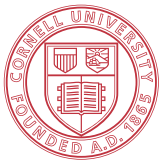
ECE 6775
High-Level Digital Design Automation
Fall 2024

More Pipelining
Resource Sharing



Cornell University

CSL

Announcements

- HW 2 due Friday (free extension to next Wed)

- Lab 4 (NN acceleration) will be posted soon
  - TWO students per group
  - **Start looking for a teammate now**

- Midterm on Tuesday 10/22
  - In class, 75 mins
  - Open notes, open book, closed Internet
  - Coverage: Lectures 01~11, 13, 14 (excluding NN tutorial)

Agenda

- Modulo scheduling case studies
- Systolic arrays: combining parallel processing and pipelining
  - Uniform recurrence equations
  - Case study on matrix multiplication
- Resource sharing basics
  - Sub-problems: functional unit, register, and connectivity binding problems
  - Key concepts: compatibility and conflict graphs

Recap: Calculating Lower Bound of II

- Minimum possible II (MII)
  - MII = max (ResMII, RecMII)
  - A lower bound, not necessarily achievable

- Resource constrained MII (ResMII)
  - $\text{ResMII} = \max_i \lceil \text{OPs}(r_i) / \text{Limit}(r_i) \rceil$
    OPs(r): number of operations that use resource of type r
    Limit(r): number of available resources of type r

- Recurrence constrained MII (RecMII)
  - $\text{RecMII} = \max_i \lceil \text{Latency}(c_i) / \text{Distance}(c_i) \rceil$
    Latency(c_i): total latency in dependence cycle c_i
    Distance(c_i): total distance in dependence cycle c_i

Case Study: Prefix Sum

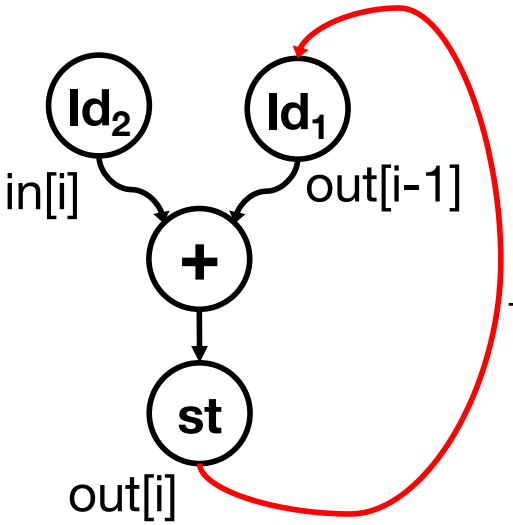
- Prefix sum computes a cumulative sum of a sequence of numbers
  - commonly used in many applications such as radix sort, histogram, etc.

```
void prefixsum ( int in[N], int out[N] )
  out[0] = in[0];
  for ( int i = 1; i < N; i++ ) {
    #pragma HLS pipeline II=?
    out[i] = out[i-1] + in[i];
  }
}
```

```
out[0] = in[0];
out[1] = in[0] + in[1];
out[1] = in[0] + in[1] + in[2];
out[1] = in[0] + in[1] + in[2] + in[3];
...
```

Prefix Sum: RecMII

- Loop-carried dependence exists between to reads on ‘out’
  - Assume chaining is not possible on memory reads (ld) and writes (st) due to target cycle time
  - RecMII = 3



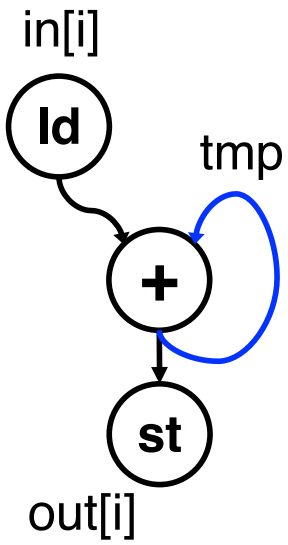
ld – Load
st – Store

```
out[0] = in[0];
for ( int i = 1; i < N; i++ )
  out[i] = out[i-1]+ in[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

Prefix Sum: Code Optimization

- Introduce an intermediate variable ‘tmp’ to hold the running sum from the previous ‘in’ values
  - Shorter dependence cycle leads to RecMII = 1



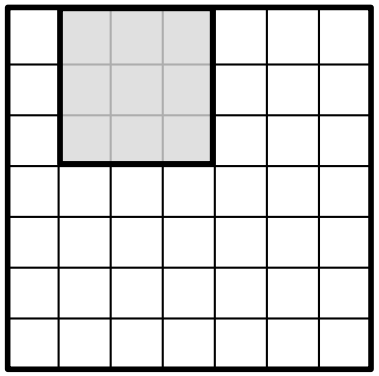
ld – Load
st – Store

