ECE 6775
High-Level Digital Design Automation
Fall 2023

More Pipelining

Cornell University

CSL

Announcements

- Lab 3 due tonight (hard deadline)

- HW 2 due Friday, cannot be late by more than **3 days**
  - Solution will be released after the deadline

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

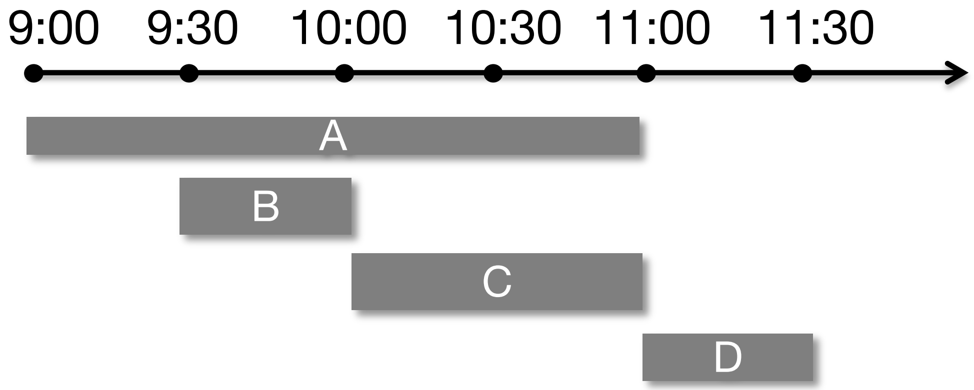
Midterm next Thursday

- Midterm on Thursday 10/19 at **8:30am**
  - In class, 75 mins
  - Open book, open notes, closed Internet

- Topics covered: lectures 01~11 & 13
  - Hardware specialization
  - Algorithm basics
  - FPGA
  - C-based synthesis
  - Control flow graph and SSA
  - Scheduling
  - Resource sharing
  - Pipelining

Review: 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



Gantt chart

Conflict graph
(chromatic number)

Compatibility graph
(clique cover)

Agenda

- Recurrence and type of dependences
- Modulo scheduling concepts
  - Recurrence and resource MII
  - Extending SDC formulation for pipelining
- Case studies

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**

Recurrence and Dependence

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

- Types of **dependences**
  - True dependences, anti-dependences, output dependences
  - Inter-iteration, intra-iteration

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

True Dependences

- True dependence
  - Also known as Read After Write (RAW) or flow dependence
  - $S1 \rightarrow^{t} S2$: S1 precedes S2 in the program execution and computes a value that S2 uses

Example 1

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

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

Example 2

```
for (i = 0; i < N; i++)
    sum += A[i];
```

Inter-iteration true dependence on “sum” (distance = 1)

Anti-Dependences

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

Example

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

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

Output Dependences

- Output dependence
  - Also known as Write After Write (WAW) dependence
  - $S1 \rightarrow^{o} S2$: S1 precedes S2 in the program execution and may write to a memory location that is later (over)written by S2
  - Renaming (e.g., SSA) can resolve many WAW dependences

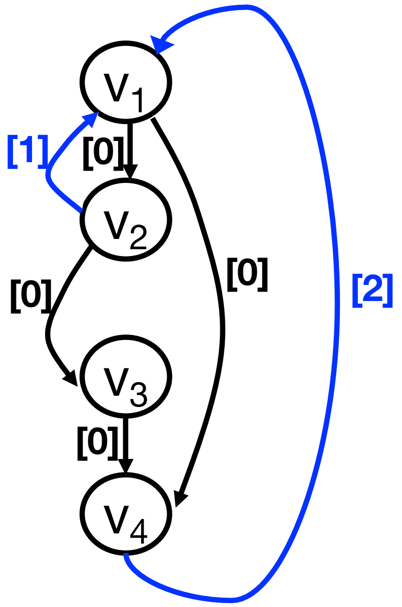
Example

Inter-iteration output dependence on "B" (distance = 2)

```
for (i = 0; i < N-2; i++) {
 B[i] = A[i-1] + 1
 A[i] = B[i+1] + b
 B[i+2] = b – a
}
```

