ECE 5997
Hardware Accelerator Design & Automation
Fall 2021

Resource Sharing
Pipelining

Cornell University

CSL

Announcements

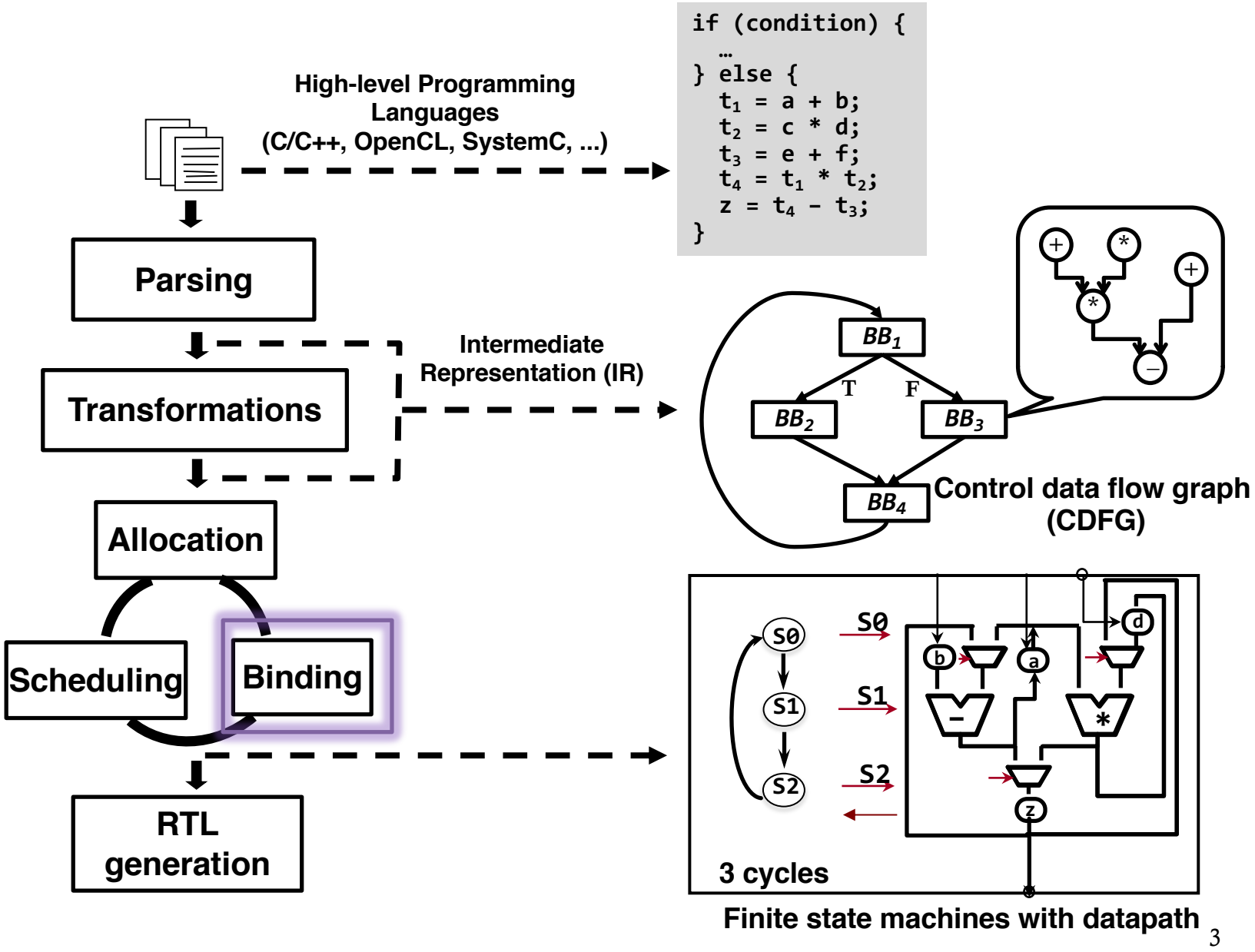
- Yichi Zhang (PhD TA) will give a tutorial on neural networks next Tuesday in Rhodes 310

- Lab 3 is released (due 12/1)
  - Go through the CORDIC tutorial asap

Agenda

- Resource sharing overview
  - Sub-problems: functional unit, register, and connectivity binding problems
  - Key concepts: compatibility and conflict graphs
- Introduction to pipelining
  - Common forms in hardware accelerators
  - Throughput restrictions
  - Dependence types

Recap: A Typical HLS Flow



Resource Sharing and Binding

- **Resource sharing** enables reuse of hardware resources to minimize cost, in resource usage/area/power
  - Typically carried out by binding in HLS
  - Other subtasks such allocation and scheduling greatly impact the resource sharing opportunities

- **Binding** maps operations, variables, and/or data transfers to the available resources
  - After scheduling: decide resource usage and detailed architecture **(focus of this lecture)**
  - Before scheduling: affect both area and delay
  - Simultaneous scheduling and binding: better result but more expensive