```
int tmp = in[0];
for  ( int i = 1; i < N; i++ )  {
  tmp += in[i];
  out[i] = tmp;
}
```

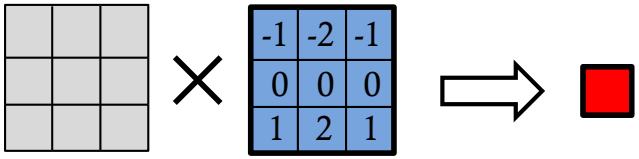
	cycle 1	cycle 2	cycle 3	cycle 4
i = 0	ld	+	st	
i = 1	***II* = 1**	ld	+	st

2D Convolution: MII

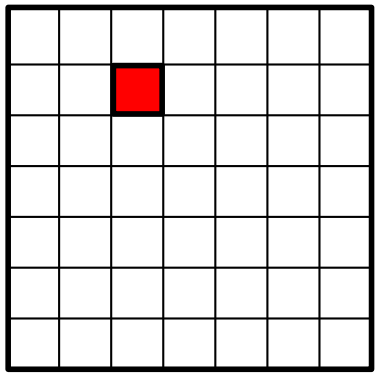
```
for (r = 1; r < R; r++)
    for (c = 1; c < C; c++) {
      #pragma HLS pipeline II=?
      for (i = 0; i  < 3; i++)
        for (j = 0; j < 3; j++)
          out[r][c] += img[r+i-1][c+j-1] * f[i][j];
     }
```



Input image frame



A K by K **dot product** is performed for each output pixel (K=3 here)



Output image frame

Exercise: Pipelining 3x3 Convolution

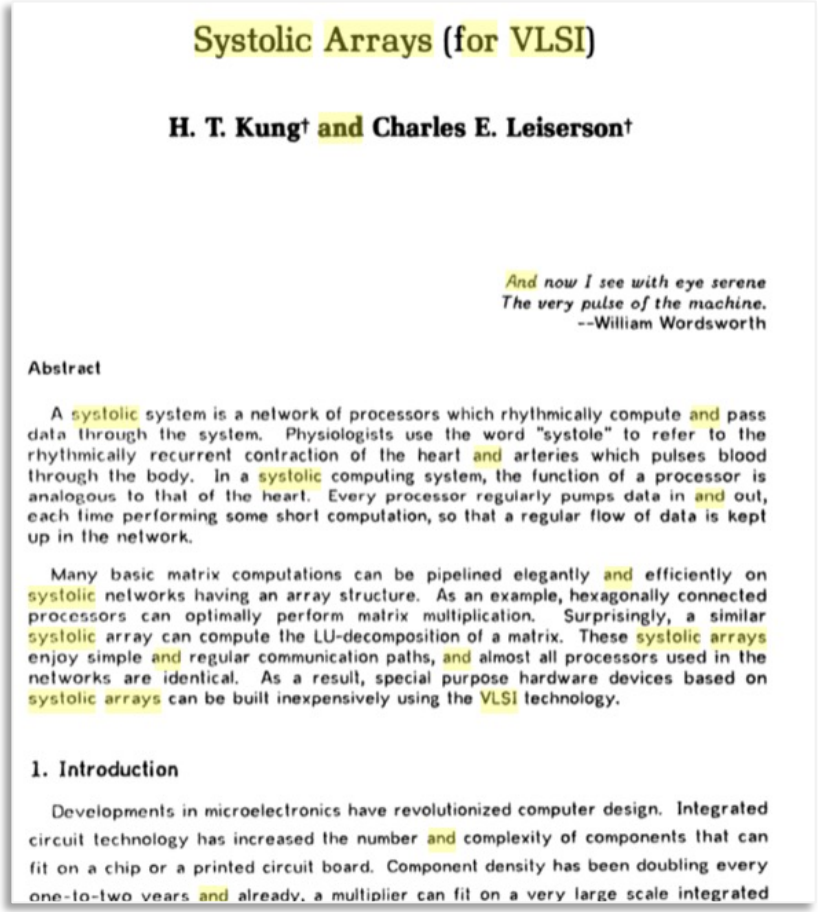
```
for (r = 1; r < R; r++)
    for (c = 1; c < C; c++) {
      #pragma HLS pipeline II=?
      for (i = 0; i  < 3; i++)
        for (j = 0; j < 3; j++)
          out[r][c] += img[r+i-1][c+j-1] * f[i][j];
     }
```

- Inner loops, “i” & “j”, are automatically unrolled
- The 3x3 filter array, “f”, is partitioned into 9 registers
- The entire input image, “img”, is stored in an on-chip SRAM with **two read ports**

ResMII = ? What about RecMII?

Recap: Systolic Arrays

- An array of processing elements (PEs) that process data in a systolic manner using nearest-neighbor communication
  - Systolic means “data flows from memory in a rhythmic fashion, passing through many processing elements before it returns to memory” – H.T. Kung



Systolic Arrays (for VLSI)

H. T. Kung† and Charles E. Leiserson†

And now I see with eye serene
The very pulse of the machine.
--William Wordsworth

Abstract

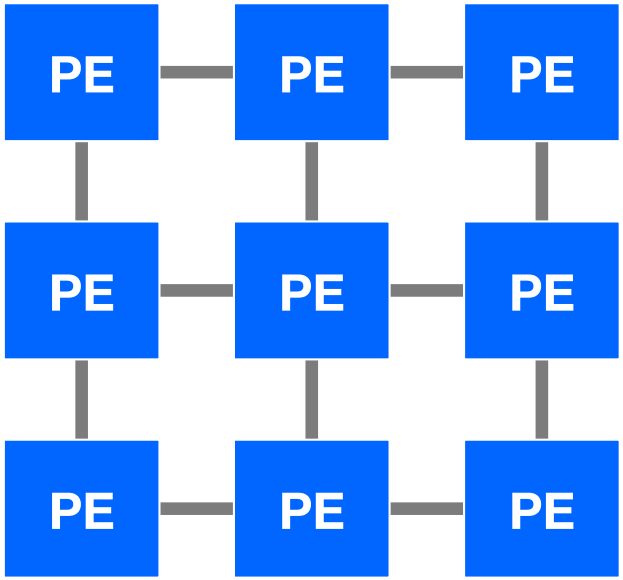
A systolic system is a network of processors which rhythmically compute and pass data through the system. Physiologists use the word "systole" to refer to the rhythmically recurrent contraction of the heart and arteries which pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network.

Many basic matrix computations can be pipelined elegantly and efficiently on systolic networks having an array structure. As an example, hexagonally connected processors can optimally perform matrix multiplication. Surprisingly, a similar systolic array can compute the LU-decomposition of a matrix. These systolic arrays enjoy simple and regular communication paths, and almost all processors used in the networks are identical. As a result, special purpose hardware devices based on systolic arrays can be built inexpensively using the VLSI technology.

1. Introduction

Developments in microelectronics have revolutionized computer design. Integrated circuit technology has increased the number and complexity of components that can fit on a chip or a printed circuit board. Component density has been doubling every one-to-two years and already, a multiplier can fit on a very large scale integrated

In Sparse Matrix Proceedings, 1978



parallel processing + pipelining

+ Simple & regular design
+ Massive parallelism
+ Short interconnection
+ Balancing compute with I/O

Uniform Recurrence Equations (UREs)

- Any **systolic algorithm** can be described by a set of UREs
  - i.e., an n-dimensional loop nest where the recurrences (inter-iteration dependences) must have constant distances

$$\mathbf{y = A * x}$$

