ECE 5997
Hardware Accelerator Design & Automation
Fall 2021

Pipelining

Cornell University

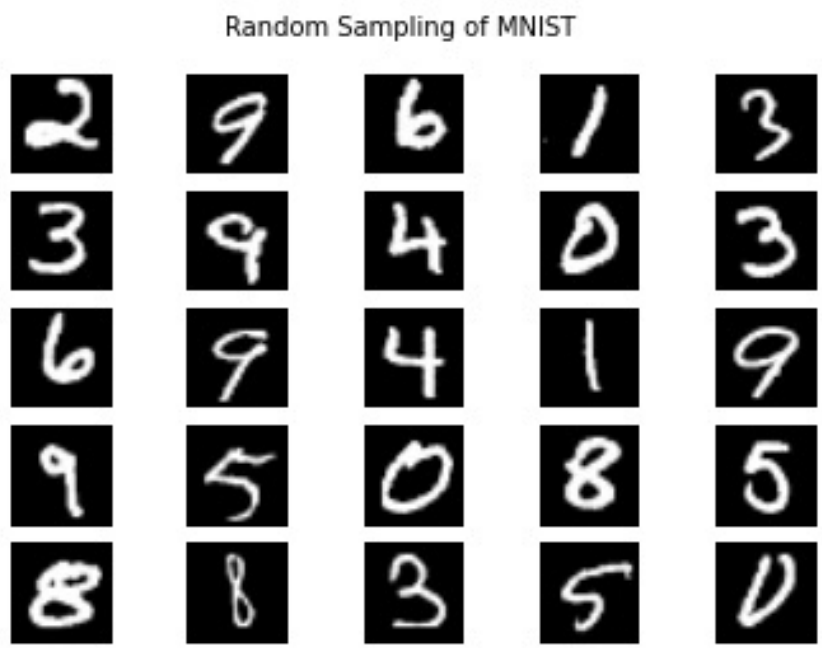
CSL

Announcements

- DNN acceleration tutorial cancelled
  - Slides will be posted on Tuesday

Digit Recognition Lab

- Use a simple machine learning algorithm to recognize handwritten digits
  - 2000 training instances per digit
  - Each training/test instance is a 7x7 bitmap after downsampling



MNIST dataset: http://yann.lecun.com/exdb/mnist/

K-Nearest-Neighbor (KNN) Implementation

```
bit4 digitrec( digit input )
{
  #include "training_data.h"
  // This array stores K minimum distances per training set
  bit6 knn_set[10][K_CONST];
  // Initialize the knn set
  for ( int i = 0; i < 10; ++i )
    for ( int k = 0; k < K_CONST; ++k )
      // Note that the max distance is 49
      knn_set[i][k] = 50;
```

Main compute loop
(10 cycles per innermost loop)

```
L2000: for ( int i = 0; i < TRAINING_SIZE; ++i ) {
  L10:   for ( int j = 0; j < 10; j++ ) {

    // Read a new instance from the training set
    digit training_instance = training_data[j * TRAINING_SIZE + i];
    // Update the KNN set
    update_knn( input, training_instance, knn_set[j] );
  }
}
```

~200K cycles by default without optimizations

10x Speedup through Parallelization

```
bit4 digitrec( digit input )
{
  #include "training_data.h"
  // This array stores K minimum distances per training set
  bit6 knn_set[10][K_CONST];
  // Initialize the knn set
  for ( int i = 0; i < 10; ++i )
    for ( int k = 0; k < K_CONST; ++k )
      // Note that the max distance is 49
      knn_set[i][k] = 50;
```

Unroll inner loop completely

```
  L2000: for ( int i = 0; i < TRAINING_SIZE; ++i ) {
    L10:   for ( int j = 0; j < 10; j++ ) {

      // Read a new instance from the training set
      digit training_instance = training_data[j * TRAINING_SIZE + i];
      // Update the KNN set
      update_knn( input, training_instance, knn_set[j] );
    }
  }
```

Partition training set into 10 banks

~20K cycles after parallelization

Further Speedup through Pipelining

```
bit4 digitrec( digit input )
{
  #include "training_data.h"
  // This array stores K minimum distances per training set
  bit6 knn_set[10][K_CONST];
  // Initialize the knn set
  for ( int i = 0; i < 10; ++i )
    for ( int k = 0; k < K_CONST; ++k )
      // Note that the max distance is 49
      knn_set[i][k] = 50;

  L2000: for ( int i = 0; i < TRAINING_SIZE; ++i ) {
    L10:   for ( int j = 0; j < 10; j++ ) {

      // Read a new instance from the training set
      digit training_instance = training_data[j * TRAINING_SIZE + i];
      // Update the KNN set
      update_knn( input, training_instance, knn_set[j] );
    }
  }
```

Pipeline outer loop

Unroll inner loop completely

Partition training set into 10 banks

~2K cycles after pipelining

Outline

- Restrictions of Pipeline Throughput
  - Types of recurrences
- Modulo scheduling concepts
  - Extending SDC formulation for pipelining
- Case studies on HLS pipelining