Binding Sub-problems

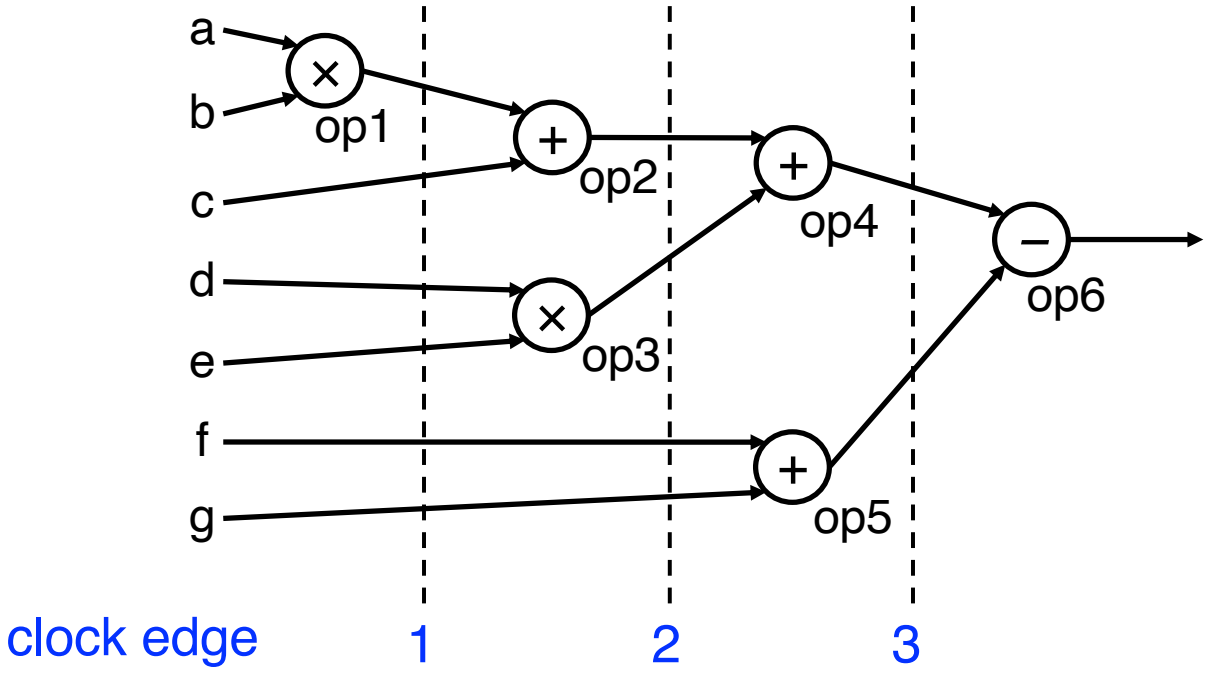
- Functional unit (FU) binding
  - Primary objective is to minimize the number of FUs
  - Considers connection cost
- Register binding
  - Primary objective is to minimize the number of registers
  - Considers connection cost
- Connectivity binding
  - Minimize connections by exploiting the commutative property of some operations / FUs
  - NP-hard

Sharing Conditions

- Functional units (registers) are shared by operations (variables) of same type whose *lifetimes* do not overlap

- **Lifetime:** [birth-time, death-time)
  - Operation: The whole execution time (if unpipelined)
  - Variable: From the time this variable is defined to the time it is last used

Operation Binding



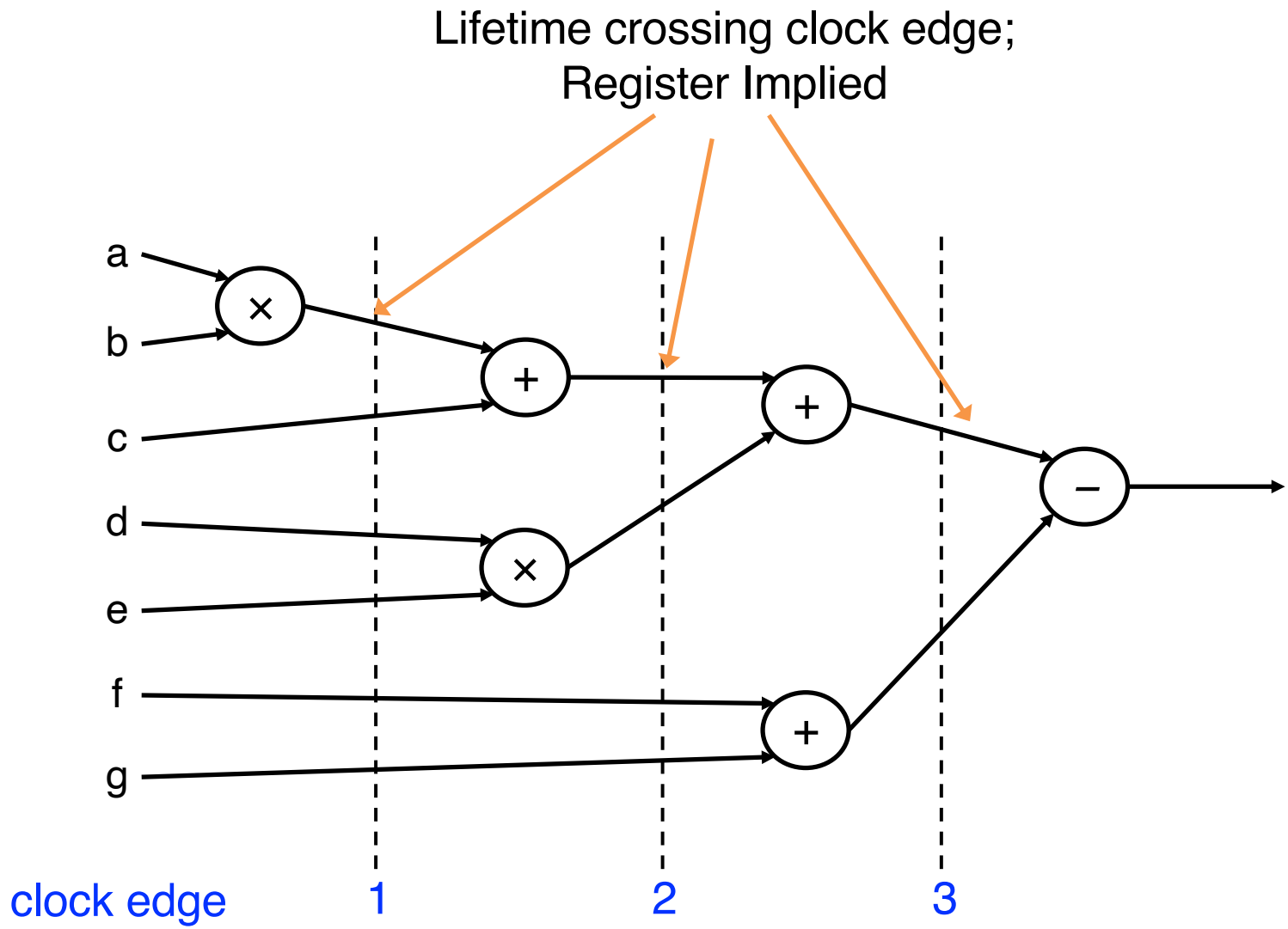
Functional Unit	Operations
Mul1	op1, op3
AddSub1	op2, op4
AddSub2	op5, op6