```
for (int i = 0; i < N; i++)
  y[i] = 0;
  for (int j = 0; j < N; j++)
    y[i] += A[i, j] * x[j]
```

Matrix Vector Multiplication (MV) in UREs

$$Z[i,j] = 0, when\ j = 0$$
$$Z[i,j] = Z[i,j-1] + A[i,j] \cdot x[j], when\ j > 0$$
$$y[i] = Z[i,N-1]$$

$$\mathbf{C = A * B}$$

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    C[i, j] = 0;
    for (int k = 0; k < N; k++)
      C[i, j] += A[i, k] * B[k, j]
```

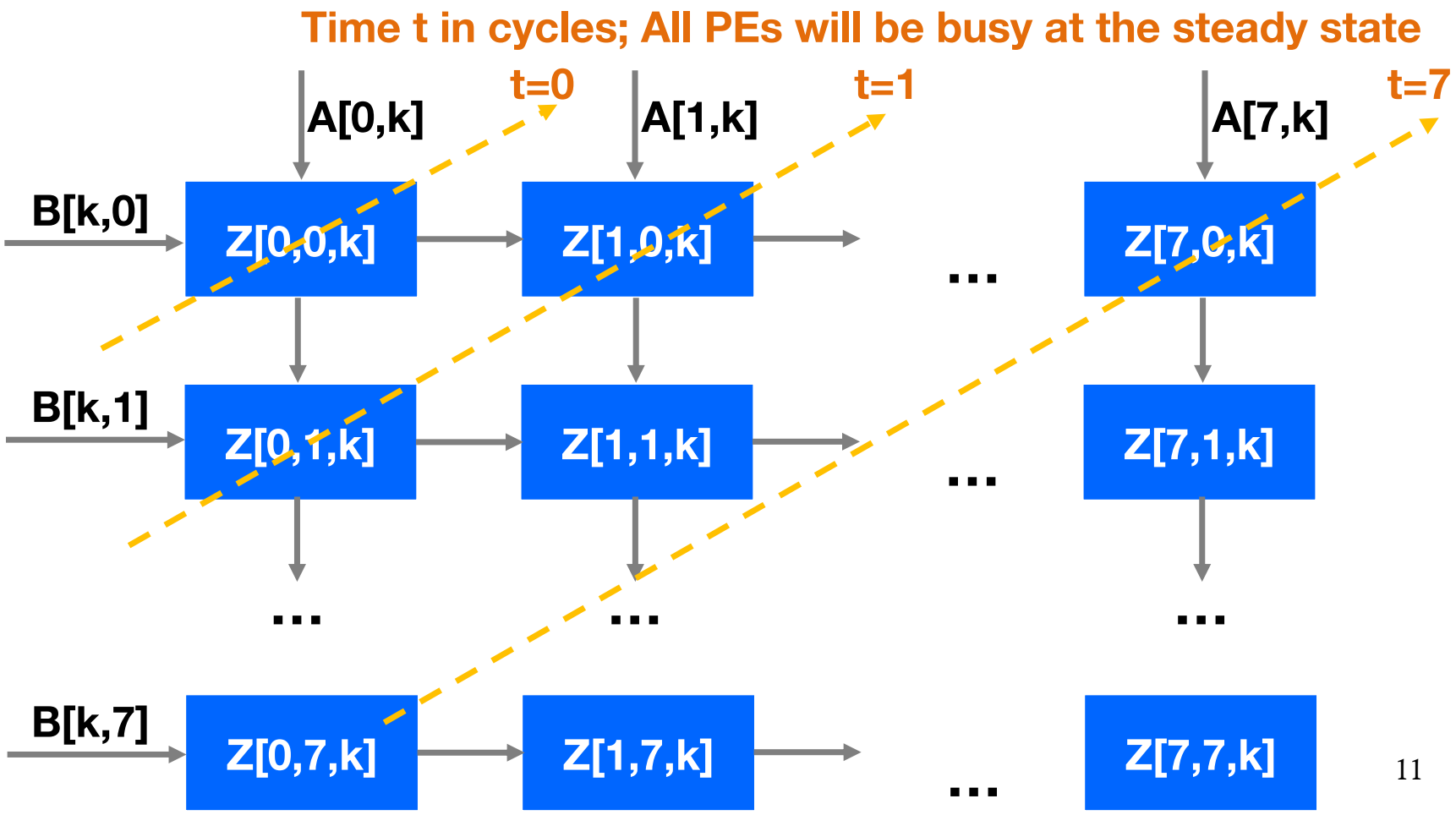
Matrix Matrix Multiplication (MM) in UREs

$$Z[i,j,k] = 0, when\ k = 0$$
$$Z[i,j,k] = Z[i,j,k-1] + A[i,k] \cdot B[k,j], when\ k > 0$$
$$C[i,j] = Z[i,j,N-1]$$