Dependence Graph

- Data dependences of a loop are often represented by a dependence graph
  - Forward edges: **Intra-iteration** (or loop-independent) dependences
  - Back edges: **Inter-iteration** (or loop-carried) dependences
  - Edges are annotated with **distance** values: number of iterations separating the two dependent operations involved
- Recurrence manifests itself as a **cycle** 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 initiation interval (II) under resource constraints

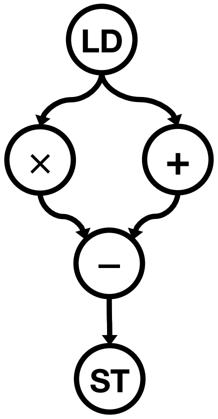
- Advantages of modulo scheduling
  - Cost efficient: No code or hardware replication
  - Easy to analyze: **Steady state determines II & resource**

- **NP-hard in general:** optimal polynomial time solution only exists without recurrences or resource constraints

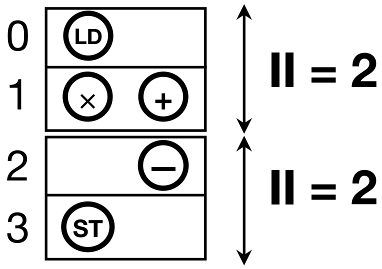
Modulo Scheduling Example

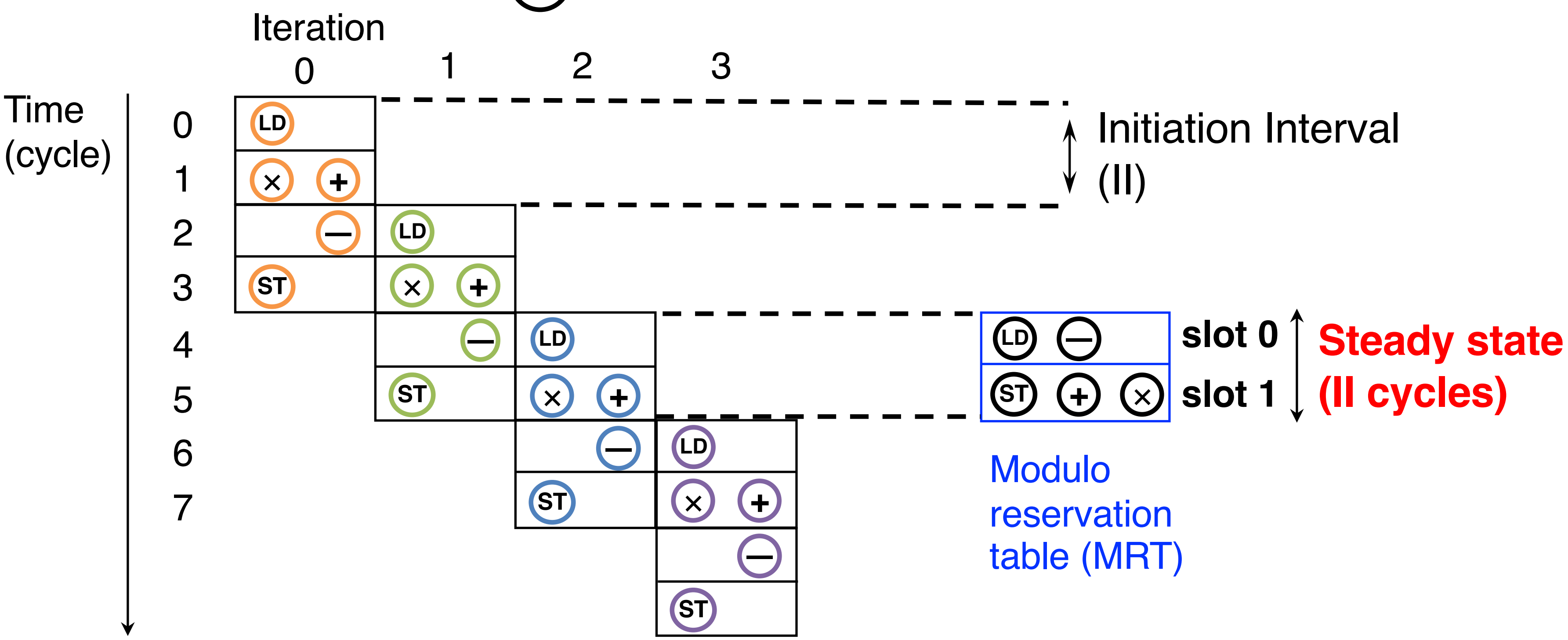
Dependence graph of a loop body

LD: Load
ST: Store



Schedule of the body





Initiation Interval (II)