Binding 1

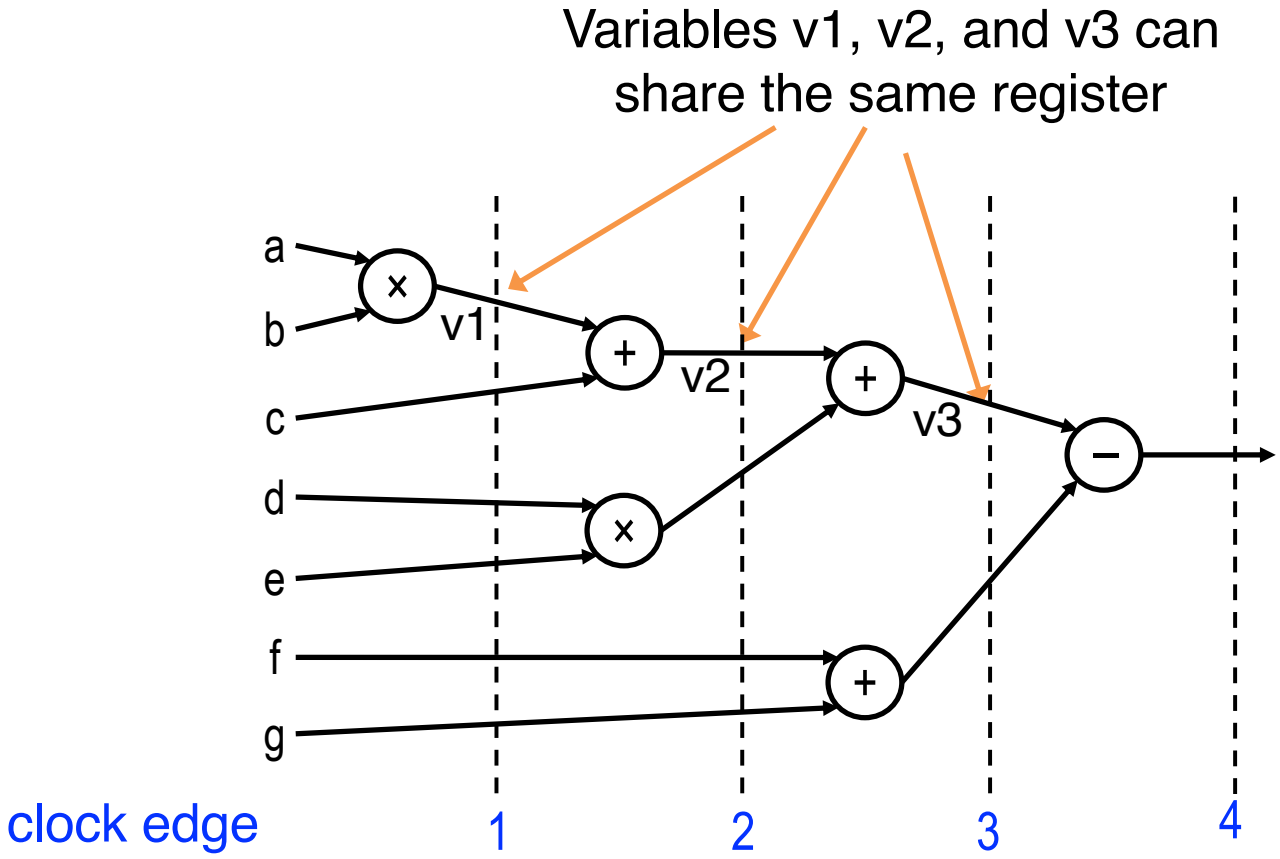
Functional Unit	Operations
Mul1	op1, op3
AddSub1	op2, op4, op6
AddSub2	op5