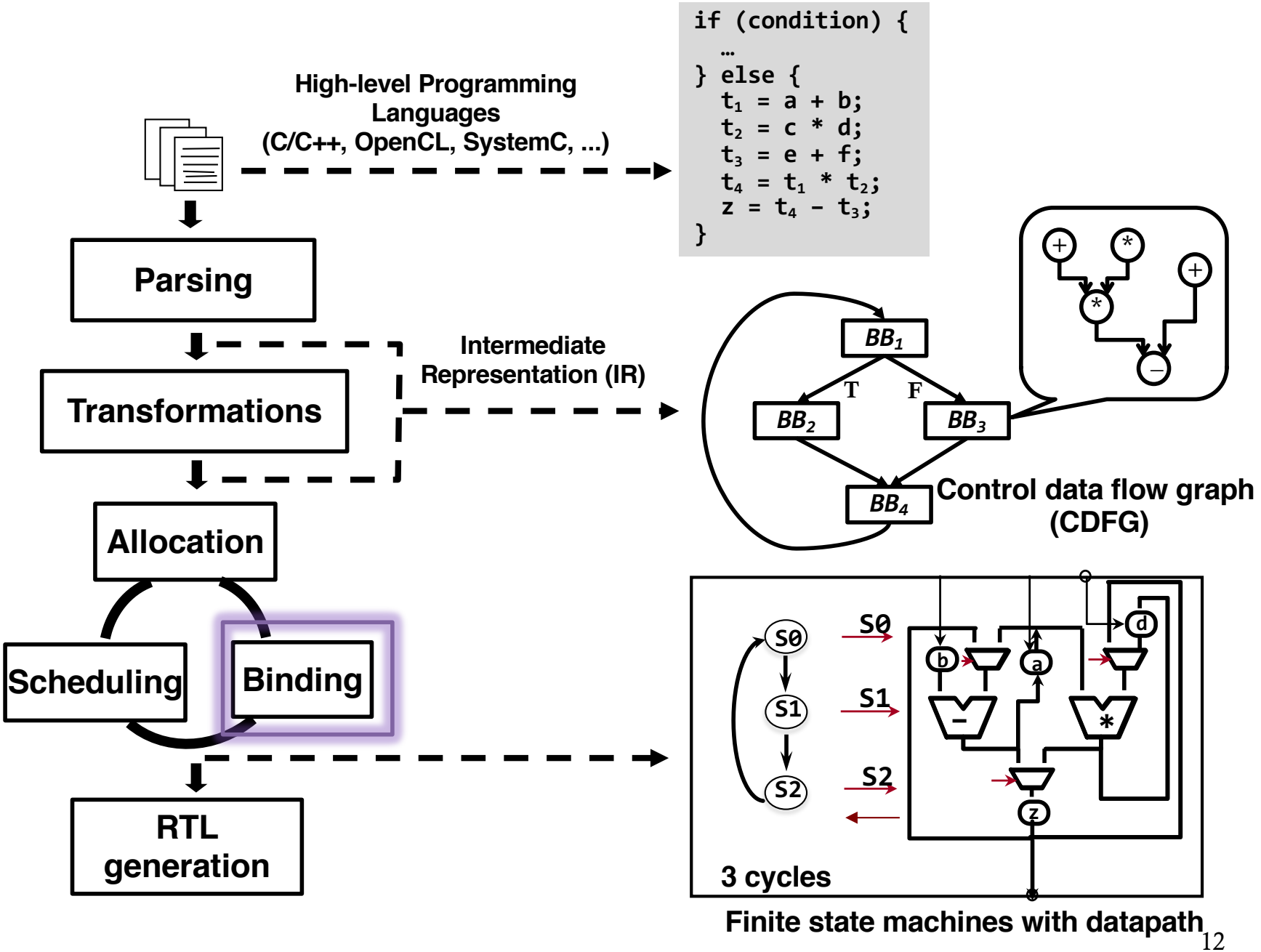
Mapping MM to a Systolic Array

- Map the n-dimensional iteration space into a physical array of PEs

$$\mathbf{C = A * B} \quad \begin{aligned} Z[i,j,k] &= 0, when\ k = 0 \\ Z[i,j,k] &= Z[i,j,k-1] + A[i,k] \cdot B[k,j], when\ k > 0 \\ C[i,j] &= Z[i,j,N-1] \end{aligned}$$



