ECE 5775
High-Level Digital Design Automation
Fall 2022

More Pipelining

Cornell University

CSL

Announcements

- HW 2 released: Due Friday, but no late penalty
- Yichi Zhang (ECE PhD student) will give a tutorial on deep neural networks (DNNs) this Thursday
- Lab 4 (on DNN acceleration) will be posted next week
  - TWO students per group
  - **Start looking for a teammate**

Agenda

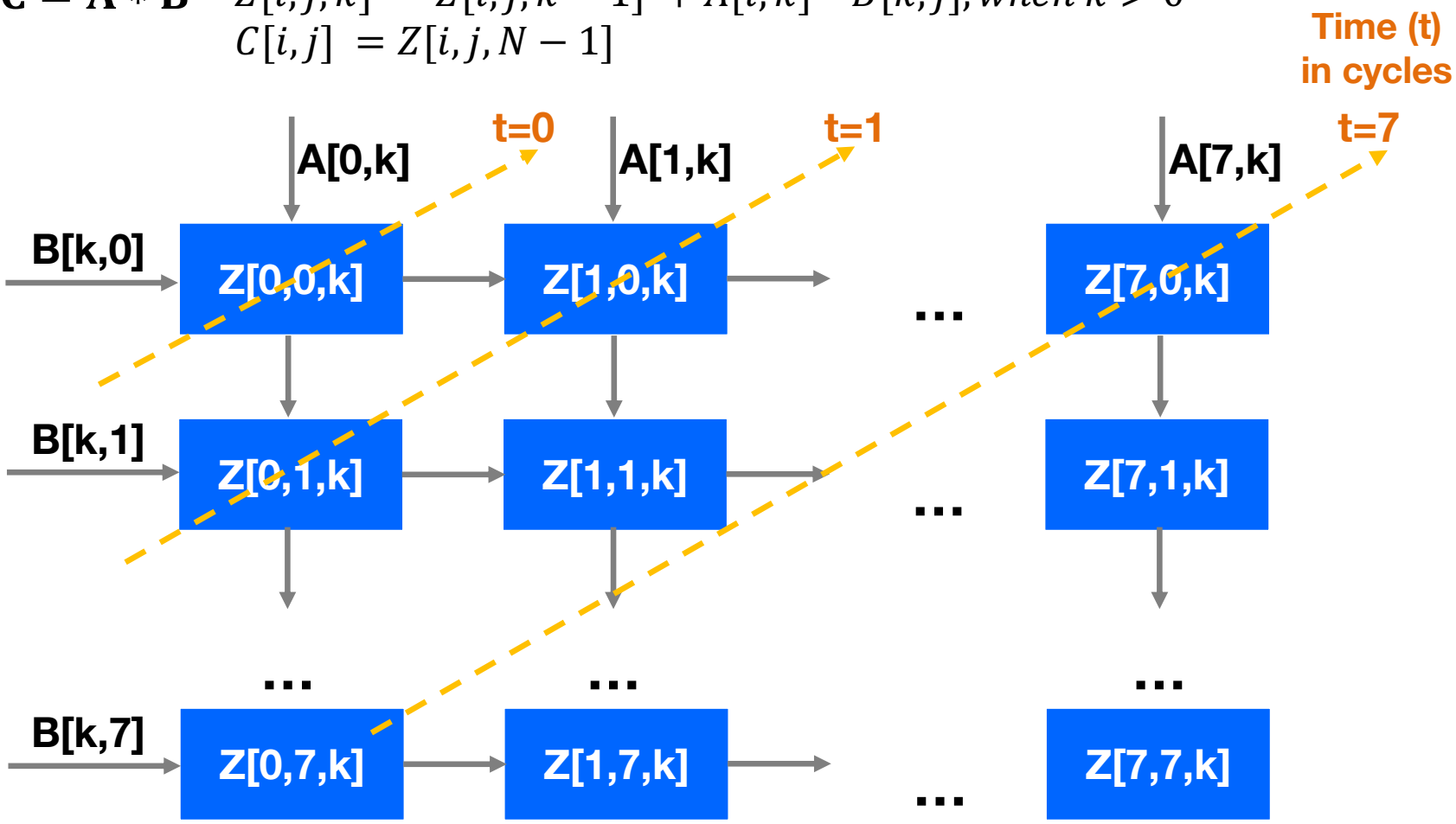
- Modulo scheduling concepts
- Extending SDC formulation for pipelining
- Case studies

Recap: Mapping MM to a Systolic Array

Uniform Recurrence Equations (UREs)

