ECE 6775
High-Level Digital Design Automation
Fall 2024

Pipelining

Cornell University

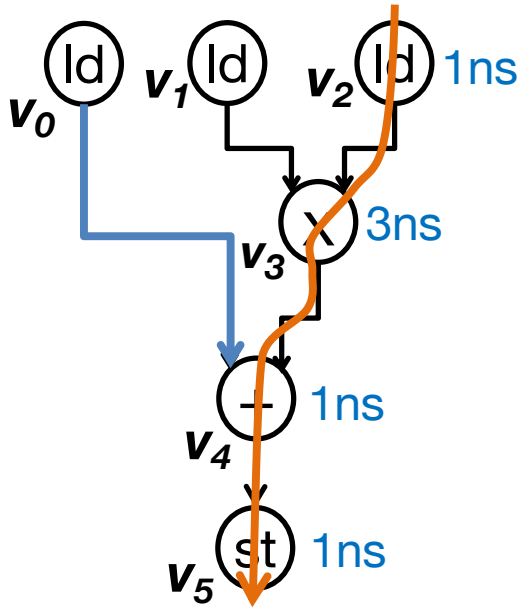
CSL

Announcements

- HW 2 posted, due Friday
  - NO penalty for late submissions, *up to 5 days late* (until Wed Oct 16); Solution will be released on Thursday Oct 17th.

Review: SDC-Based Scheduling

- A linear programming formulation based on system of integer difference constraints (SDC)



- Target cycle time: 5ns
- Delay estimates
  - Mul (x): 3ns
  - Add (+): 1ns
  - Load/Store (ld/st): 1ns

s_i : schedule variable for operation *i*

- Dependence constraints
  - $<v_0 , v_4> : s_0 - s_4 \le 0$
  - $<v_1 , v_3> : s_1 - s_3 \le 0$
  - $<v_2 , v_3> : s_2 - s_3 \le 0$
  - $<v_3 , v_4> : s_3 - s_4 \le 0$
  - $<v_4 , v_5> : s_4 - s_5 \le 0$

Operation chaining is naturally supported

- Cycle time constraints
  - $v_1 \rightarrow v_5 : s_1 - s_5 \le -1$
  - $v_2 \rightarrow v_5 : s_2 - s_5 \le -1$

Timing constraints

To meet the cycle time, v_2 and v_5 should have a minimum separation of one cycle

[J. Cong & Z. Zhang, DAC, 2006] [Z. Zhang & B. Liu, ICCAD, 2013]

Agenda

- Introduction to pipelined scheduling
  - Parallel processing vs. Pipelining
  - Common forms in hardware accelerators
  - Throughput restrictions: resources and recurrences
- Modulo scheduling concepts
  - Recurrence and resource MII
  - Extending SDC formulation for pipelining

Parallelization Techniques

- Parallel processing
  - Emphasizes concurrency by *replicating* a hardware structure several times (typically 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 (typically 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

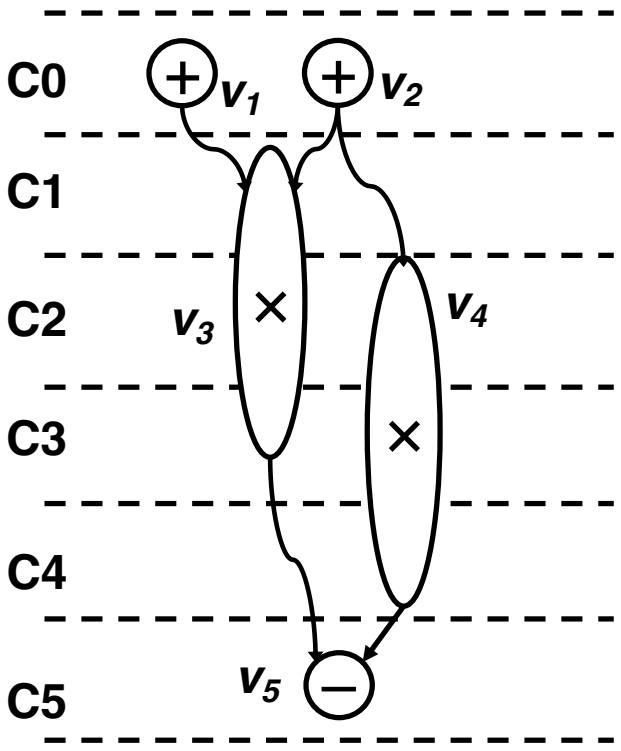
- Operator pipelining
  - Fine-grained pipeline (e.g., functional units, memories)
  - Execute a sequence of operations on a pipelined resource