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

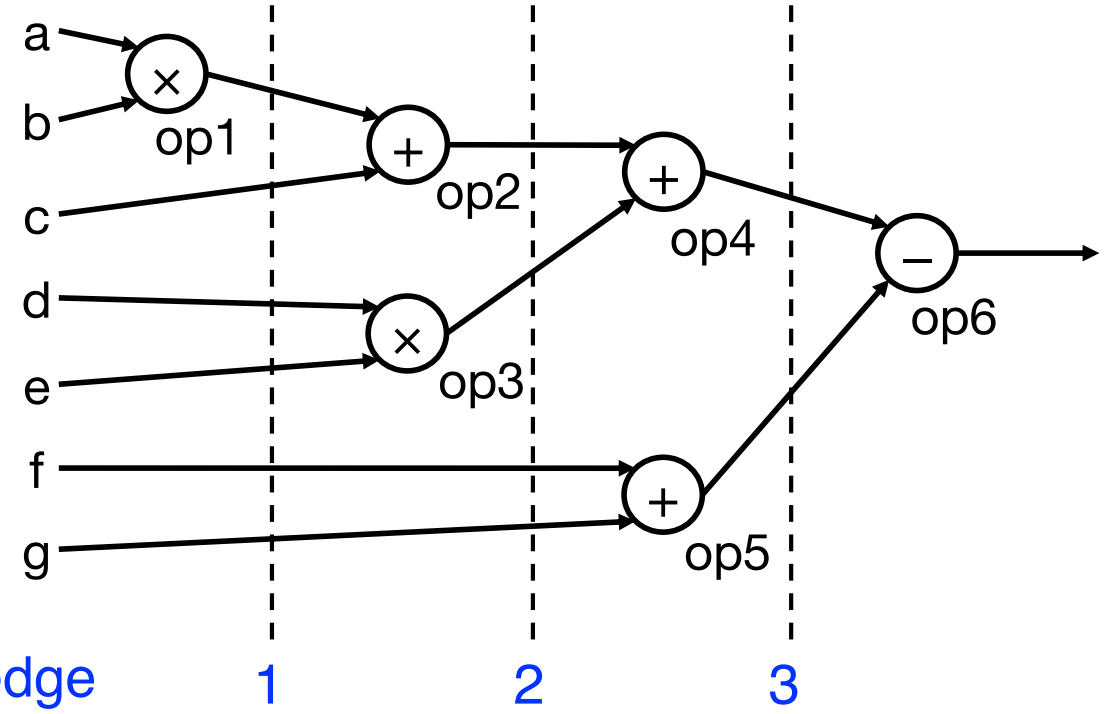
- Functional units (registers) are shared by operations (variables) of same type whose *lifetimes* do not overlap

- **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

- **In this lecture, we assume no pipelining to simplify discussion**
  - With pipelining (modulo scheduling), we use slots to determine overlaps rather than control steps

Operation Binding

a scheduled DFG
(unpipelined)



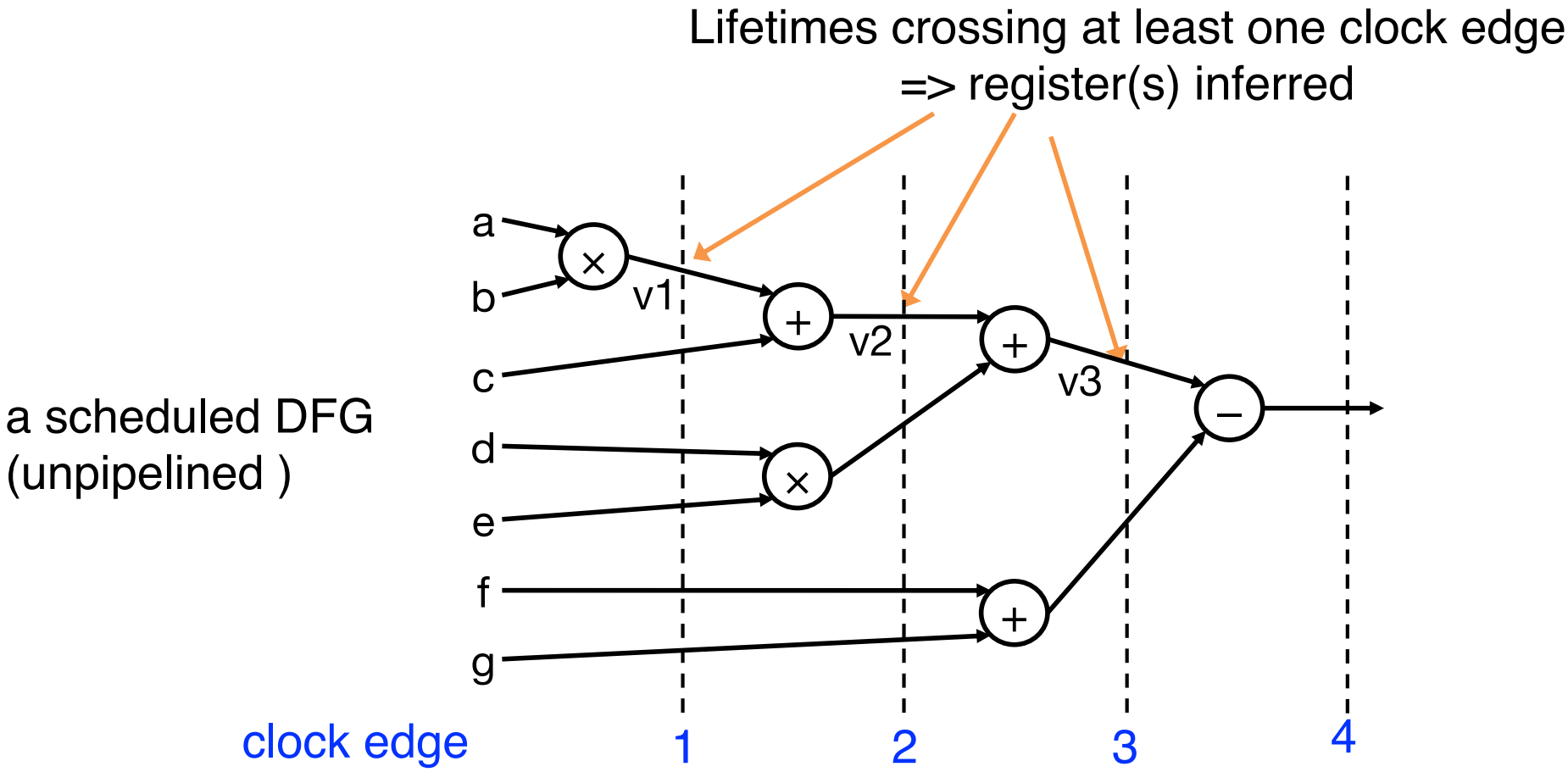
Functional Unit	Operations
Mul1	op1, op3
AddSub1	op2, op4
AddSub2	op5, op6