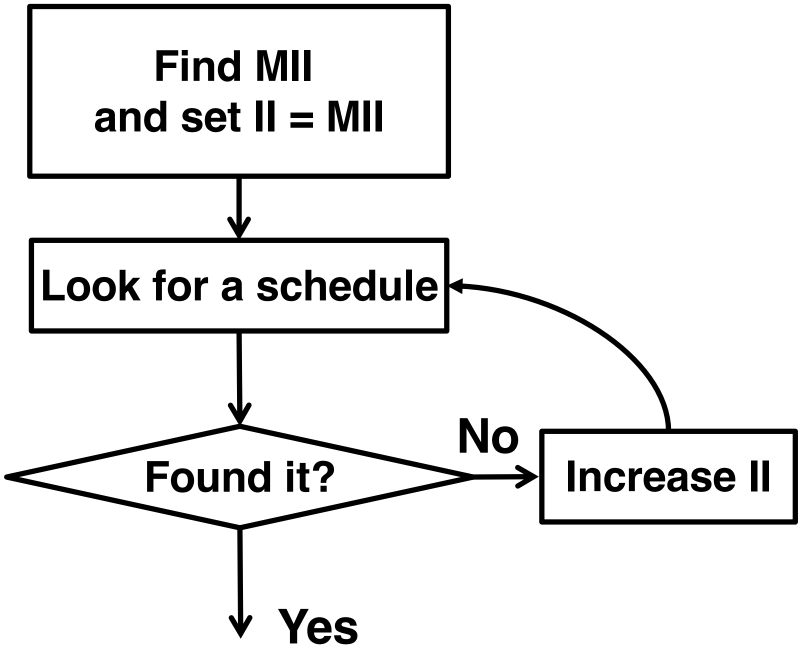
Steady state (II cycles)

Modulo reservation table (MRT)

Steady state determines both performance and resource usage

Heuristics for Modulo Scheduling

- A common, iterative 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



Find MII and set II = MII

Look for a schedule

Found it?

No

Increase II

Yes

Calculating Lower Bound of Initiation Interval

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

Minimum II due to Resource Limits (ResMII)

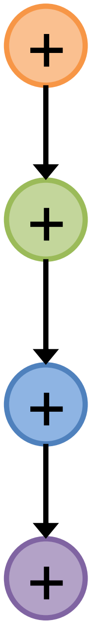
- Compute ResMII*:* Max among all types of resources

$$\textbf{ResMII} = \max_i \lceil \mathrm{OPs}(r_i) / \mathrm{Limit}(r_i) \rceil$$

OPs(r): # of operations that use resource r
Limit(r): # of available resources of type r

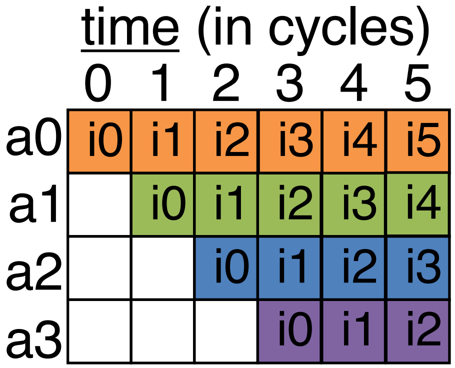
Take the max ratio among all resource types

Dependence



Resource Allocation & Binding

4 adders
(a0~a3)

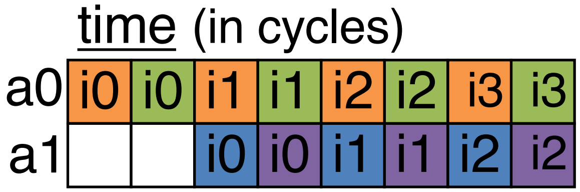


time (in cycles)