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

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



Recap: Restrictions of Pipeline Throughput

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

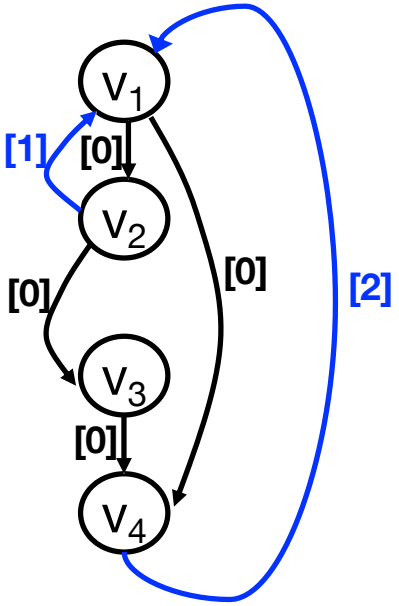
  …

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

Dependence Graph

- Data dependences of a loop are often represented by a dependence graph
  - Forward edges: **Intra-iteration** (loop-independent) dependences
  - Back edges: **Inter-iteration (loop-carried)** dependences
  - Edges are annotated with **distance** values: number of iterations separating the two dependent operations involved

- Recurrence manifests itself as a **circuit** in the dependence graph



Edges annotated with distance values

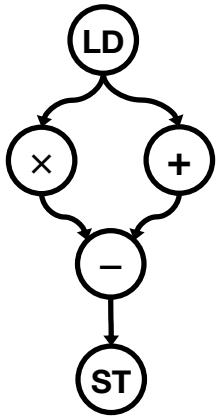
Modulo Scheduling

- A regular form of loop (or function) pipelining technique
  - Also applies to software pipelining in compiler optimization
  - **Loop iterations use the same schedule, which are initiated at a constant rate**
  - Typical objective: minimize II under resource constraints
    - **NP-hard in general:** optimal polynomial time solution exists without recurrences or resource constraints

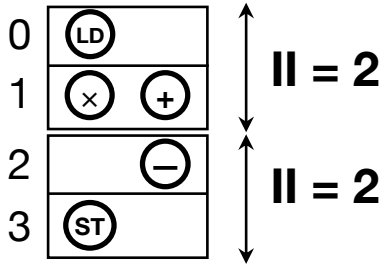
- Advantages of modulo scheduling
  - Cost efficient: No code or hardware replication
  - Easy to analyze: **steady state determines performance & resource**