Binding 1

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

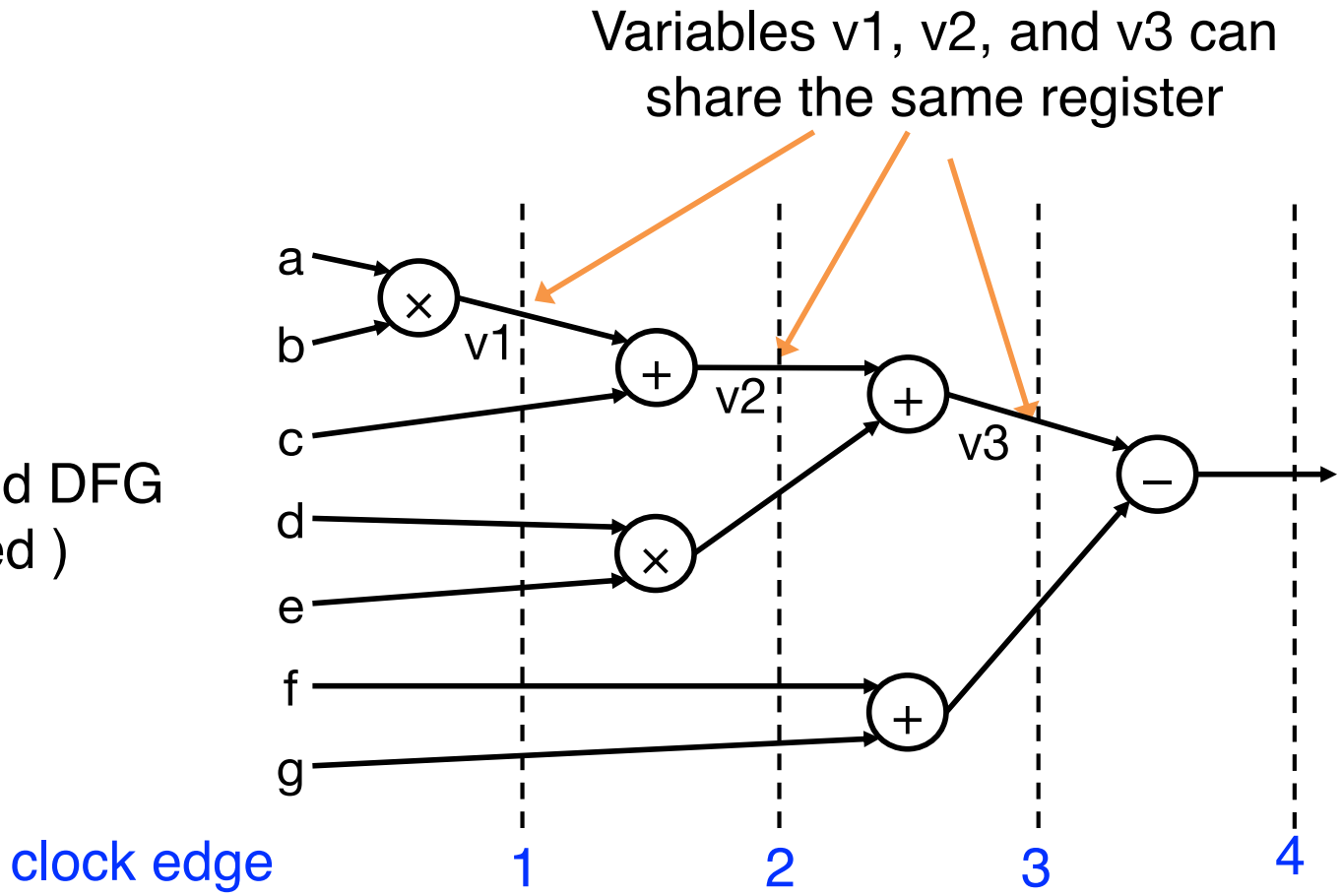
Binding 2

Register Binding



Variable Lifetime Analysis

a scheduled DFG
(unpipelined)

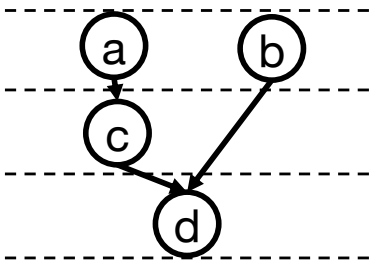


v1	**[1, 2)**		
v2		**[2, 3)**	
v3			**[3, 4)**

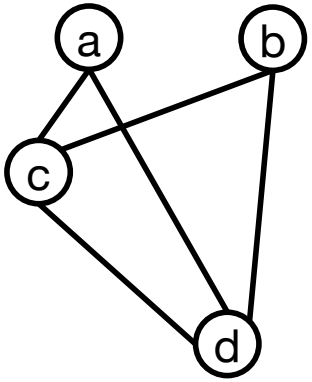
Variable lifetimes [birth-time, death-time)

Compatibility and Conflict Graphs

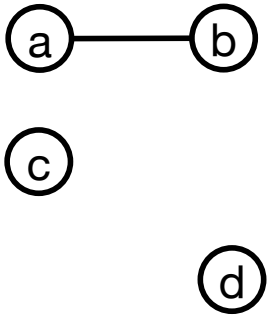
- Operation/variables compatibility
  - Same type, non-overlapping lifetimes
- **Compatibility graph**
  - Vertices: operations/variables
  - Edges: compatibility relation
- **Conflict graph**: Complement of compatibility graph