Binding 2

Register Binding



Variable Lifetime Analysis

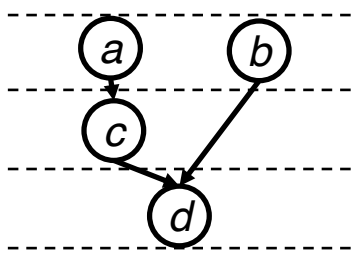


v1	**[1, 2)**		
v2		**[2, 3)**	
v3			**[3, 4)**

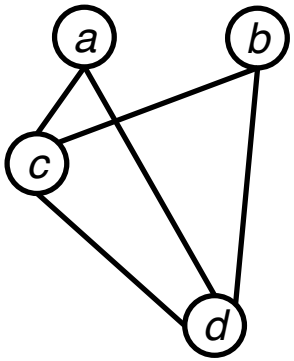
Variable lifetimes

Compatibility and Conflict Graphs

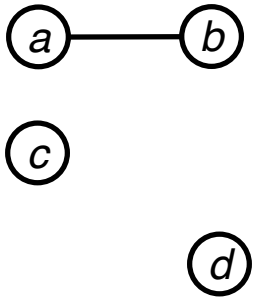
- Operation/variables compatibility
  - Same type, non-overlapping lifetimes
- **Compatibility graph**
  - Vertices: operations/variables
  - Edges: compatibility relation
- **Conflict graph**: Complement of compatibility graph



A scheduled DFG (operations have the same type)



Compatibility graph



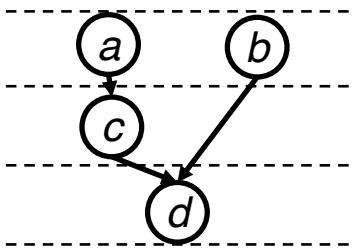
Conflict graph

Note: A compatibility/conflict graphs for variables/registers can be constructed in a similar way

Clique Cover Number and Chromatic Number

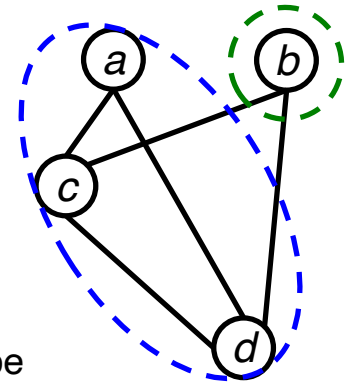
- Compatibility graph
  - Partition the graph into a **minimum number of cliques**
    - Clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge

- Conflict graph
  - Color the vertices by a **minimum number of colors** (chromatic number), where adjacent vertices cannot use the same color

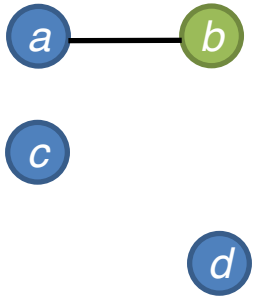


Operations have same type

A scheduled DFG



Clique partitioning on compatibility graph

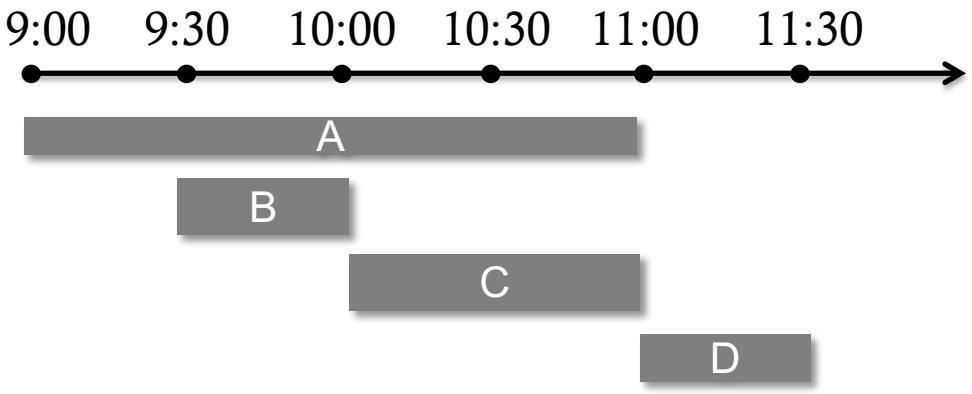


Coloring on conflict graph

Example: Meeting Assignment Problem

Meeting	Schedule (am)
A	9:00~11:00
B	9:30~10:00
C	10:00~11:00
D	11:00~11:30

Conflict graph (chromatic number?)



Gantt chart

Compatibility graph (max clique size?)

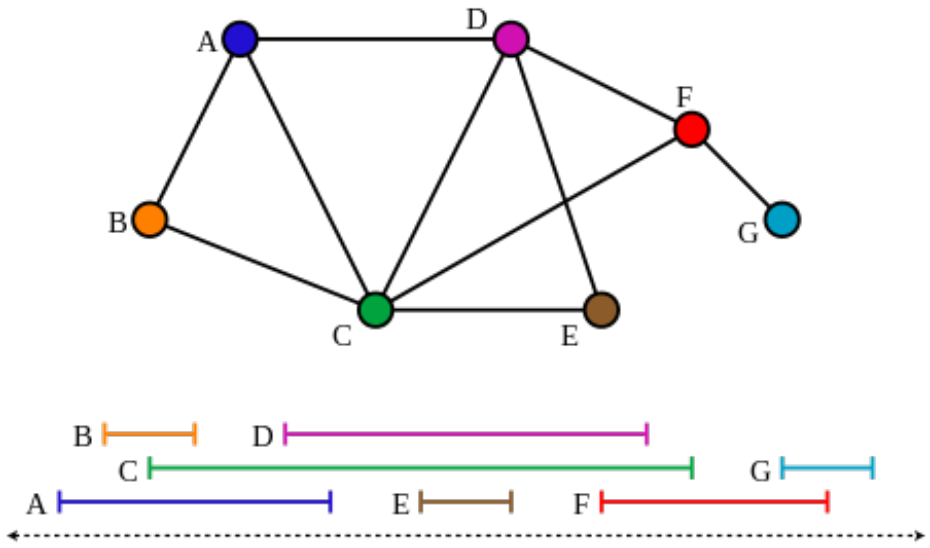
Perfect Graphs

- Clique partitioning and graph coloring problems are NP-hard on general graphs, with the exception of perfect graphs

- Definition of perfect graphs
  - For every induced subgraph, the size of the maximum (largest) clique equals the chromatic number of the subgraph
  - Examples: bipartite graphs, chordal graphs, etc.
    - Chordal graphs: every cycle of four or more vertices has a chord, i.e., an edge between two vertices that are not consecutive in the cycle.