	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, 4, 5 : time (cycles)
a0, a1, a2, a3 : available adders
i0, i1, i2, … : loop iterations

2 adders
(a0,a1)



time (in cycles)

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

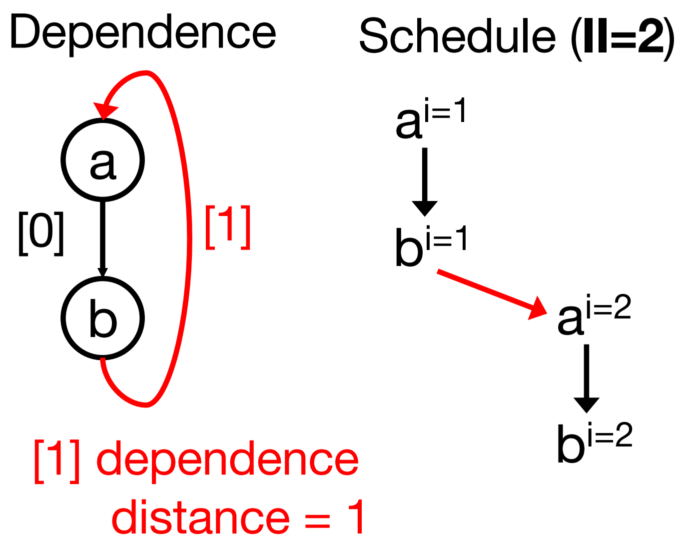
Minimum II due to Recurrences (RecMII)

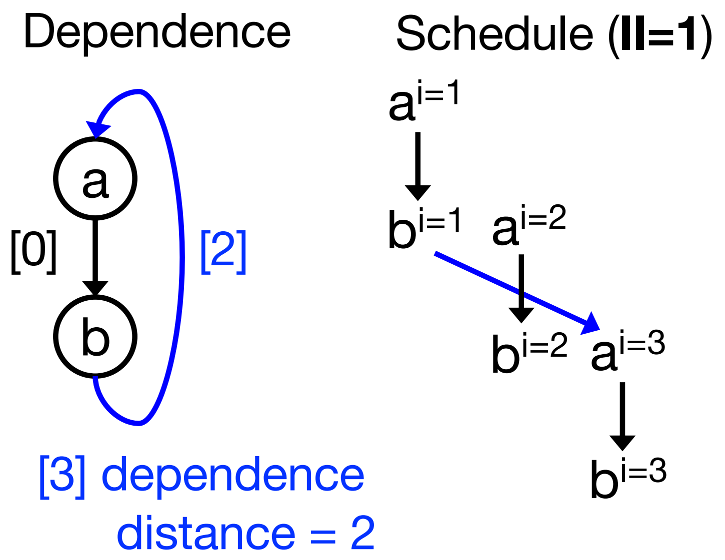
- Compute recurrence MII (RecMII)

Take the max ratio among all dependence cycles

$$\textbf{RecMII} = \max_i \left\lceil \text{Latency}(c_i) \,/\, \text{Distance}(c_i) \right\rceil$$

