and is its LatencyCostIfRegistered value if the vertex is registered. The maximum path length in the solution is the maximum path length over all paths in the delay graph.

Similarly, in order to calculate the Throughput cost of a register assignment solution, we need to calculate the maximum cycle length in the solution. Here, the length of a cycle is defined as the sum of vertex Throughput costs for all vertices along a cycle. The throughput cost of a vertex is 0 if it is not registered, and is its ThroughputCostIfRegistered if the vertex is registered. The maximum cycle length in the solution is the maximum cycle length over all cycles in the delay graph.

If we only consider the forward edges in the graph and calculate path lengths for all paths, then Latency cost is computed for paths starting from vertices without input edges and ending at vertices without output edges. Similarly, Throughput cost is computed for paths starting from vertices with feedback input edges and ending at vertices with the matching feedback output edges.

### ****Topological Sort based implementation****

The following function in DelayGraphAlgorithms.cs implements a topological sort based all paths longest paths algorithm to calculate maximum latency cost for a solution:

internal static long FindMaxLatencyCore(IEnumerable<DelayGraphVertex> sort, DelayGraph graph, HashSet<DelayGraphVertex> registered)

### ****WavefrontDictionary****

The latency and throughput calculations make use of the WaveFrontDictionary class data structure to manage data indexed by DelayGraphVertex. When a data is entered into the dictionary for a vertex, a reference count is specified. Each time the data is looked up from the dictionary, its reference count is decremented, and when that is decremented to 0, the data item is removed from the dictionary. This is useful for keeping track of, for example, all paths lengths coming into a vertex, for use by its downstream vertices, and freeing this data when all downstream vertices have accessed this data.

private class WaveFrontDictionary<T> : Dictionary<DelayGraphVertex, Packet<T>>

## ****Register Assignment****

The register assignment problem is an open problem. The included algorithms are heuristic and greedy, and can be stuck in local minima. It would be interesting to see what optimization algorithms might be applied to this problem. Some interesting approaches are:

* Better greedy heuristics
* Genetic algorithms,
* Simulated annealing,
* Simulated evolution,
* Integer linear programming