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**

Type of Recurrences

- Types of dependences
  - True dependences, anti-dependences, output dependences
  - Intra-iteration vs. inter-iteration dependences

- Recurrence – if one iteration has dependence on the same operation in a previous iteration
  - Direct or indirect
  - Data or control dependence

- Distance – number of *iterations* separating the two dependent operations
  (0 = same iteration or intra-iteration)

True Dependences

- True dependence
  - Aka flow or RAW (Read After Write) dependence
  - S1 $\rightarrow^{t}$ S2
    - Statement S1 precedes statement S2 in the program and computes a value that S2 uses

Example:

```
for (i = 0; i < N; i++)
    A[i] &= A[i-1] - 1;
```

Inter-iteration true dependence
on A (distance = 1)

Anti-Dependences

- Anti-dependence
  - Aka WAR (Write After Read) dependence
  - S1 $\rightarrow^{a}$ S2
    - S1 precedes S2 and may read from a memory location that is later updated by S2
  - Renaming (e.g., SSA) can resolve many of the WAR dependences

Example:

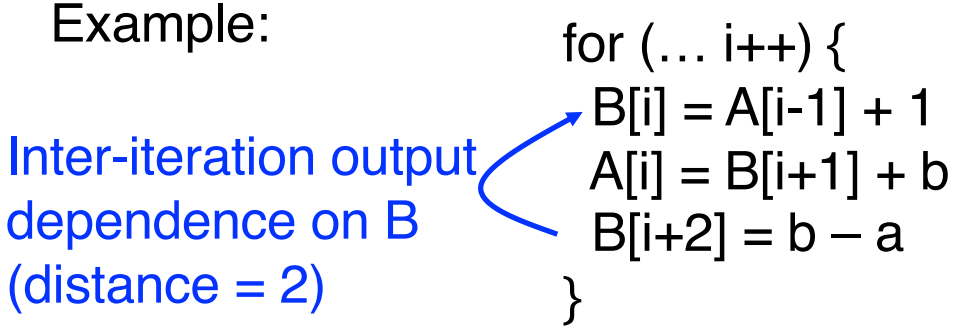
```
for (… i++ ) {
  A[i-1] = b – a;
  B[i] = A[i] + 1
}
```

Inter-iteration anti-dependence on A (distance = 1)

Output Dependences

- Output dependence
  - Aka WAW (Write After Write) dependence
  - S1 precedes S2 and may write to a memory location that is later (over)written by S2
  - Renaming (e.g., SSA) can resolve many of the WAW dependences

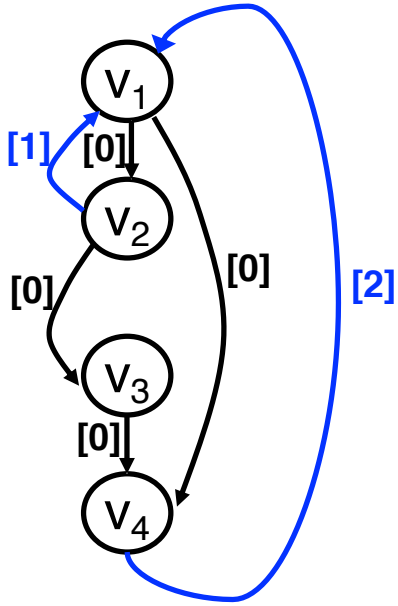
Example:



Dependence Graph

- Data dependences of a loop 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