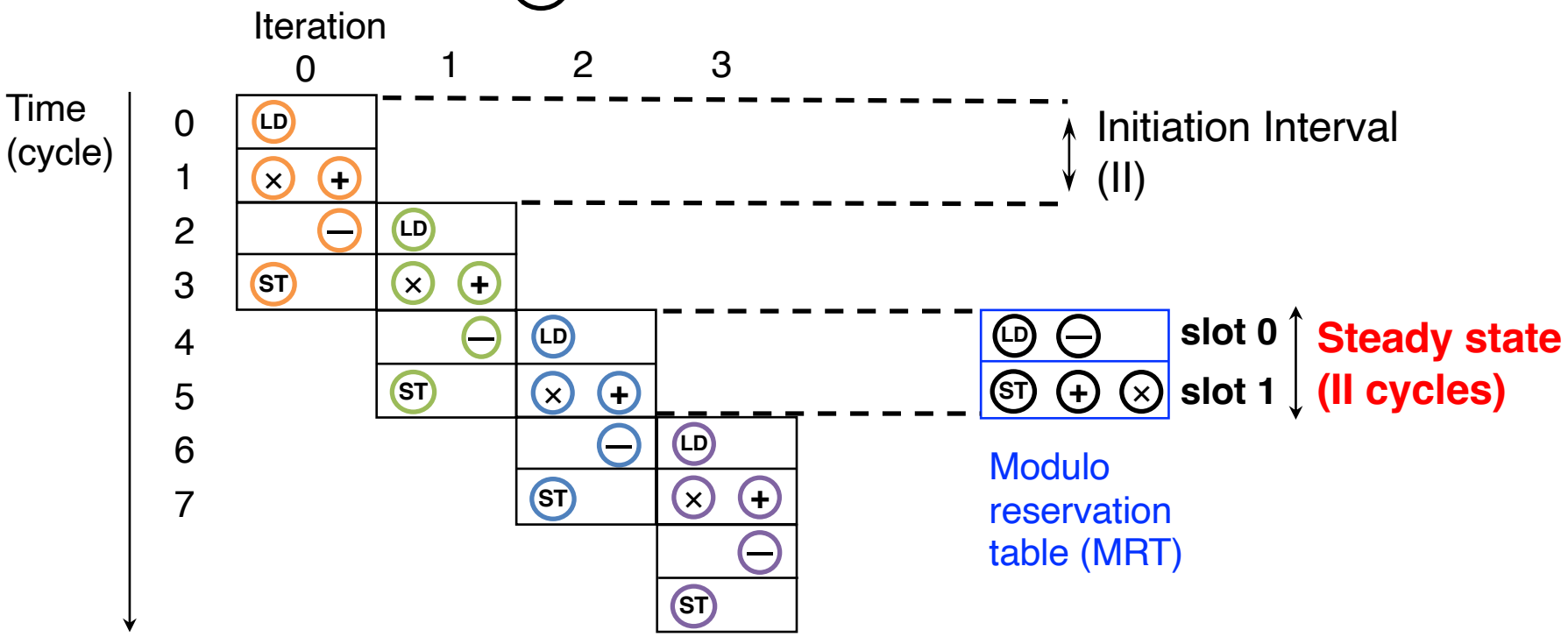
Modulo Scheduling Example

Dependence graph of the loop body



Schedule of the body

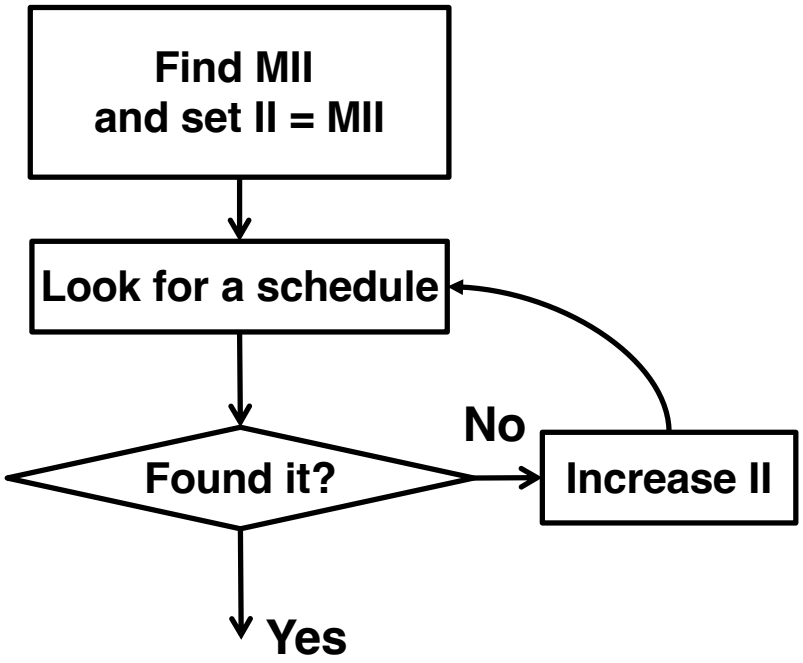




Steady state determines both performance and resource usage

Algorithmic Scheme for Modulo Scheduling

- Common scheme of heuristic algorithms
  - Find a lower bound on II: MII = max (ResMII, RecMII)
  - Look for a schedule with the given II
  - If a feasible schedule not found, increase II and try again



Calculating Lower Bound of Initiation Interval

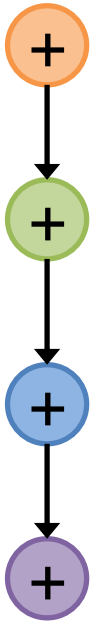
- Minimum possible II (MII)
  - MII = max (ResMII, RecMII)
  - A lower bound, not necessarily achievable

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

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

Minimum II due to Resource Limits (ResMII)

Dependence



Resource Allocation & Binding

4 adders

	time 0	1	2	3	4	5
a0	i0	i1	i2	i3	i4	i5
a1		i0	i1	i2	i3	i4
a2			i0	i1	i2	i3
a3				i0	i1	i2

0, 1, 2, 3, … : time (clock cycles)
a0, a1, a2, a3 : available adders
i0, i1, i2, … : loop iterations