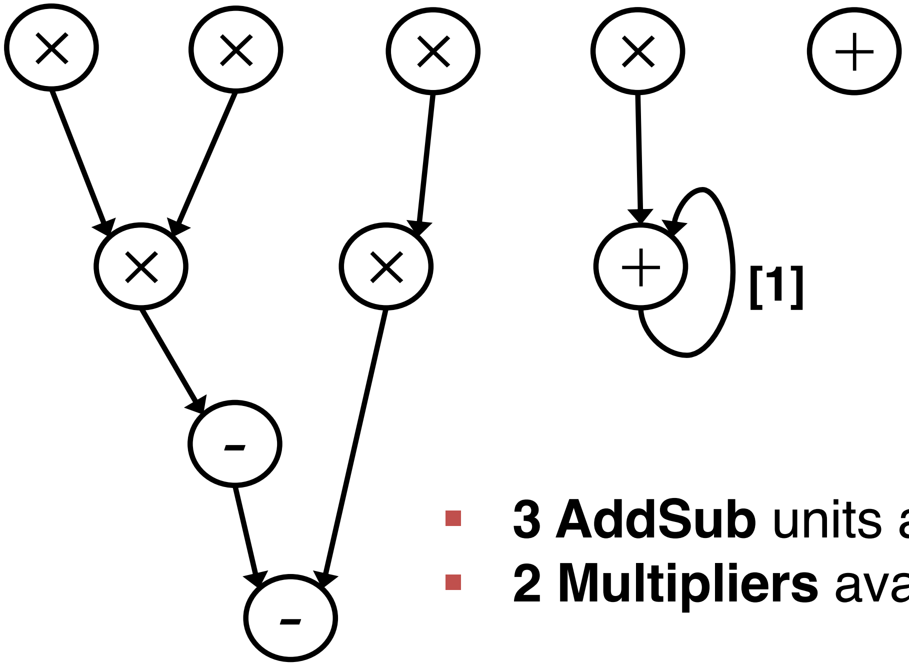
Latency(c): sum of operation latencies along cycle *c*
Distance(c): sum of dependence distances along cycle *c*