Interval Graph

- Intersection graphs of a (multi)set of intervals on a line
  - Vertices correspond to intervals
  - Edges correspond to interval intersection
  - A special class of chordal graphs



[Figure source: en.wikipedia.org/wiki/Interval_graph]

Left Edge Algorithm

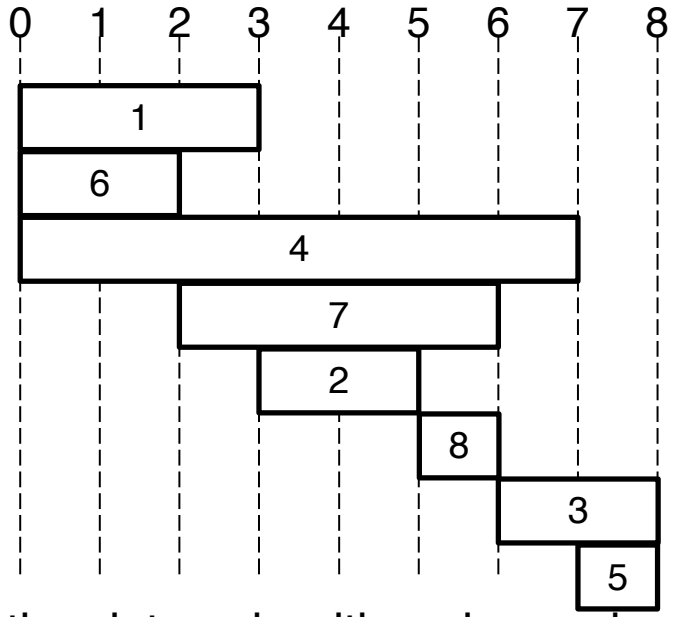
- Problem statement
  - Given: Input is a group of intervals with starting and ending time
  - Goal: Minimize the number of colors of the corresponding interval graph

```
Repeat
    create a new color group c
    Repeat
        assign leftmost feasible interval to c
    until no more feasible interval
until no more interval
```

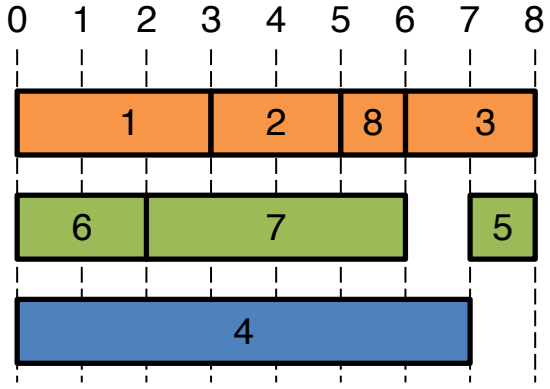
Interval are **sorted** according to their **left endpoints**