- 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)

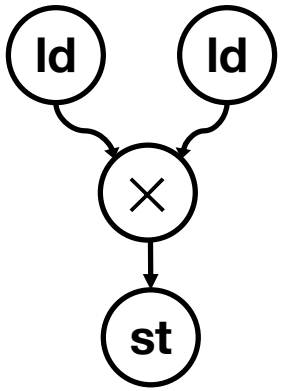


Loop Pipelining

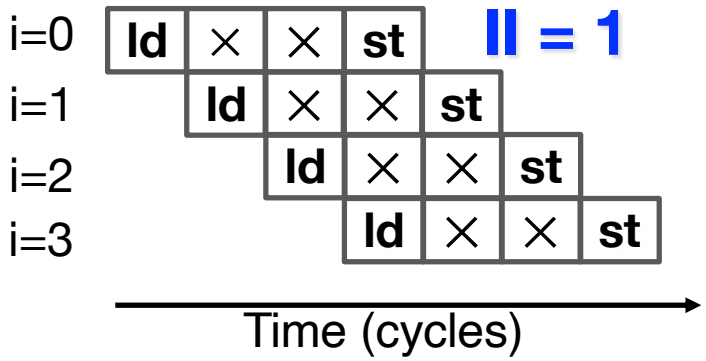
- Pipelining is one of the most important optimizations for HLS
  - Key factor: **Initiation Interval (II)**
  - Allows a new iteration to begin processing, II cycles after the start of the previous iteration **(II=1 means the loop is fully pipelined)**

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

Dataflow of loop body



Pipelined schedule



ld – Load (memory read)
st – Store (memory write)

Here we assume multiplication (×) takes two cycles

Exercise: Loop Pipeline Performance

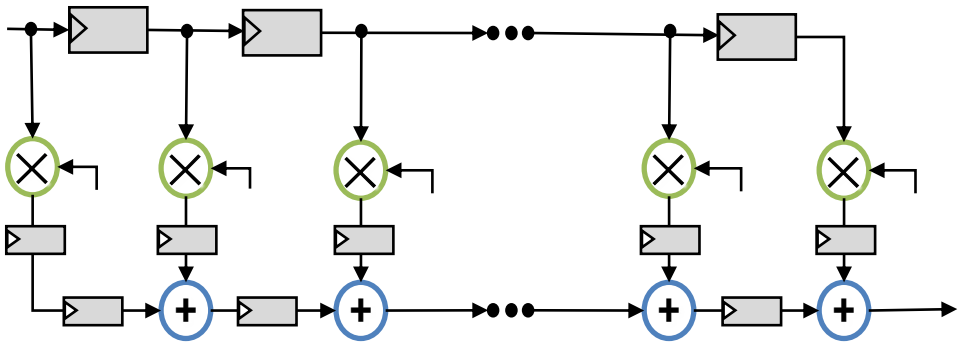
- Given a 100-iteration loop, where its loop body takes 50 cycles to execute
  - With II = 1, how many cycles is needed to complete execution of the entire loop?
  - What about II = 2?