above example assumes single-cycle operations and no chaining

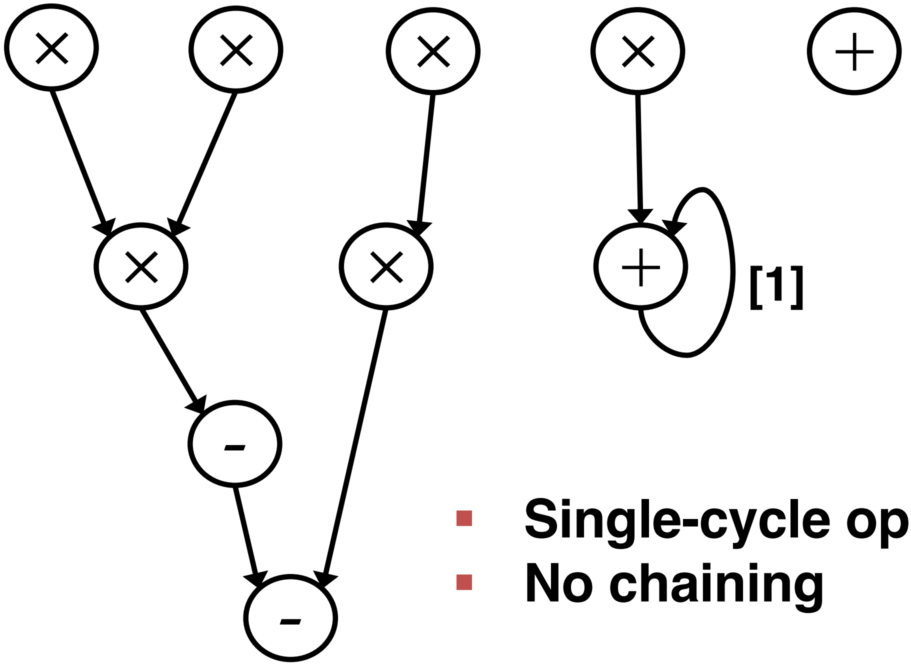
What's the ResMII



- **3 AddSub** units available
- **2 Multipliers** available

Analyze the MII for pipelining the above DFG

What's the RecMII

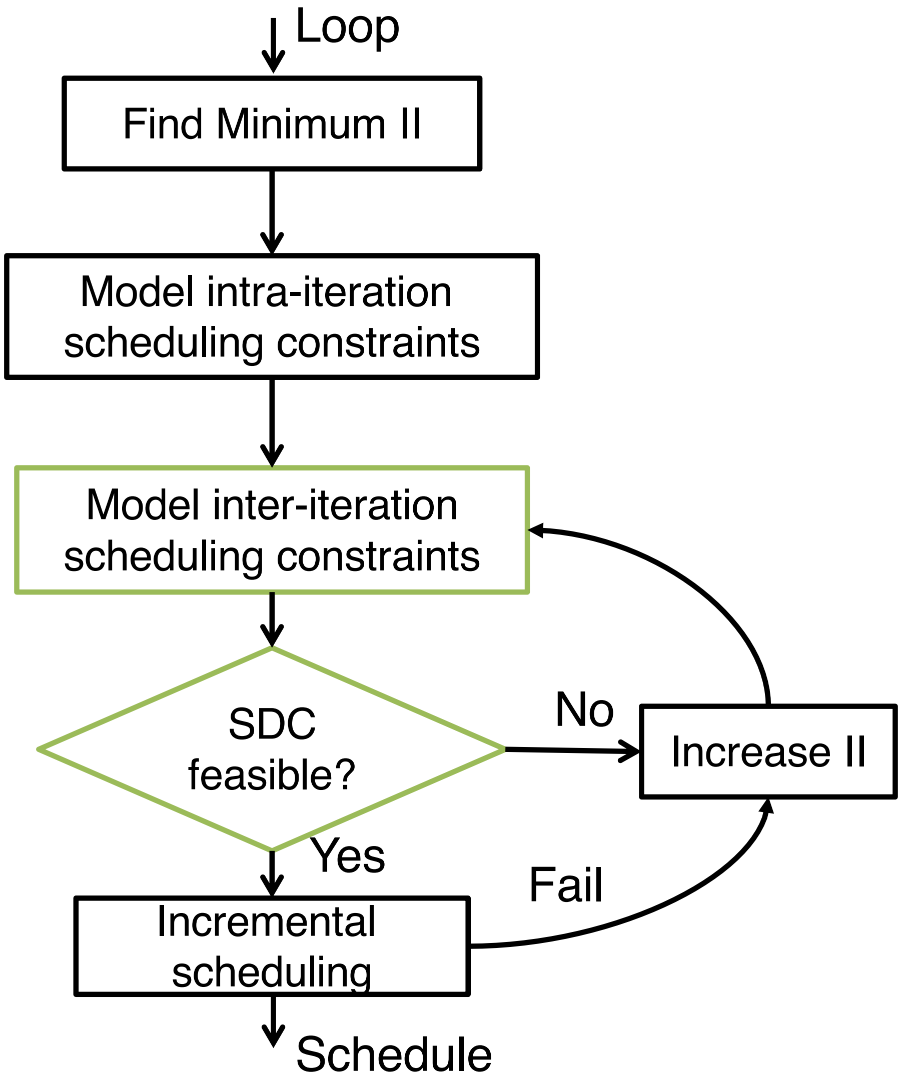


- **Single-cycle operations**
- **No chaining**

Analyze the MII for pipelining the above DFG

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

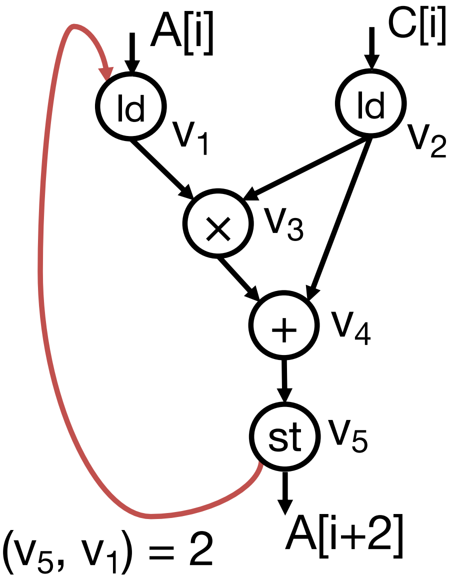


[Z. Zhang & B. Liu, ICCAD 2013]