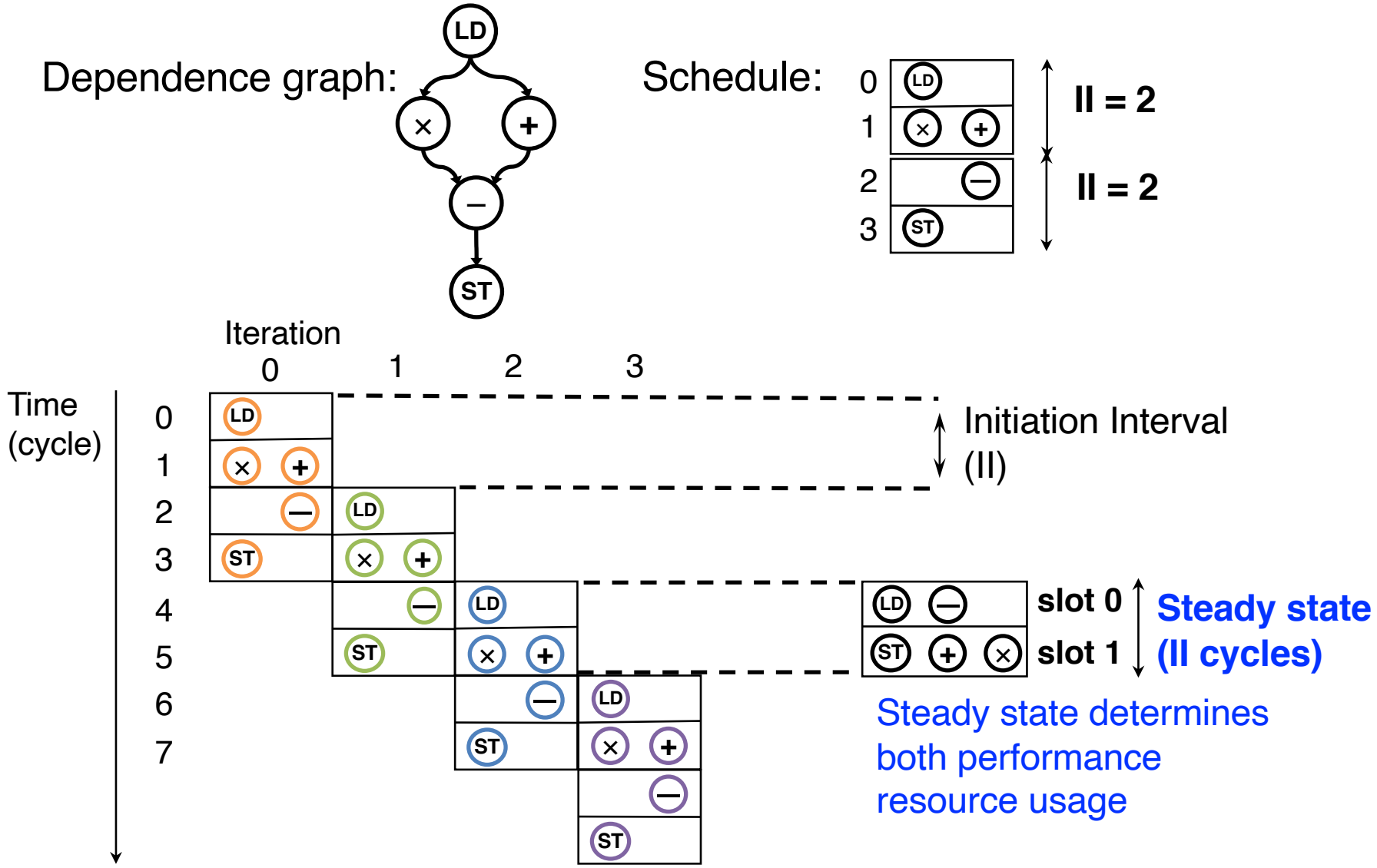
- A regular form of loop (or function) pipelining technique
  - Also applies to software pipelining in compiler optimization
  - **Loop iterations use the same schedule, which are initiated at a constant rate**

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

- Optimization objective
  - minimize II under resource constraints
  - minimize resource usage under II constraint

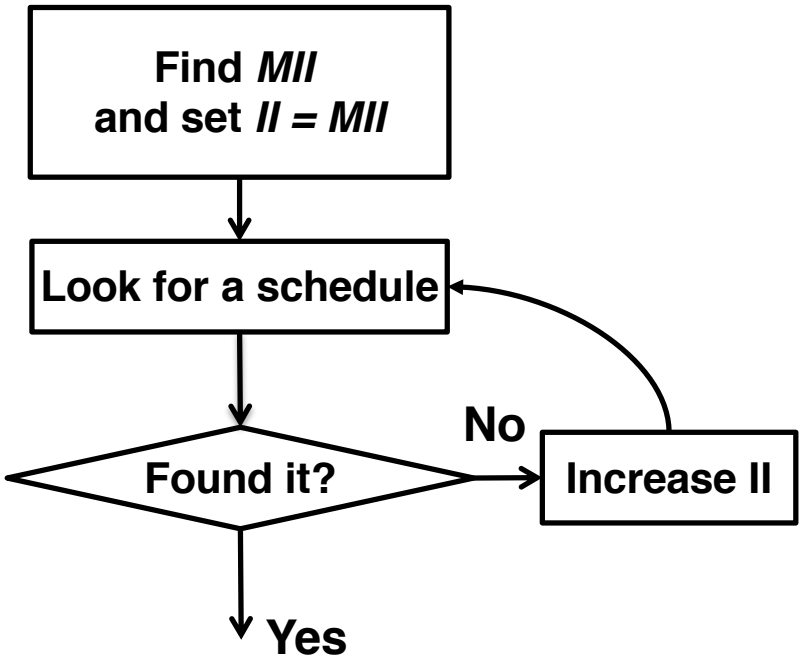
NP-hard in general
Optimal polynomial time solution exists without recurrences or resource constraints

Modulo Scheduling Example



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

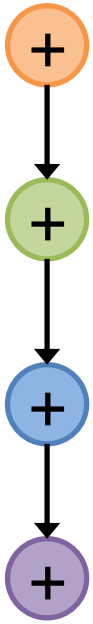
- Minimum possible II (MII)
  - MII = max(ResMII, RecMII)
  - A lower bound, not necessarily achievable