Greedy algorithm, O(nlogn) time

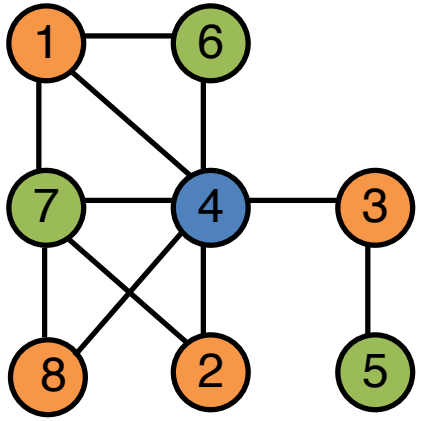
Left Edge Demonstration



Lifetime intervals with a given schedule

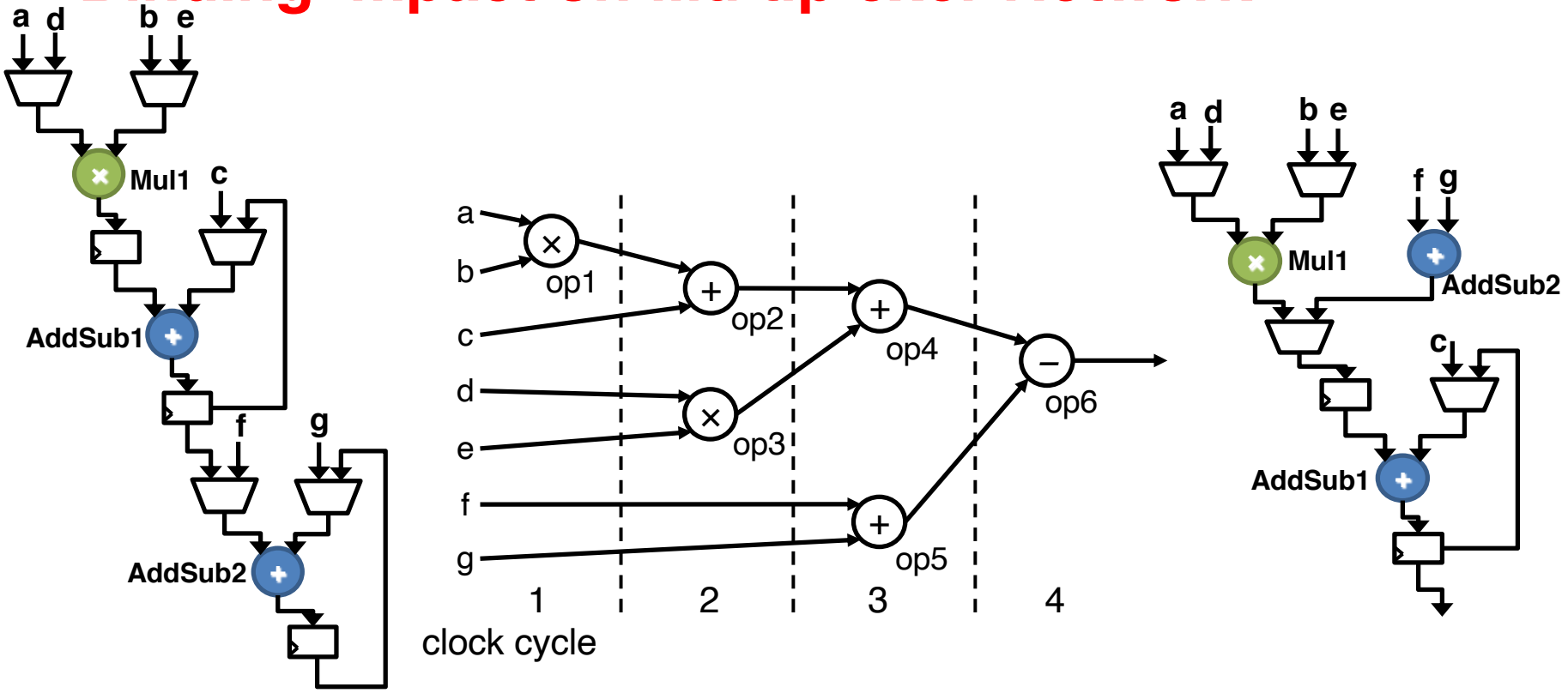


Assign colors (or tracks) using left edge algorithm



Corresponding colored conflict graph

Binding Impact on Multiplexer Network



Functional Unit	Operations
Mul1	op1, op3
AddSub1	op2, op4
AddSub2	op5, op6

Binding 1

Functional Unit	Operations
Mul1	op1, op3
AddSub1	op2, op4, op6
AddSub2	op5

Binding 2

Binding Summary

- Resource sharing directly impacts the complexity of the resulting datapath
  - # of functional units and registers, multiplexer networks, etc.
- Binding for resource usage minimization
  - Left edge algorithm: greedy but optimal for DFGs
  - **NP-hard problem with the general form of CDFG**
  - Polynomial-time algorithm exists for SSA-based register binding, although more registers are required
- Connectivity binding problem (e.g., multiplexer minimization) is NP-Hard

Parallelization Techniques

- Parallel processing
  - Emphasizes concurrency by replicating a hardware structure several times (Homogeneous)
    - High performance is attained by having all structures execute simultaneously on different parts of the problem to be solved

- Pipelining
  - Takes the approach of decomposing the function to be performed into smaller stages and allocating separate hardware to each stage (Heterogeneous)
    - Data/instructions flow through the stage of a hardware pipeline at a rate (often) independent of the length of the pipeline

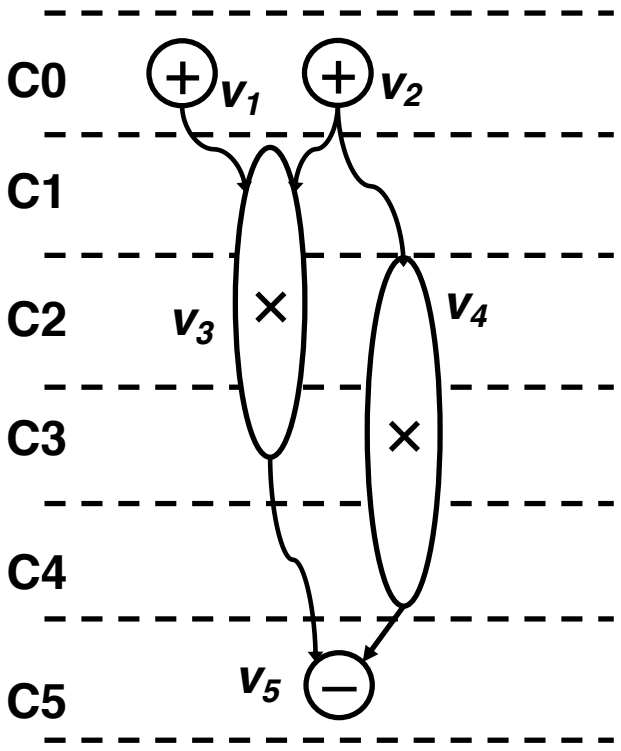
[source: Peter Kogge, The Architecture of Pipelined Computers]

Common Forms of Pipelining

- Operator pipelining
  - Fine-grained pipeline (e.g., functional units, memories)
  - Execute a sequence of operations on a pipelined resource
- Loop/function pipelining **(focus of this class)**
  - Statically scheduled
  - Overlap successive loop iterations / function invocations at a fixed rate
- Task pipelining
  - Coarse-grained pipeline formed by multiple concurrent processes (often expressed in loops or functions)
  - Dynamically controlled
  - Start a new task before the prior one is completed