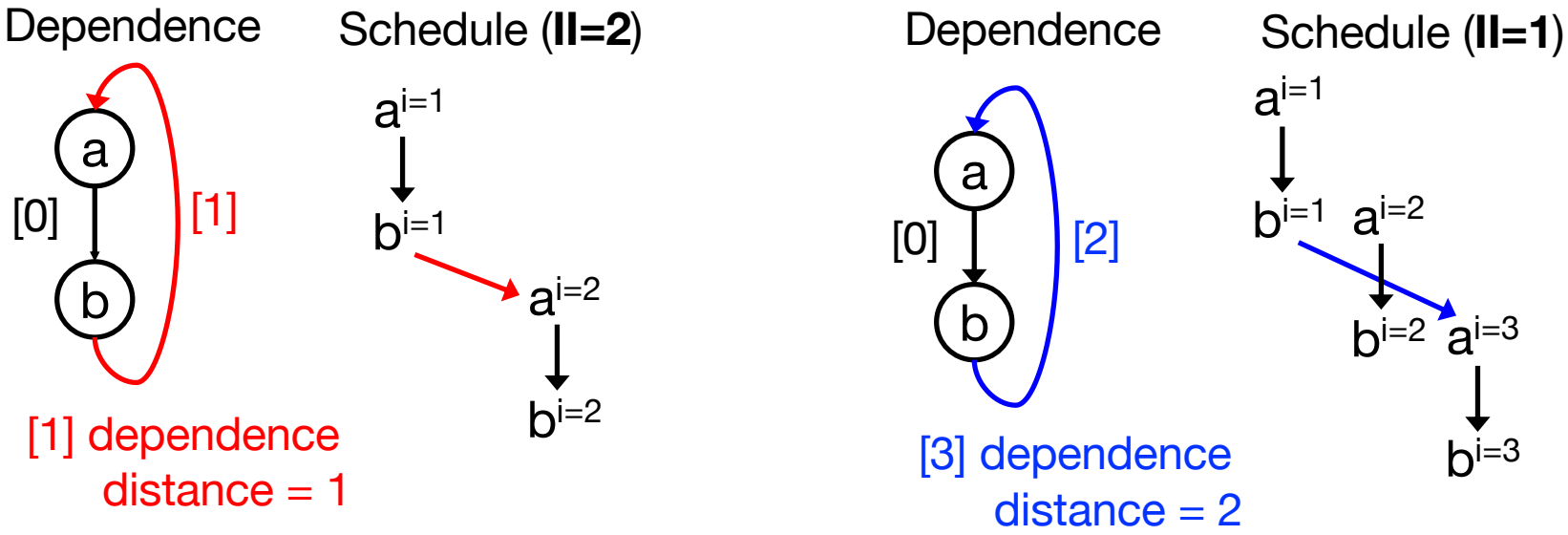
2 adders

a0	i0	i0	i1	i1	i2	i2	i3	i3
a1			i0	i0	i1	i1	i2	i2

due to limited resources, cannot initiate iterations less than 2 cycles apart

- Compute ResMII*:* Max among all types of resources
  - **ResMII = $\max_i$ ⌈OPs(r_i) / Limit(r_i)⌉**

Minimum II due to Recurrences (RecMII)