A scheduled DFG (unpipelined; operations have the same type)



Compatibility graph

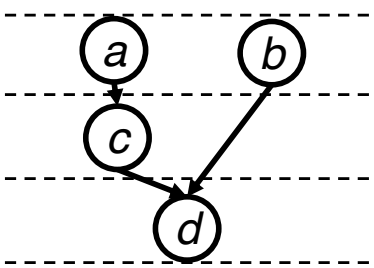


Conflict graph

Clique Cover Number and Chromatic Number

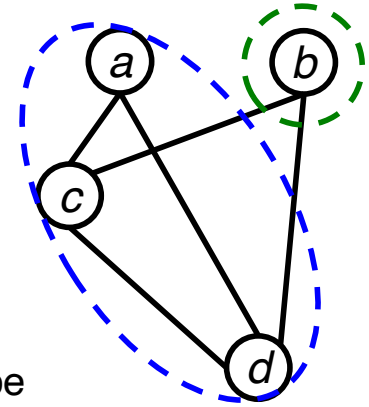
- Compatibility graph
  - Partition the graph into a **minimum number of cliques**
    - Clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge

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

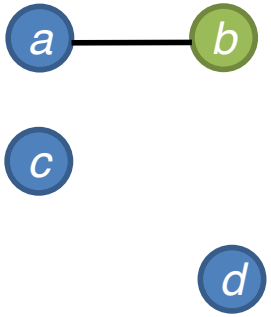


Operations have same type

A scheduled DFG



Clique partitioning on compatibility graph



Coloring on conflict graph

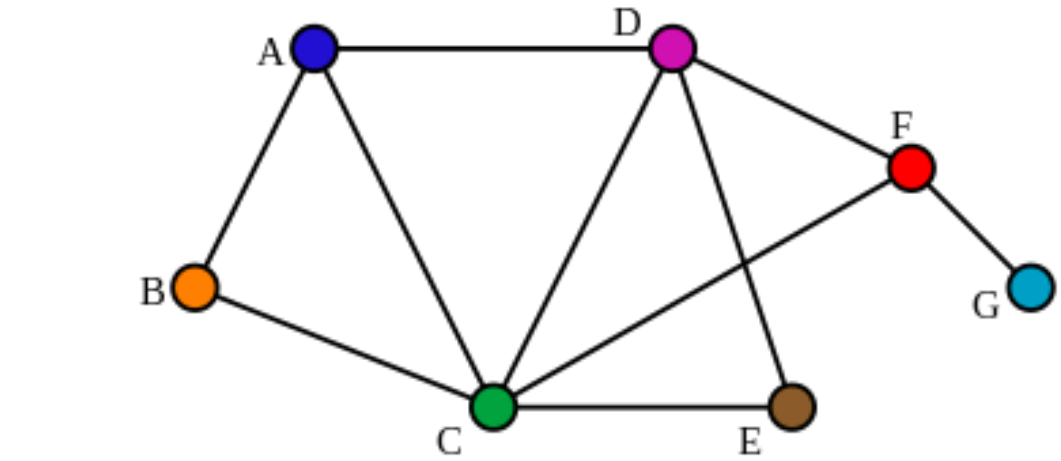
Perfect Graphs

- 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; a *chord* is 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

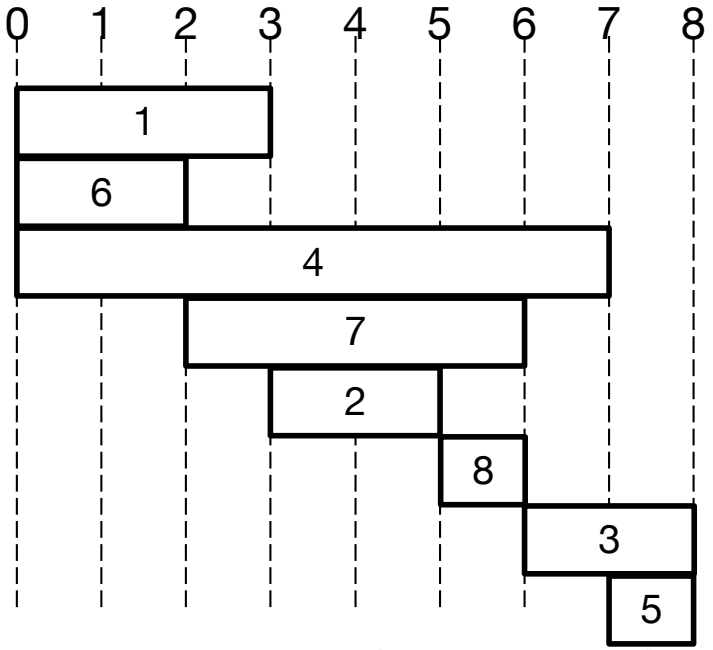
- Problem statement
  - Given: Input is a group of intervals with starting and ending time
  - Goal: Minimize the number of colors of the corresponding interval graph

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

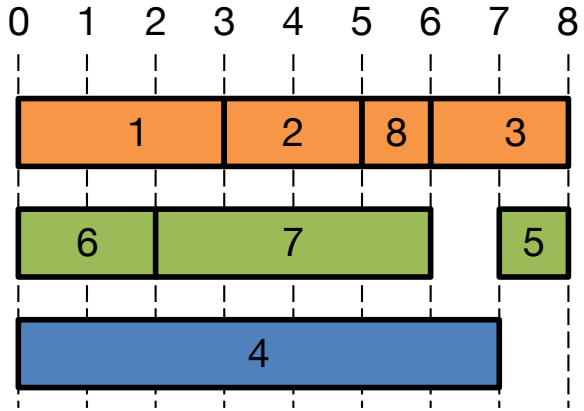
Interval are **sorted** according to their **left endpoints**