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];
}
```

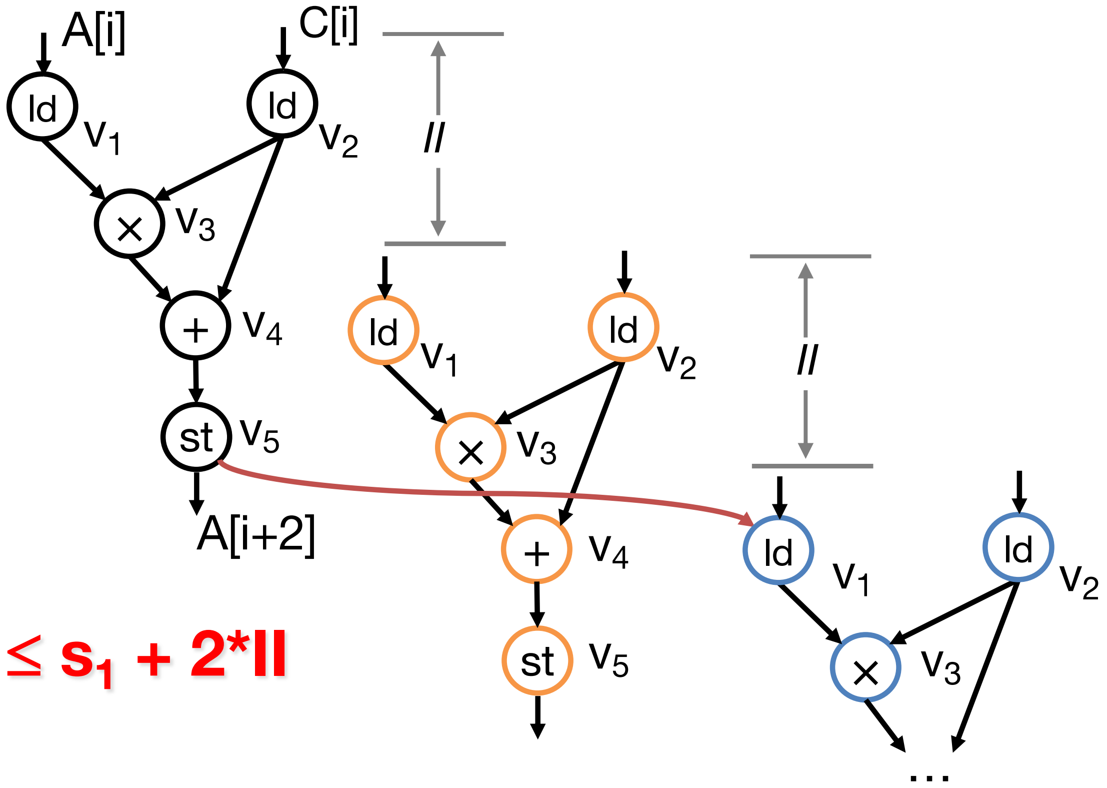


K = Dist(v_5, v_1) = 2

Modeling Loop-Carried Dependence with SDC

- Loop-carried dependence u → v with Distance(u, v) = K
  $s_u + \text{Latency}_u \leq s_v + K*II$

```
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];

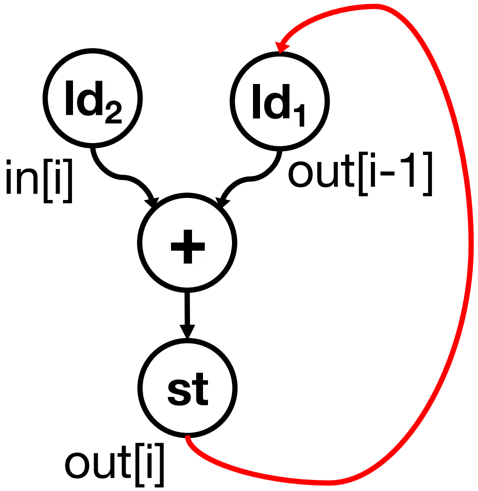
out[1] = in[0] + in[1] + in[2] + in[3];

…

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

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

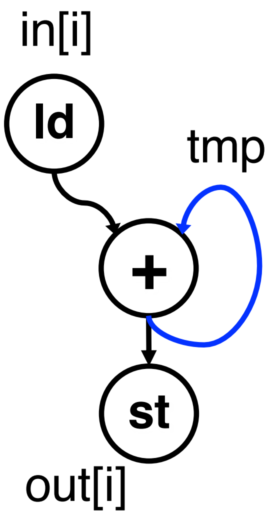


ld – Load
st – Store

	cycle 1	cycle 2	cycle 3	cycle 4
i = 0	ld_1 ld_2	+	**st**	
i = 1	~~***II* = 1**~~	$\mathbf{ld_1}$ 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		ld	+	st

***II* = 1**

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 heuristic

Acknowledgements

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