- 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



Reservation tables

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

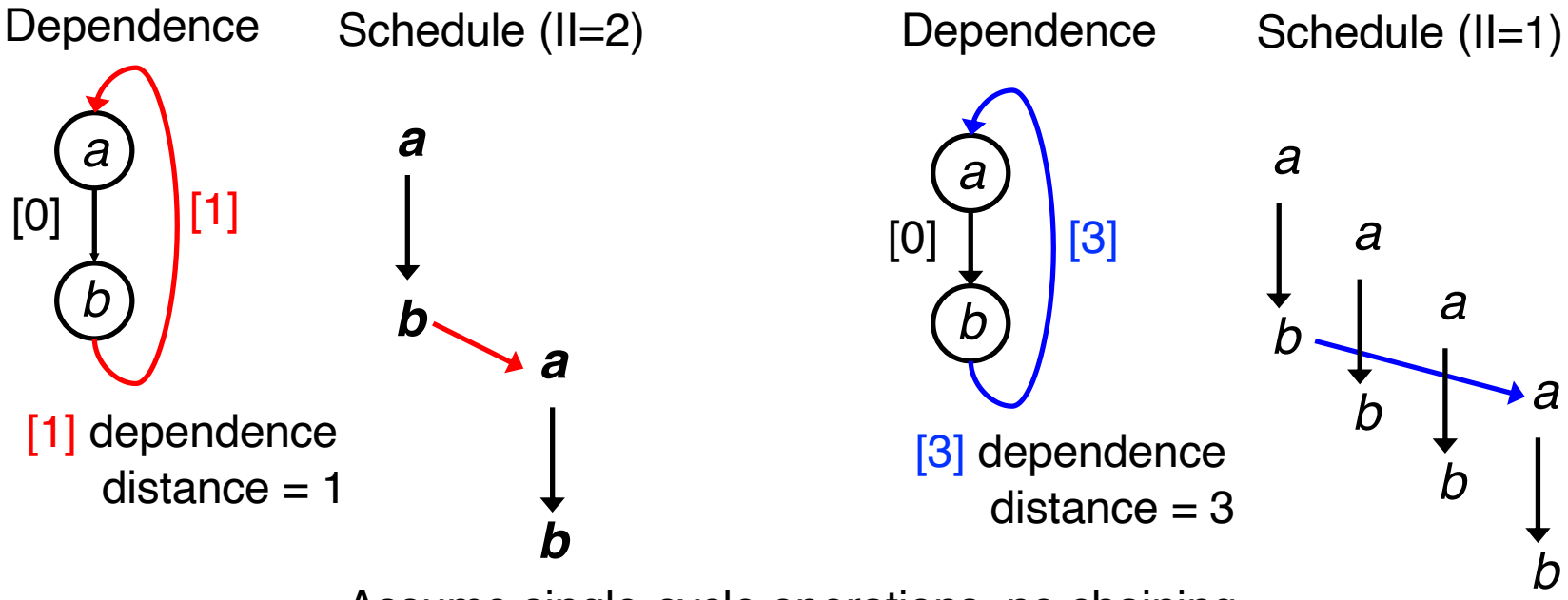
2 adders

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 = $\max_i \lceil OPs(r_i) / Limit(r_i) \rceil$**

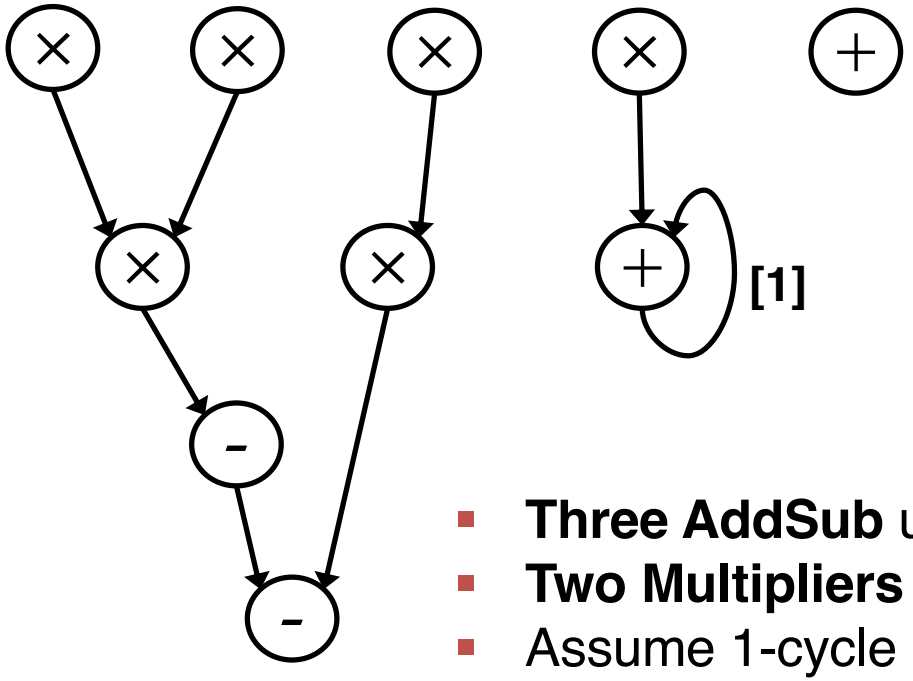
Minimum II due to Recurrences (RecMII)



Assume single-cycle operations, no chaining

- Compute Recurrence Minimum II (*RecMII*):
  - Max among all circuits of:
    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*