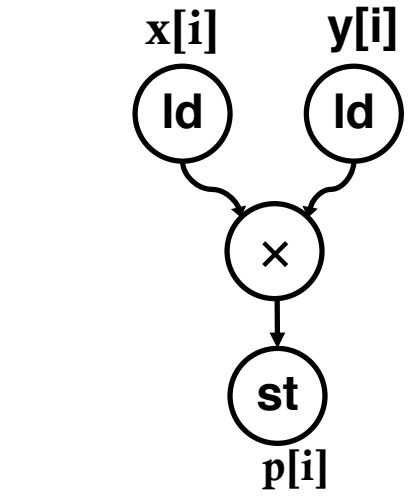
Operator Pipelining

- Pipelined multi-cycle operations
  - v_3 and v_4 can share the same pipelined multiplier (3 stages)



Loop Pipelining

- Loop pipelining is one of the most important optimizations for high-level synthesis
  - Key metric: **Initiation Interval (II)** in # cycles
  - Allows a new iteration to begin processing every II cycles, before the previous iteration is complete



ld – Load
st – Store

```
for (i = 0; i < N; ++i)
    p[i] = x[i] * y[i];
```

Pipelining

II = 1

Pipeline schedule

Pipeline Performance

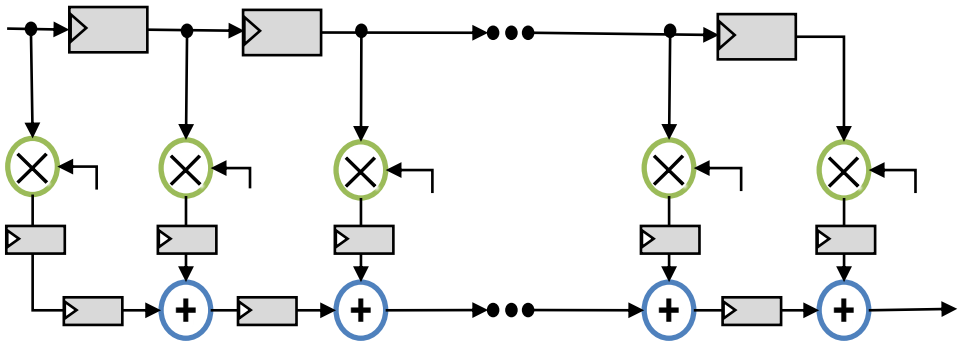
- Given a 100-iteration loop with the loop body taking 50 cycles to execute
  - If we pipeline the loop with II = 1, how many cycles do we need to complete execution of the entire loop ?
  - What about II = 2 ?

Function Pipelining

- Function pipelining: Entire function is becomes a pipelined datapath

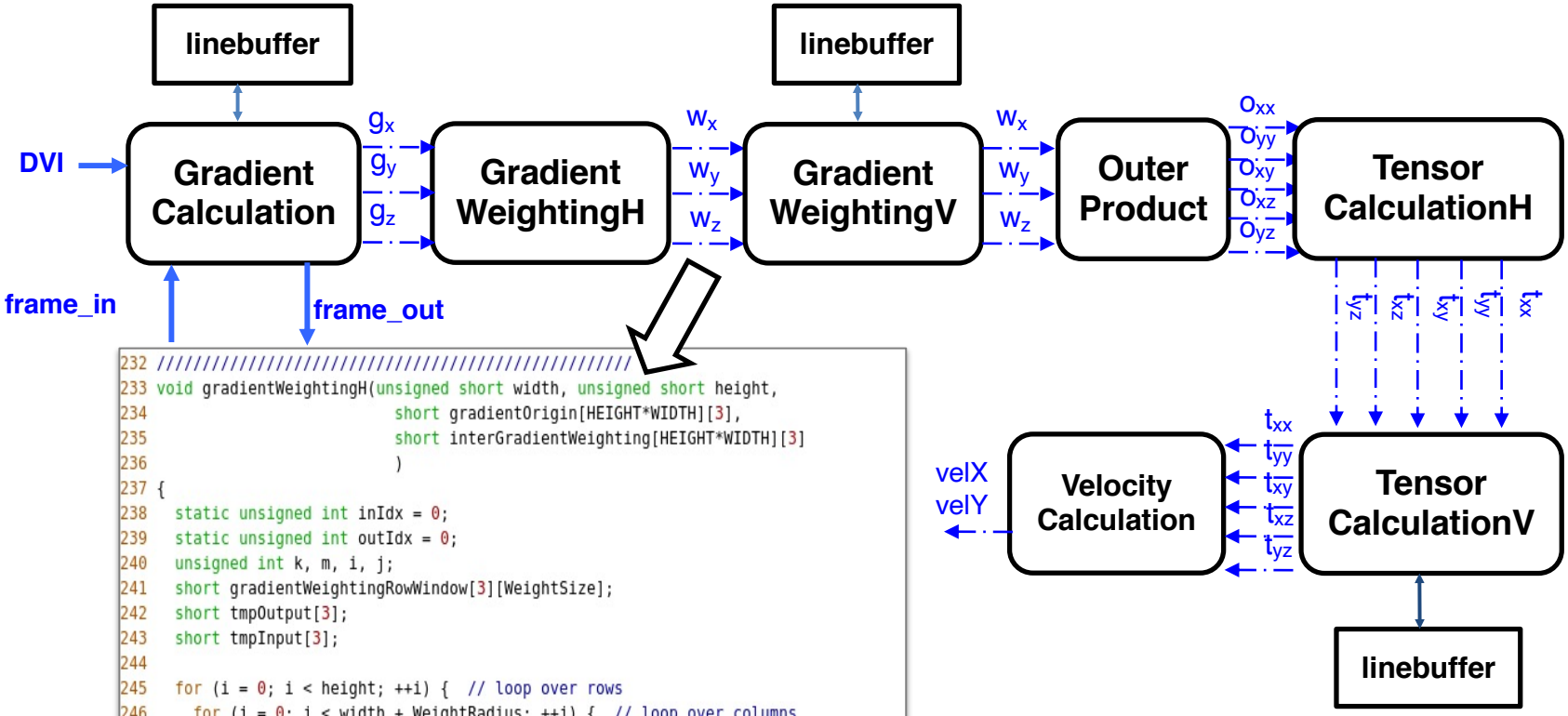
```
void fir(int *x, int *y)
{
    static int shift_reg[NUM_TAPS];
    const int taps[NUM_TAPS] =
       {1, 9, 14, 19, 26, 19, 14, 9, 1};
    int acc = 0;
    for (int i = 0; i < NUM_TAPS; ++i)
       acc += taps[i] * shift_reg[i];
    for (int i = NUM_TAPS - 1; i > 0; --i)
       shift_reg[i] = shift_reg[i-1];

    shift_reg[0] = *x;
    *y = acc;
}
```