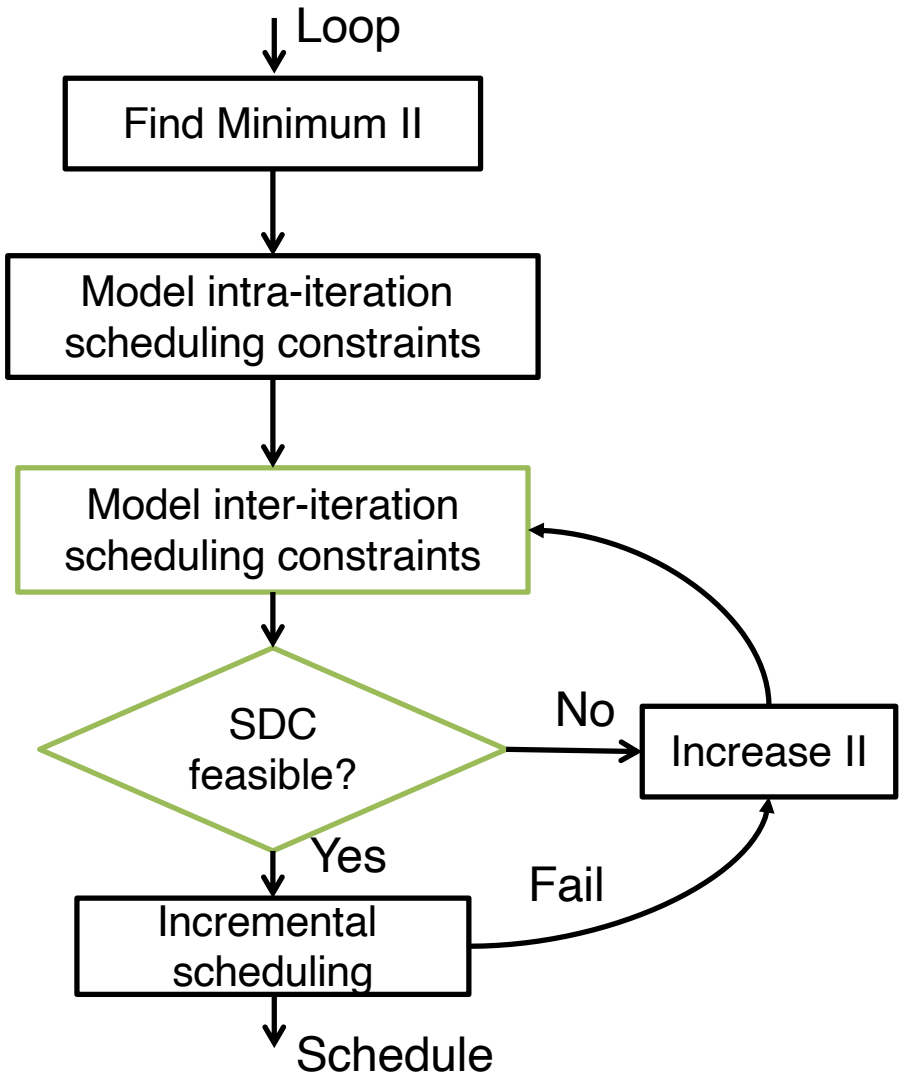


Assume (1) single-cycle operations; (2) no chaining

- Compute recurrence MII (RecMII)
  - Max among all circuits: **RecMII = $\max_i$ ⌈Latency(c_i) / Distance(c_i)⌉**
  - **Latency(c)**: sum of operation latencies along circuit *c*
  - **Distance(c)**: sum of dependence distances along circuit *c*

SDC-Based Modulo Scheduling

- The SDC formulation can be extended to support modulo scheduling
  - Unifies intra-iteration and inter-iteration scheduling constraints in a single SDC
  - Iterative algorithm with efficient incremental SDC update

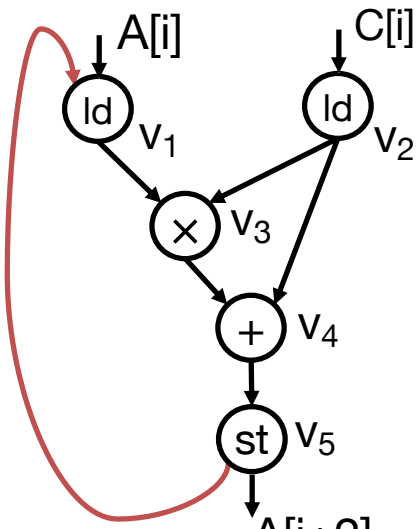


[Z. Zhang & B. Liu, ICCAD 2013]

Modeling Loop-Carried Dependence with SDC

- Loop-carried dependence u → v with Distance(u, v) = K

```
for (i = 0; i < N-2; i++)
{
    B[i] = A[i] * C[i];
    A[i+2] = B[i] + C[i];
}
```