Example: Calculating the Minimum II

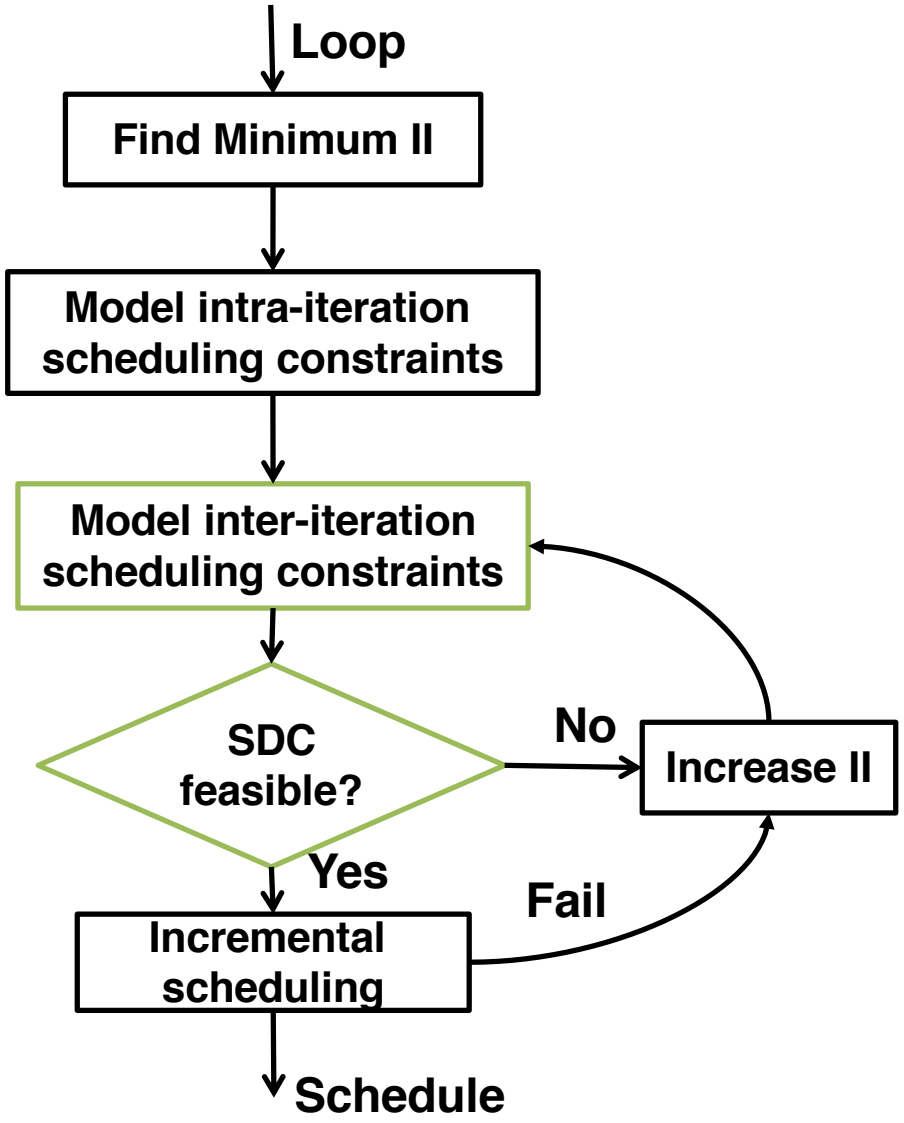


- **Three AddSub** units available
- **Two Multipliers** available
- Assume 1-cycle operations, no chaining

What is the minimum II (MII) ?

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

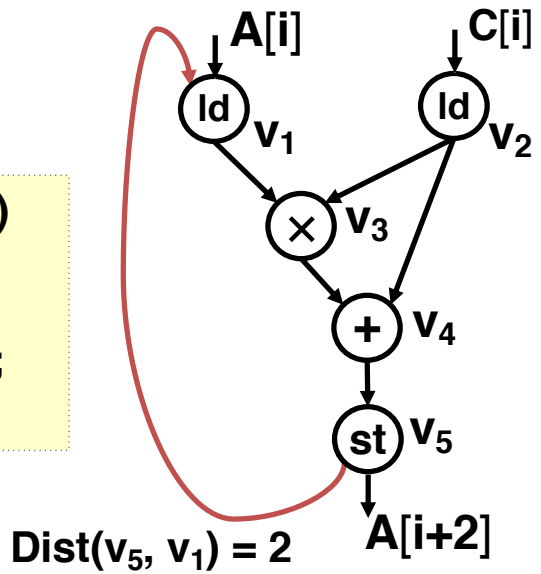


Modeling Loop-Carried Dependence with SDC

- The dependence between two operations from different iterations is termed inter-iteration (loop-carried) dependence
  - Loop-carried dependence $u \rightarrow v$ with $Dist(u, v) = K$