Function Pipelining

- Function pipelining: Entire function is becomes a pipelined datapath

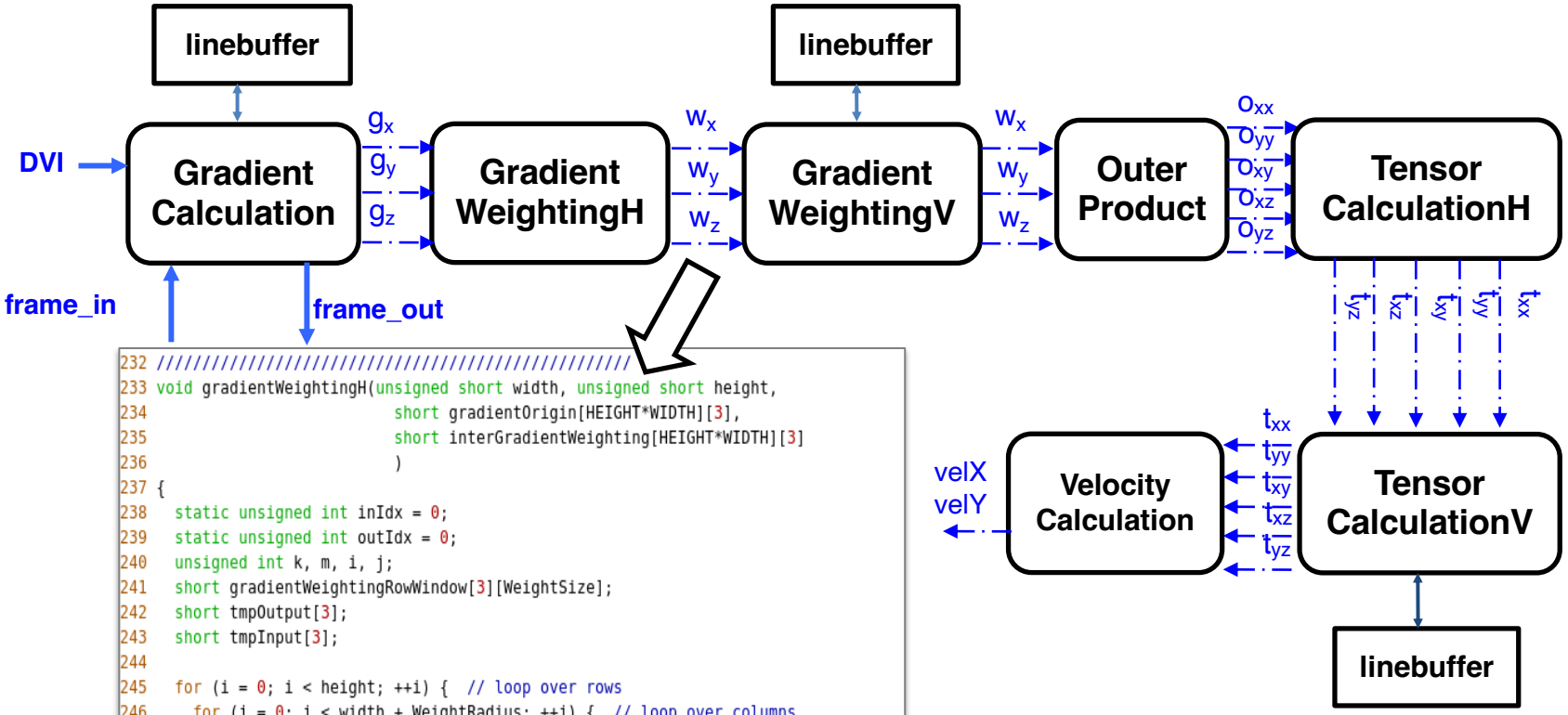
```
void fir(int *x, int *y)
{
    static int shift_reg[NUM_TAPS];
    const int taps[NUM_TAPS] =
       {1, 9, 14, 19, 26, 19, 14, 9, 1};
    int acc = 0;
    for (int i = 0; i < NUM_TAPS; ++i)
       acc += taps[i] * shift_reg[i];
    for (int i = NUM_TAPS - 1; i > 0; --i)
       shift_reg[i] = shift_reg[i-1];

    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
(called *dataflow* pipeline in the HLS tool)

Restrictions of Pipeline Throughput

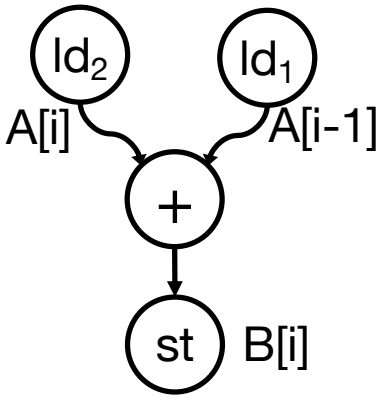
- Resource limitations
  - Limited compute resources
  - Limited memory resources (esp. memory port limitations)