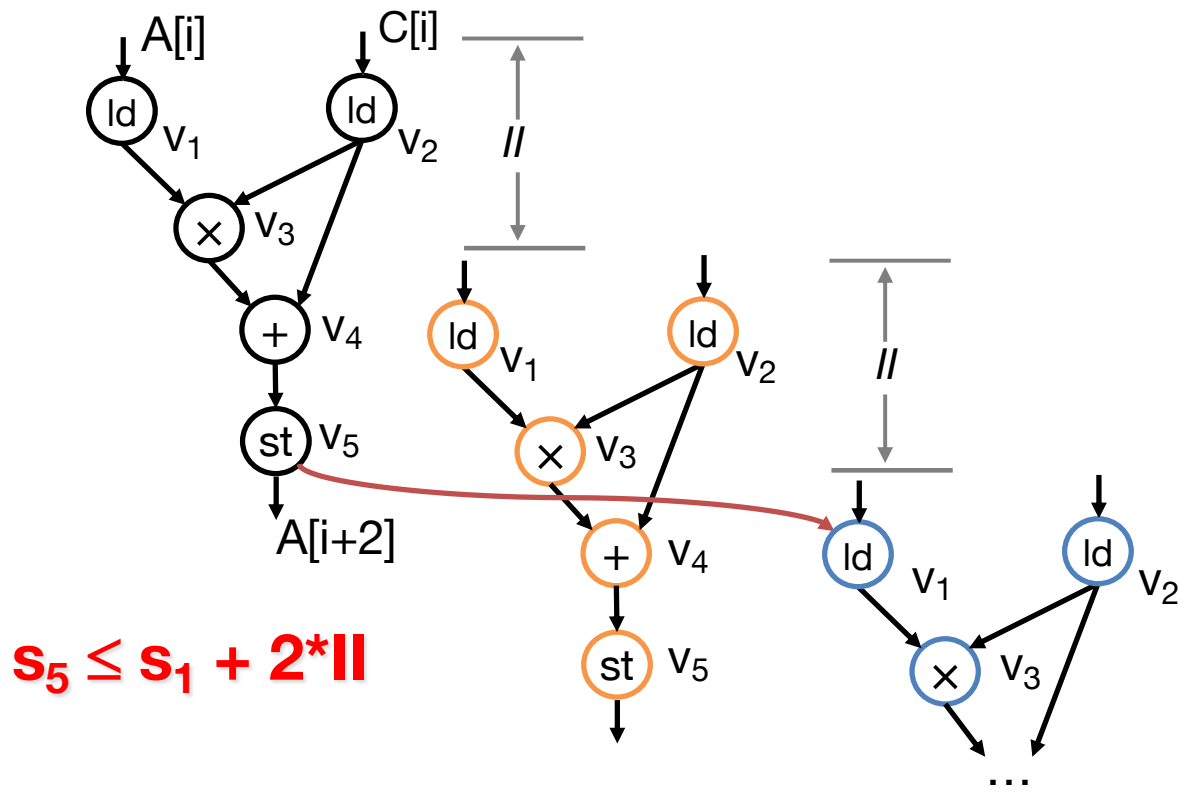


K = Dist(v_5, v_1) = 2

Modeling Loop-Carried Dependence with SDC

- Loop-carried dependence u → v with Distance(u, v) = K
  $s_u + Lat_u \leq s_v + K*II$

```
for (i = 0; i < N-2; i++)
{
    B[i] = A[i] * C[i];
    A[i+2] = B[i] + C[i];
}
```



$s_5 \leq s_1 + 2*II$

Case Study: Prefix Sum

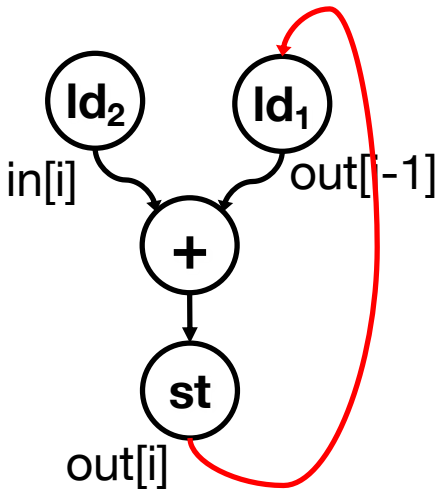
- Prefix sum computes a cumulative sum of a sequence of numbers
  - commonly used in many applications such as radix sort, histogram, etc.

```
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



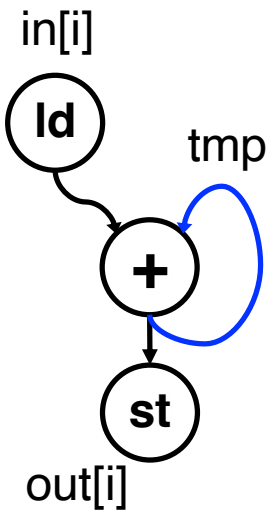
ld – Load
st – Store

```
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 circuit leads to RecMII = 1



ld – Load
st – Store

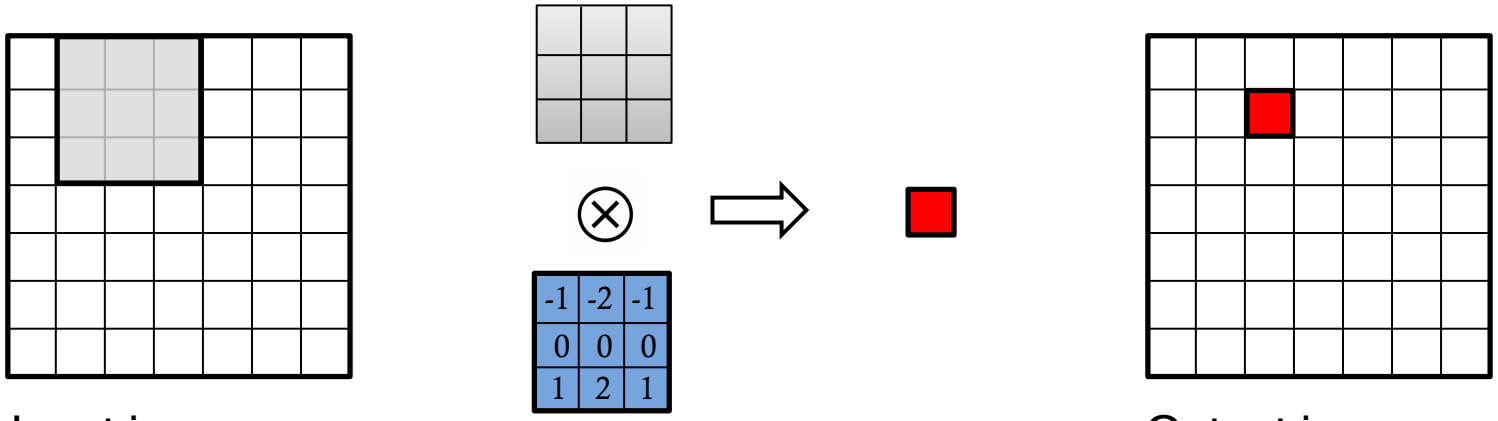
```
int tmp = in[0];
for  ( int i = 1; i < N; i++ )  {
  tmp += in[i];
  out[i] = tmp;
}
```