$$s_u + Lat_u \leq s_v + K^* II$$

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

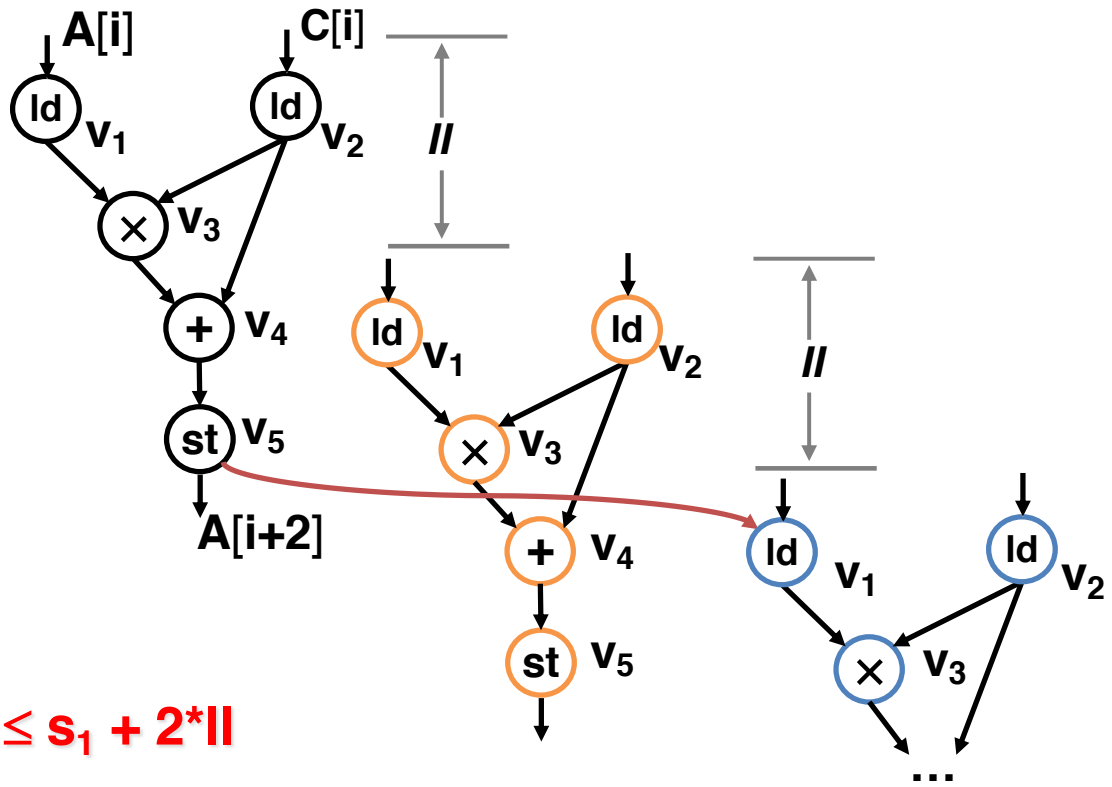


Modeling Loop-Carried Dependence with SDC

- The dependence between two operations from different iterations is termed inter-iteration (loop-carried) dependence
  - Loop-carried dependence $u \rightarrow v$ with $Dist(u, v) = K$

$$s_u + Lat_u \leq s_v + K^* II$$

```
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

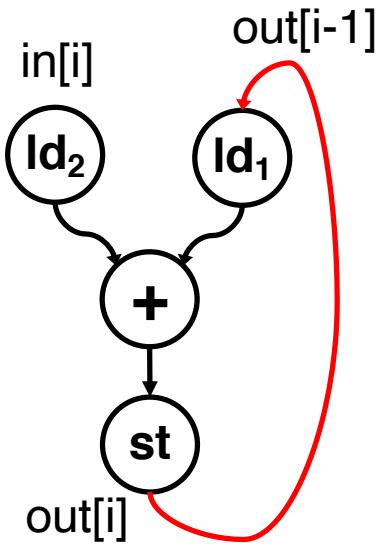
- Prefix sum computes a cumulative sum of a sequence of numbers
  - commonly used in many applications such as radix sort, histogram, etc.

```
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

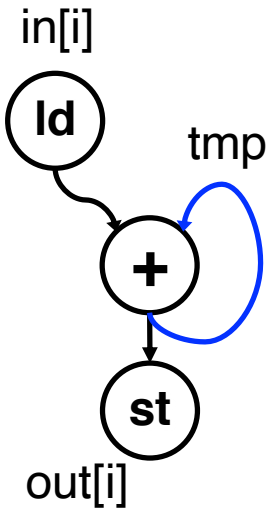
```
out[0] = in[0];
for ( int i = 1; i < N; i++ )
  out[i] = out[i-1]+ in[i];
```