  - Restricted I/O bandwidth
  - Low throughput of subcomponent

  …

- 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	~~**//**~~	**ld_1**	ld_2	+

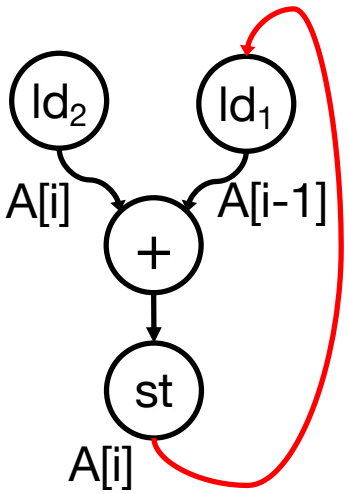
Port conflict

Assuming arrays A and B are held in two different SRAMs

Only one read port per SRAM → 1 load / cycle

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 operation chaining is not allowed here due to cycle time constraint

More on Recurrences

- **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
  - Intra-iteration, also known as loop-independent dependences (Lec 09)
  - **Inter-iteration,** also known as **loop-carried** dependences (focus of this lecture)

- Dependence **distance** – *number of iterations* separating the two dependent operations
  - Intra-iteration, distance = 0 (same iteration)
  - Inter-iteration, distance > 0

True Dependences

- True dependence, also known as Read After Write (RAW) or flow dependence

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 and Output Dependences

- Anti-dependence, also known as Write After Read (WAR) dependence

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 dependence: also known as Write After Write (WAW) dependence

Example

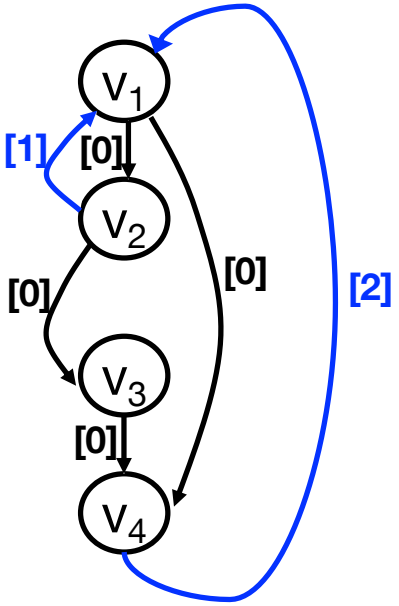
```
for (i = 0; i < N-2; i++) {
  B[i] = A[i-1] + 1
  A[i] = B[i+1] + b
  B[i+2] = b – a
}
```

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

Dependence Graph

- Data dependences of a loop are often represented by a dependence graph
  - Forward edges: **Intra-iteration**