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

Case Study: Convolution for Image Processing

- **Convolution** is pervasive in image/video processing and ML – performed over overlapping windows (aka stencils)

$$(Img \otimes f)_{\left[n+\frac{k-1}{2},m+\frac{k-1}{2}\right]} = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} Img_{[n+i][m+j]} \cdot f_{[i,j]}$$



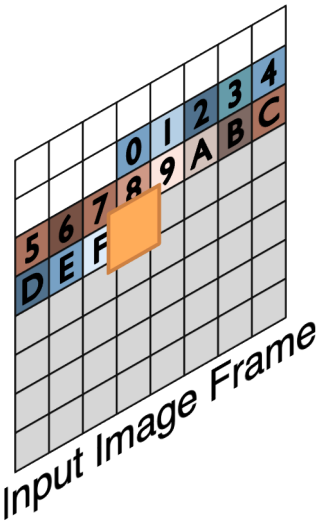
Exercise: Pipelining 3x3 Convolution

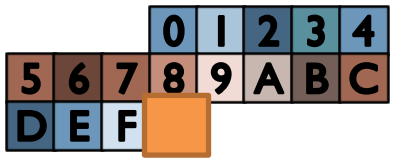
```
for (r = 1; r < H; r++)
    for (c = 1; c < W; c++) {
      #pragma HLS pipeline II=?
      for (i = 0; i  < 3; i++)
        for (j = 0; j < 3; j++)
          out[r][c] += img[r+i-1][c+j-1] * f[i][j];
     }
```

- 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 buffer with **two read ports**

ResMII = ? What about RecMII?

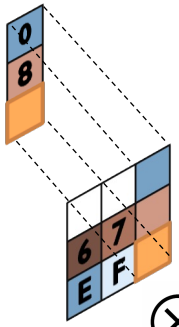
Achieving II=1 for 3x3 Convolution using a Line Buffer and Shift Registers





Pixels in **line buffer**
(stores 2 lines using on-chip SRAM)

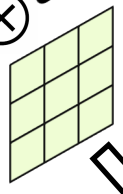
1. **Push 3 pixels into shift registers** –
   1 new pixel +
   2 from line buffer