Greedy algorithm, O(nlogn) time

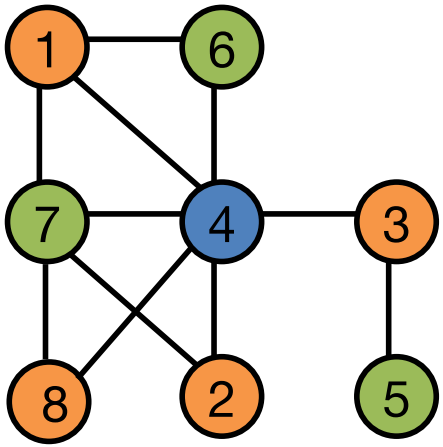
Left Edge Demonstration



Lifetime intervals with a given schedule

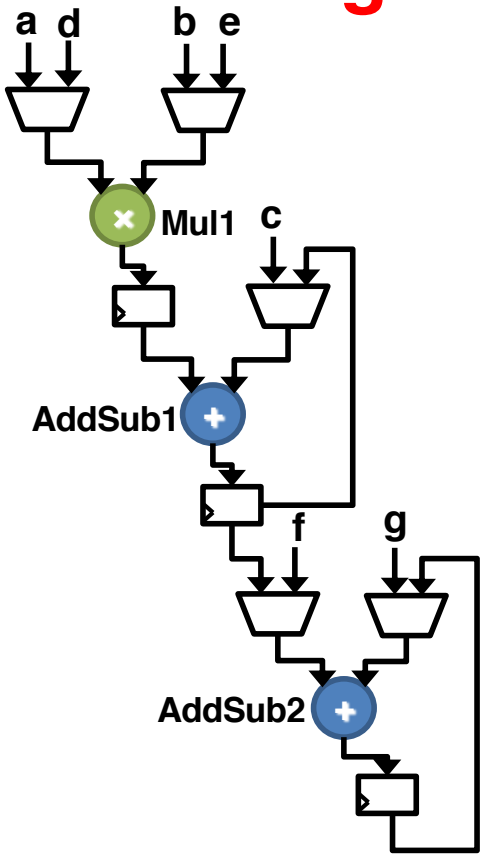


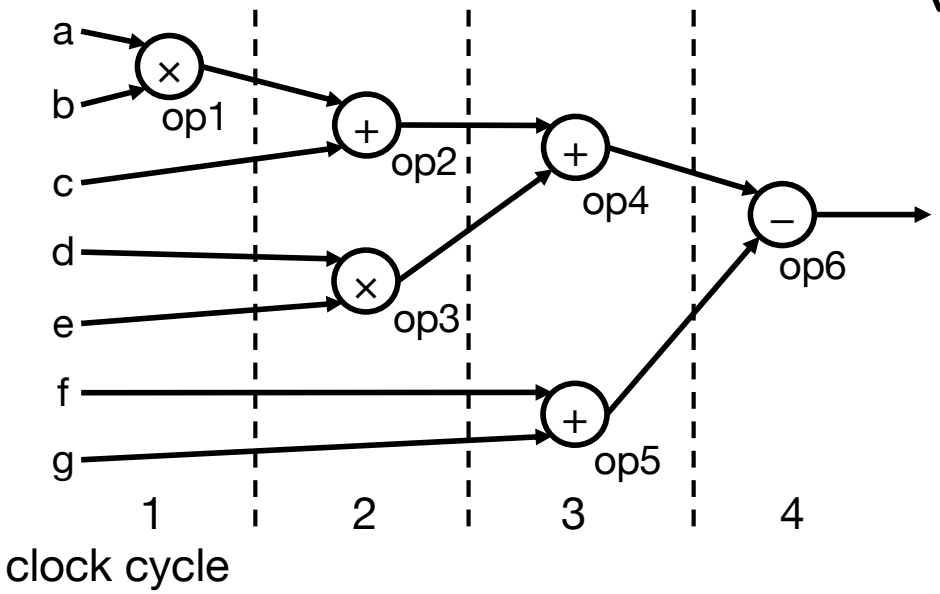
Assign colors (or tracks)
using left edge algorithm

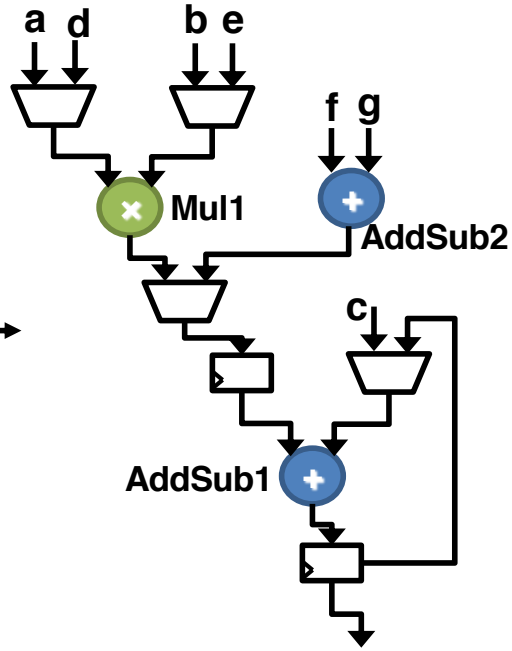


Corresponding
colored conflict graph

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

Acknowledgements

- These slides contain/adapt materials developed by
  - Prof. Jason Cong (UCLA)
  - Prof. Ryan Kastner (UCSD)
  - Dr. Stephen Neuendorffer (AMD Xilinx)