  - Back edges: **Inter-iteration**
  - 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

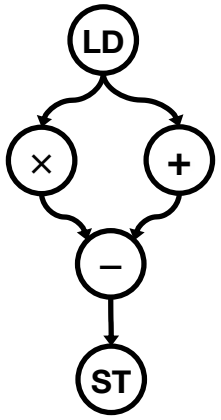
- Advantages of modulo scheduling
  - Cost efficient: No code or hardware replication
  - Easy to analyze: **Steady state determines II & resource**

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

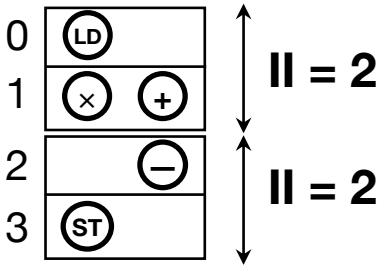
Modulo Scheduling Example

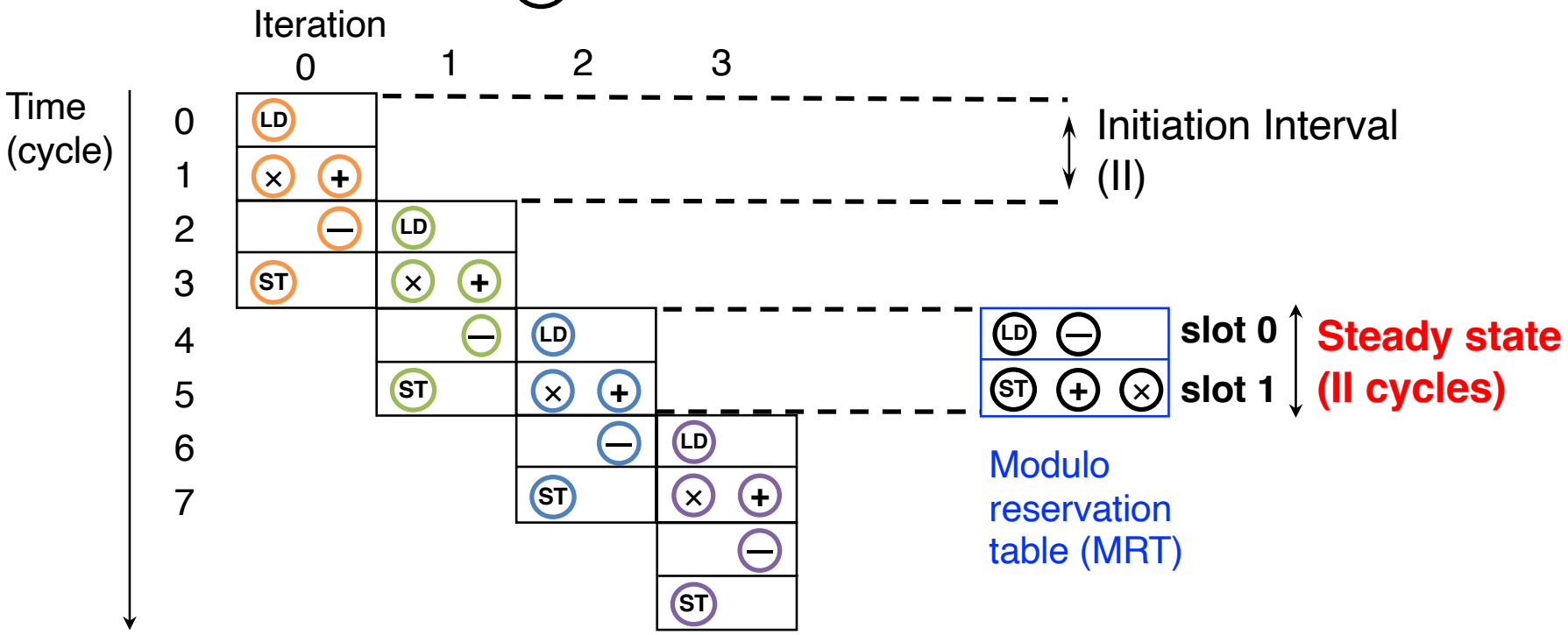
Dependence graph of a **loop body**

LD: Load
ST: Store



Schedule of the loop body

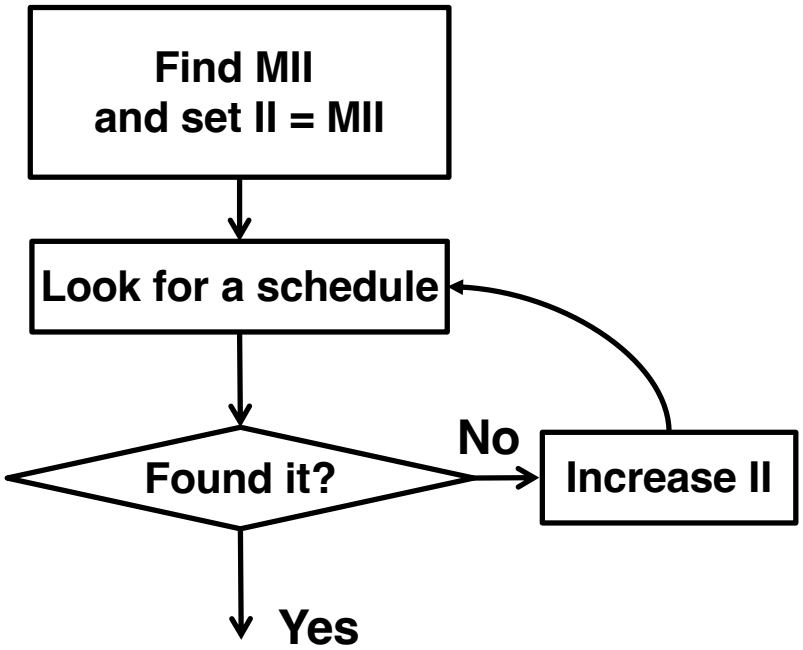




Steady state determines both performance and resource usage

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



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 cycle c_i