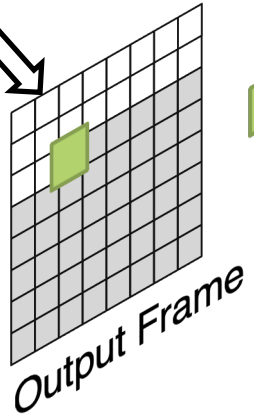


Shift registers

⊗ *Convolution (conv)*

Filter weights





Output pixel produced by conv

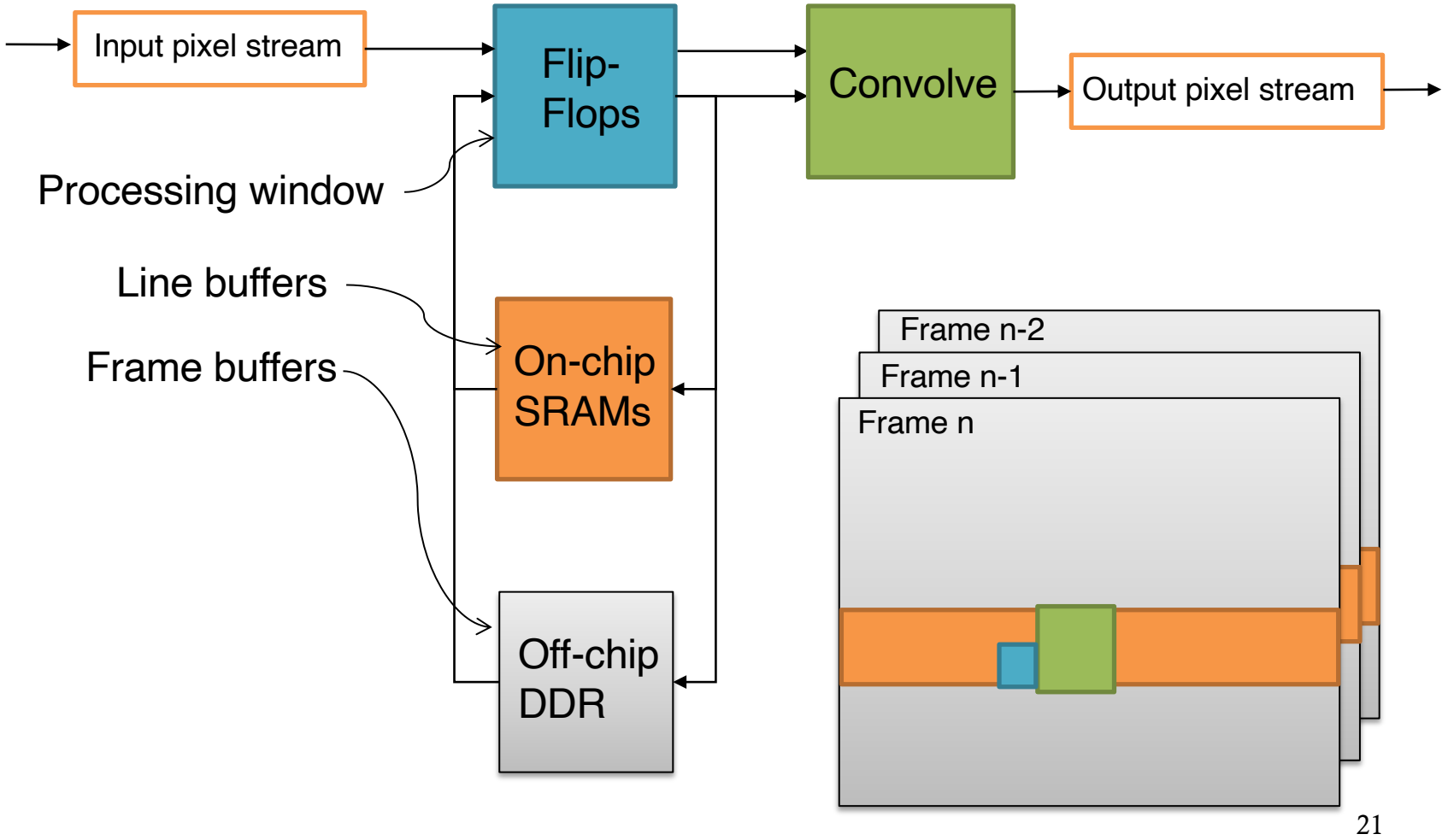
New pixel read from frame buffer in main memory (DRAM)

2. **Update line buffer** by removing the oldest pixel and shifting in the new one

Line Buffer + Shift Registers:
a custom “cache” + a custom “register file”

Resulting Specialized Memory Hierarchy

- Memory architecture customized for convolution



HLS Code Snippet

```
1   LineBuffer<2,C,pixel_t> linebuf;
2   Window<3,3,pixel_t> window;
3   for (int r = 1; r < R+1; r++) {
4     for (int c = 1; c < C+1; c++) {
5       #pragma HLS pipeline II=1
6       pixel_t new_pixel = img[r][c];
7       // Update shift window
8       window.shift_left();
9       if (r < R && c < C) {
10        for (int i = 0; i < 2; i++ )
11          window.insert(buf[i][c]);
12      }
13      else { // zero padding
14        for (int i = 0; i < 2; i++)
15          window.insert(0);
16      }
17      window.insert(new_pixel);
18      // Update line buffer
19      linebuf.shift_up(c);
20      if (r < R && c < C)
21        linebuf[1].insert(c, new_pixel);
22      else // Zero padding
23        linebuf[1].insert(c, 0);
24      // Perform 3x3 convolution
25      out[r-1][c-1] = convolve(window, weights);
26    }
27  }
```

Summary

- Pipelining is one of the most commonly-used techniques in HLS to boost the performance
  - Recurrences and resource restrictions limit the pipeline throughput
- Modulo scheduling
  - A regular form of software pipeline technique
    - Also applies to loop pipelining for hardware synthesis
    - NP-hard problem in general
  - SDC-based approach provides an efficient heurisitic

Next Lecture

- Neural network tutorial

Acknowledgements

- These slides contain/adapt materials developed by
  - Prof. Ryan Kastner (UCSD)
  - Prof. Scott Mahlke (UMich)
  - Dr. Stephen Neuendorffer (AMD Xilinx)