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

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

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

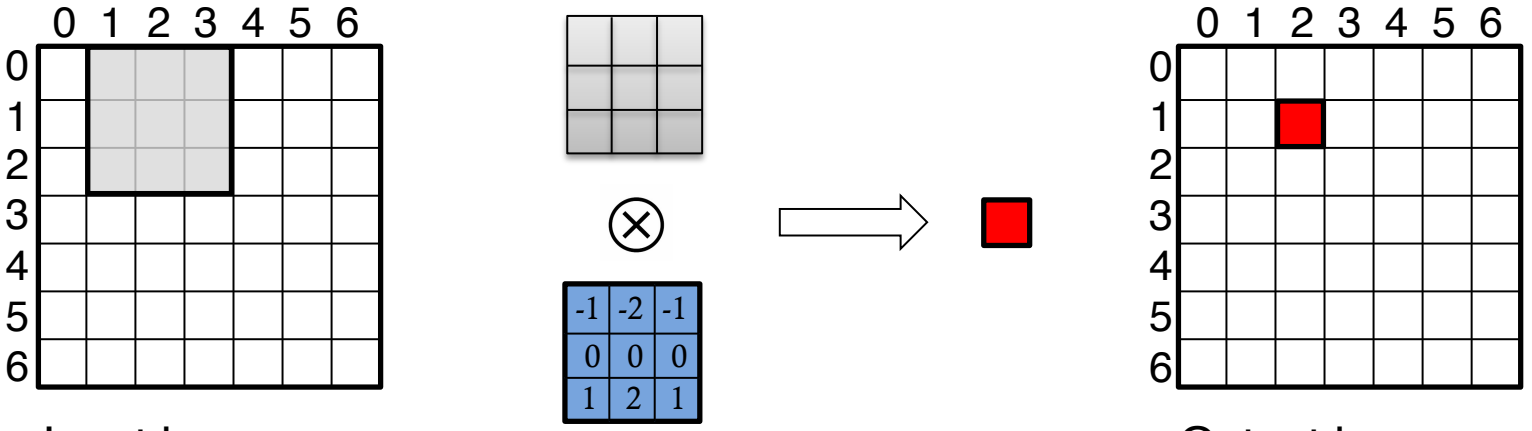
```
int tmp = in[0];
for  ( int i = 1; i < N; i++ )  {
  tmp += in[i];
  out[i] = tmp;
}
```

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

Case Study: Convolution for Image Processing

- A common computation of image/video processing is performed over overlapping stencils, termed as convolution