Pipeline the entire function of the FIR filter
(with all loops unrolled and arrays completely partitioned)

Task Pipelining




```
232 ////////////////////////////////////////////////////
233 void gradientWeightingH(unsigned short width, unsigned short height,
234                          short gradientOrigin[HEIGHT*WIDTH][3],
235                          short interGradientWeighting[HEIGHT*WIDTH][3]
236                          )
237 {
238   static unsigned int inIdx = 0;
239   static unsigned int outIdx = 0;
240   unsigned int k, m, i, j;
241   short gradientWeightingRowWindow[3][WeightSize];
242   short tmpOutput[3];
243   short tmpInput[3];
244
245   for (i = 0; i < height; ++i) {  // loop over rows
246     for (j = 0; j < width + WeightRadius; ++j) {  // loop over columns
247       for (m = 0; m < 3; ++m)
248         tmpOutput[m] = 0;
249
250       if (j < width) { // make sure it read height*width times
251         for (m = 0; m < 3; ++m)
252           tmpInput[m] = gradientOrigin[inIdx][m];
253         ++inIdx;
254       }
255
256       if (j < width && i >= WeightRadius && i < height - WeightRadius) {
257         for (m = 0; m < 3; ++m) {
258           for (k = 0; k < WeightSize-1; ++k)
259             gradientWeightingRowWindow[m][k] = gradientWeightingRowWindow[m][k+1];
260           gradientWeightingRowWindow[m][WeightSize-1] = tmpInput[m];
261         }
```

A coarse-grained pipeline for the optical flow algorithm

Restrictions of Pipeline Throughput

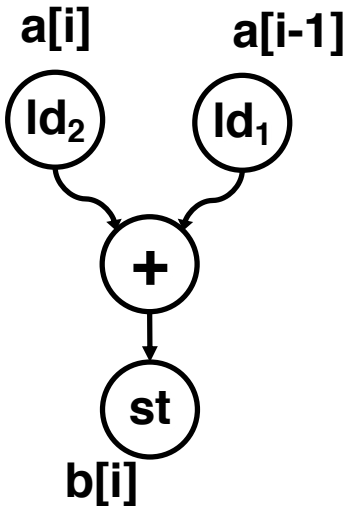
- Resource limitations
  - Limited compute resources
  - Limited Memory resources (esp. memory port limitations)
  - Restricted I/O bandwidth
  - Low throughput of subcomponent

  …

- Recurrences
  - Also known as feedbacks, carried dependences
  - **Fundamental limits of the throughput of a pipeline**

Resource Limitation

- Memory is a common source of resource contention
  - e.g. memory port limitations




```
for (i = 1; i < N; ++i)
    b[i] = a[i-1] + a[i];
```

	cycle 1	cycle 2	cycle 3	cycle 4
i = 0	ld_1	**ld_2**	+	st
i = 1	~~*II = 1*~~	**ld_1**	ld_2	+

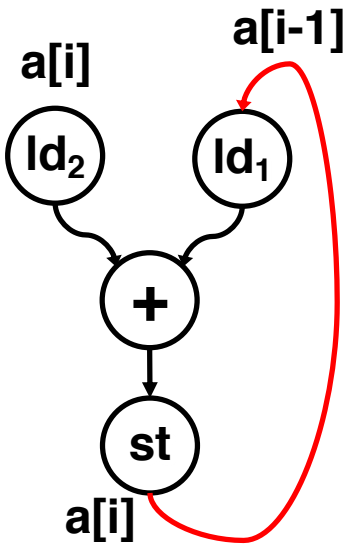
Port conflict

Only one memory read port → 1 load / cycle

Assuming 'a' and 'b' are held in two different memories

Recurrence Restriction

- Recurrences restrict pipeline throughput
  - **Computation of a component depends on a previous result from the same component**



ld – Load
st – Store

```
for (i = 1; i < N; ++i)
    a[i] = a[i-1] + a[i];
```

	cycle 1	cycle 2	cycle 3	cycle 4
i = 0	ld_1 ld_2	+	**st**	
i = 1	~~**II = 1**~~	**ld_1** ld_2	+	st

Assume chaining is not possible on memory reads (i.e., ld) and writes (i.e., st) due to cycle time constraint

Next Lecture

- More Pipelining

Acknowledgements

- These slides contain/adapt materials developed by
  - Prof. Deming Chen (UIUC)
  - Prof. Jason Cong (UCLA)