    Distance(c_i): total distance in dependence cycle c_i

Minimum II due to Resource Limits (ResMII)

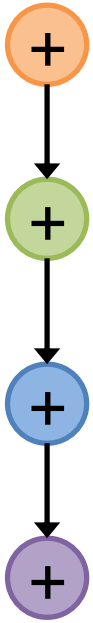
- Compute ResMII: Max among all types of resources

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

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)

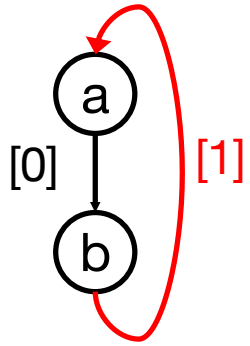
- Compute recurrence MII (RecMII)

Take the max ratio among all dependence cycles

RecMII = $\max_i$ $\lceil$ Latency(c_i) / Distance(c_i) $\rceil$

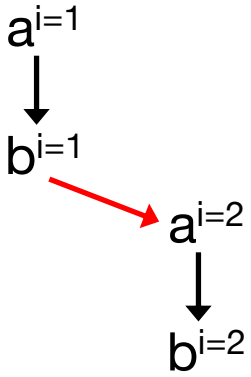
Latency(c): sum of operation latencies along cycle *c*
Distance(c): sum of dependence distances along cycle *c*

Dependence



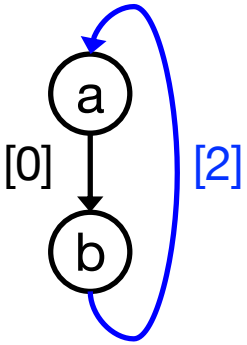
[1] dependence distance = 1