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



Input image frame

3x3 convolution

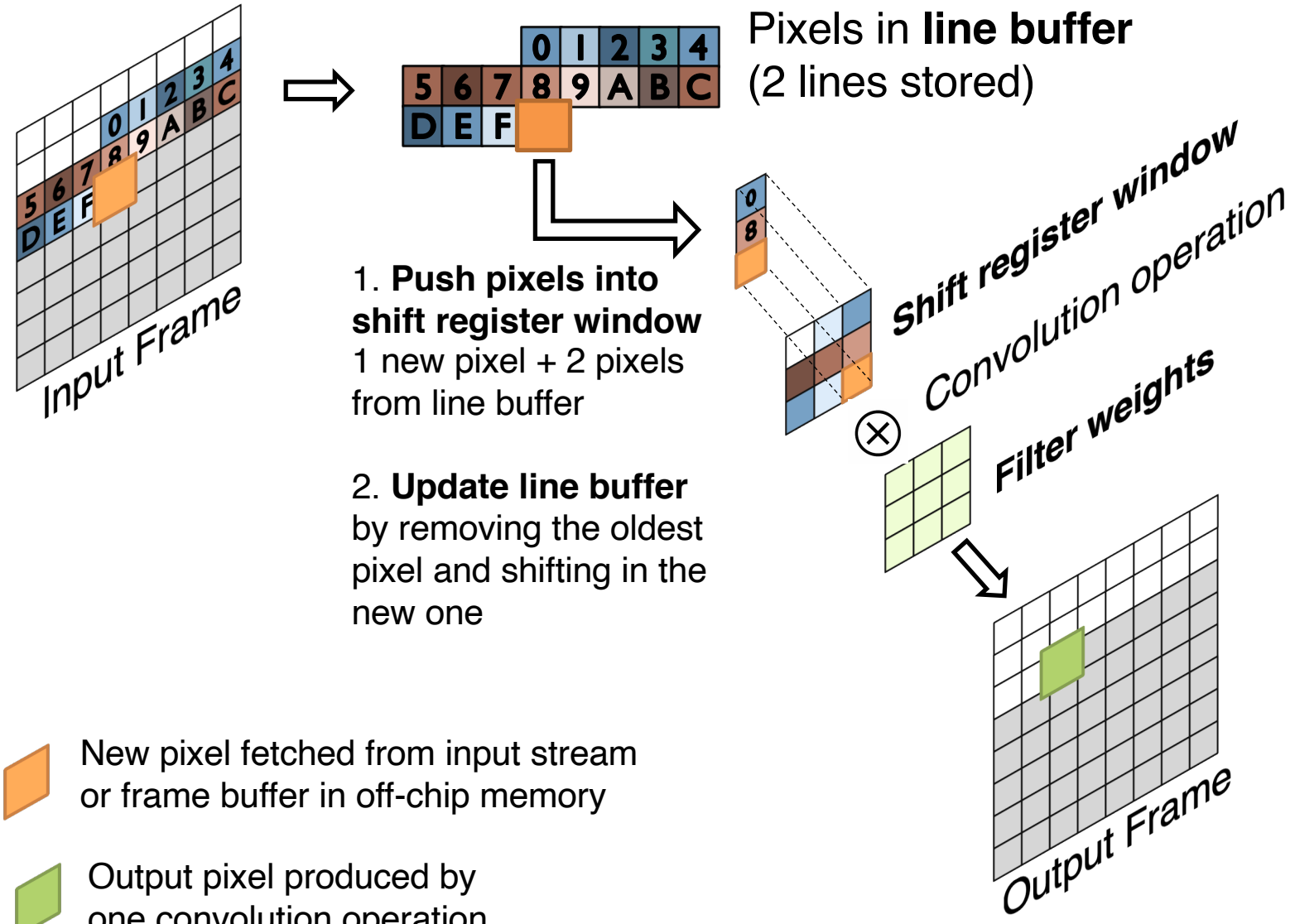
Output image frame

Achieving High Throughput with Pipelining

```
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];
    }
```

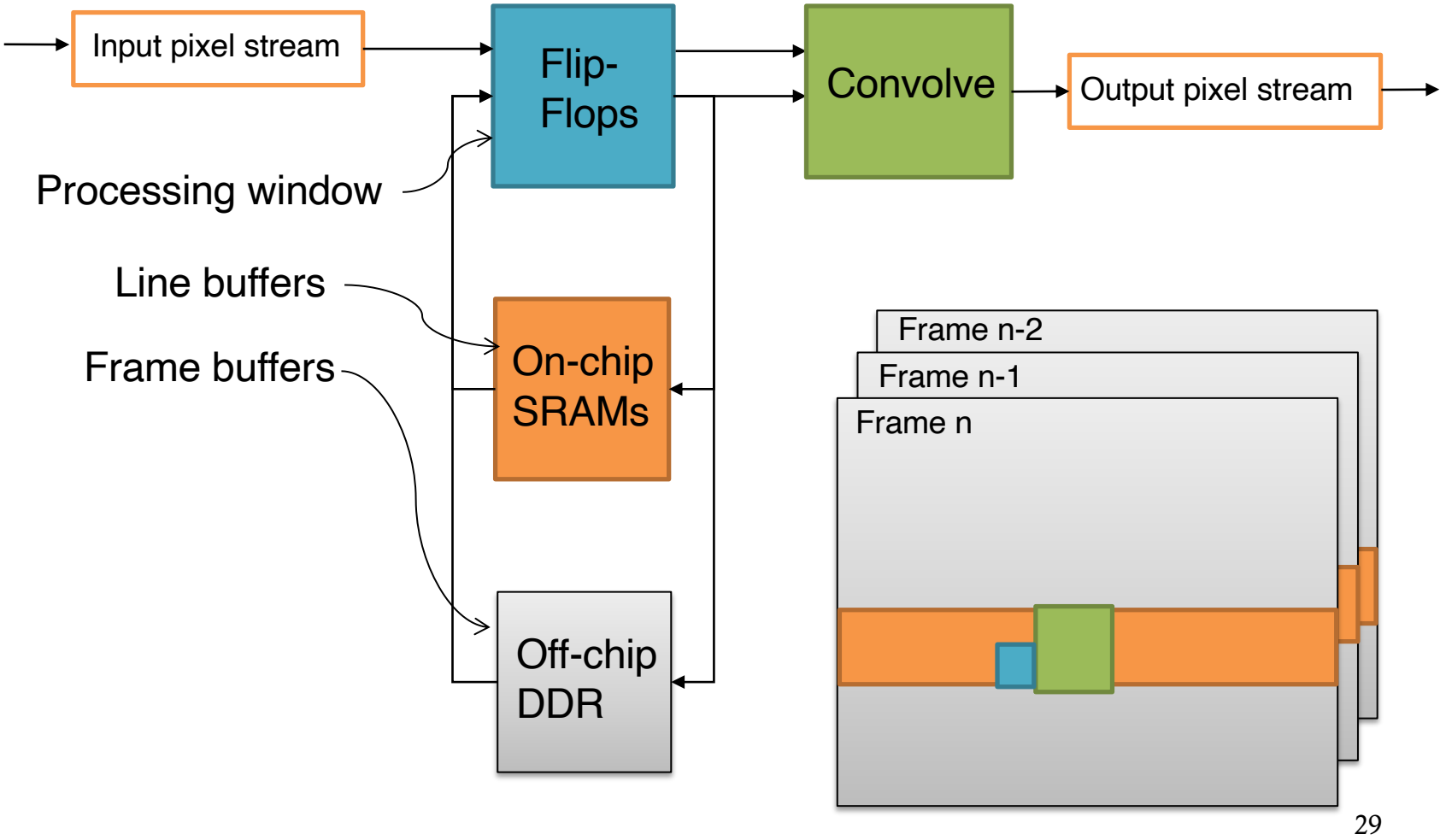
- Inner loops (i & j) are automatically unrolled
- With a 3x3 convolution kernel, 9 pixels are required for calculating the value of one output pixel
- **If** the entire input image is stored in an on-chip buffer with **two read ports**
  - ResMII = ?
  - What about RecMII?

Achieving II=1 for 3x3 Convolution



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 performance of the synthesized hardware
- 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

Acknowledgements

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