RecMII = 2/1 = 2



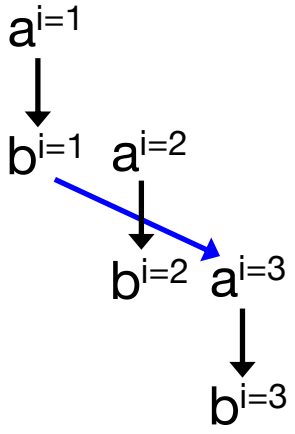
an II=2 schedule

Dependence



[2] dependence distance = 2

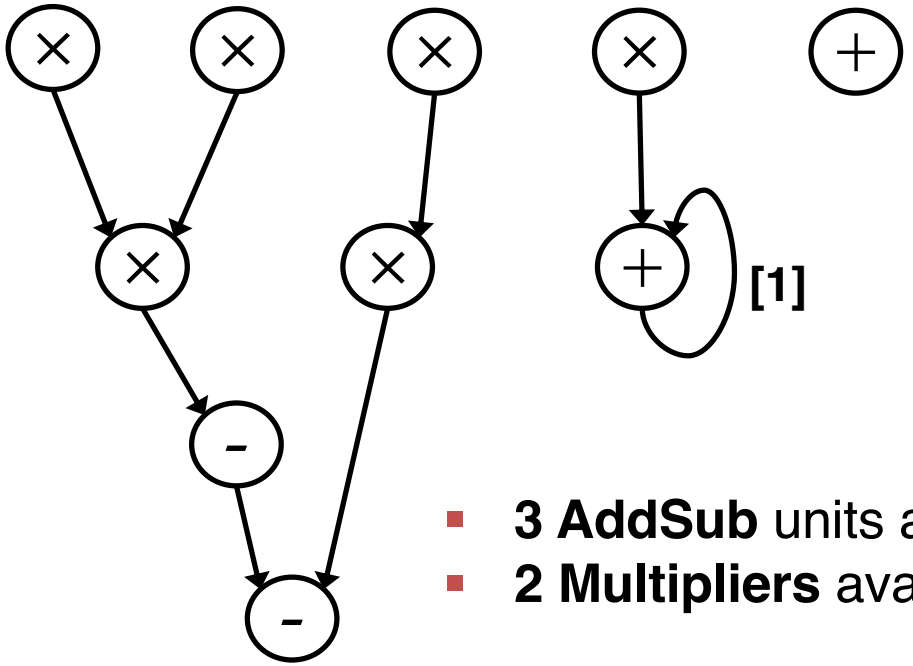
RecMII = 2/2 = 1



an II=1 schedule

above examples assume single-cycle operations and no chaining

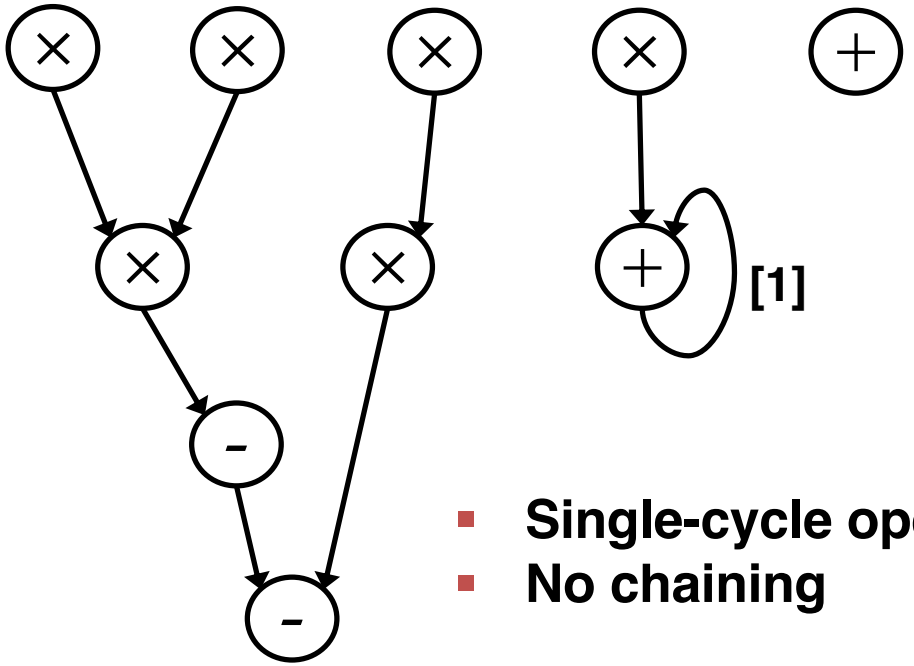
What's the ResMII



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

Analyze the ResMII for pipelining the above DFG

What's the RecMII

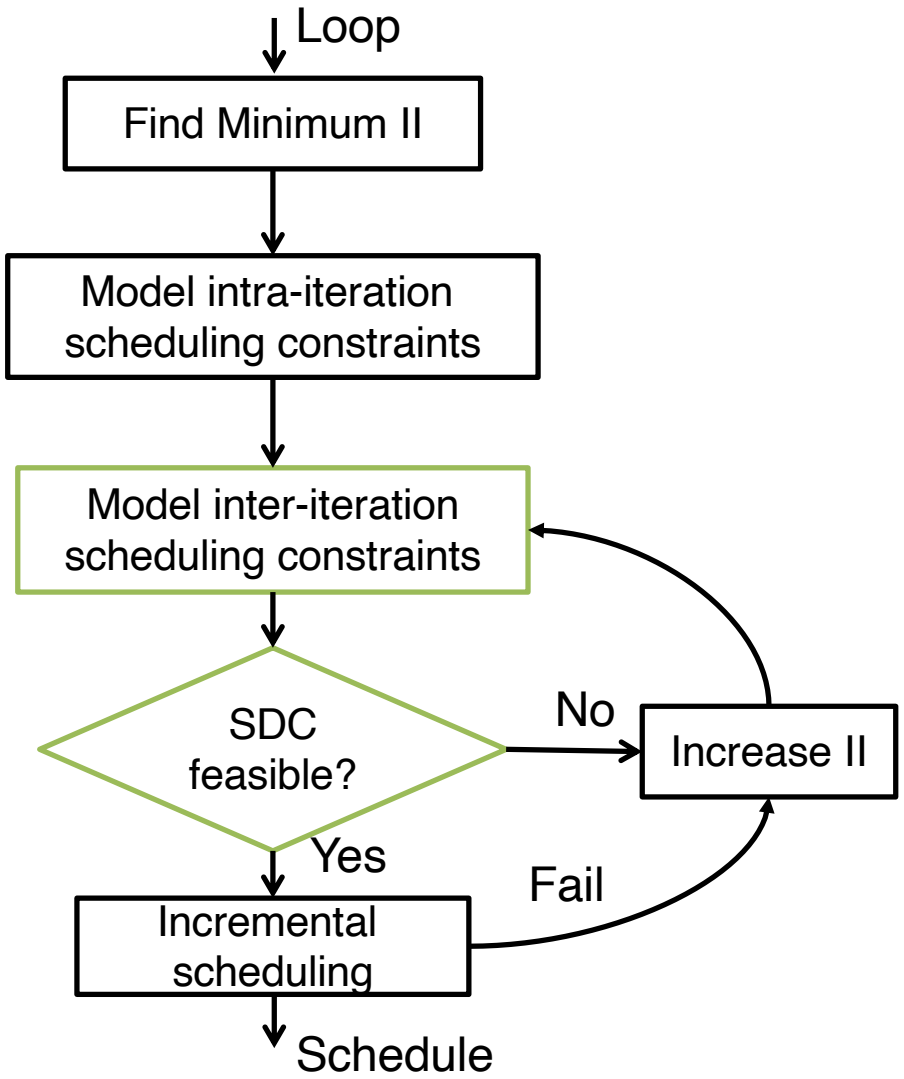


- **Single-cycle operations**
- **No chaining**

Analyze the RecMII 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

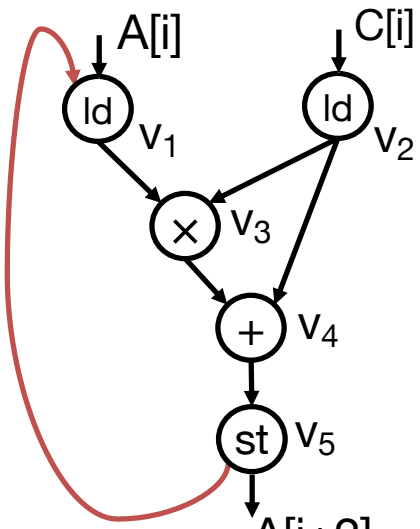


[Z. Zhang & B. Liu, ICCAD 2013]

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];
}
```

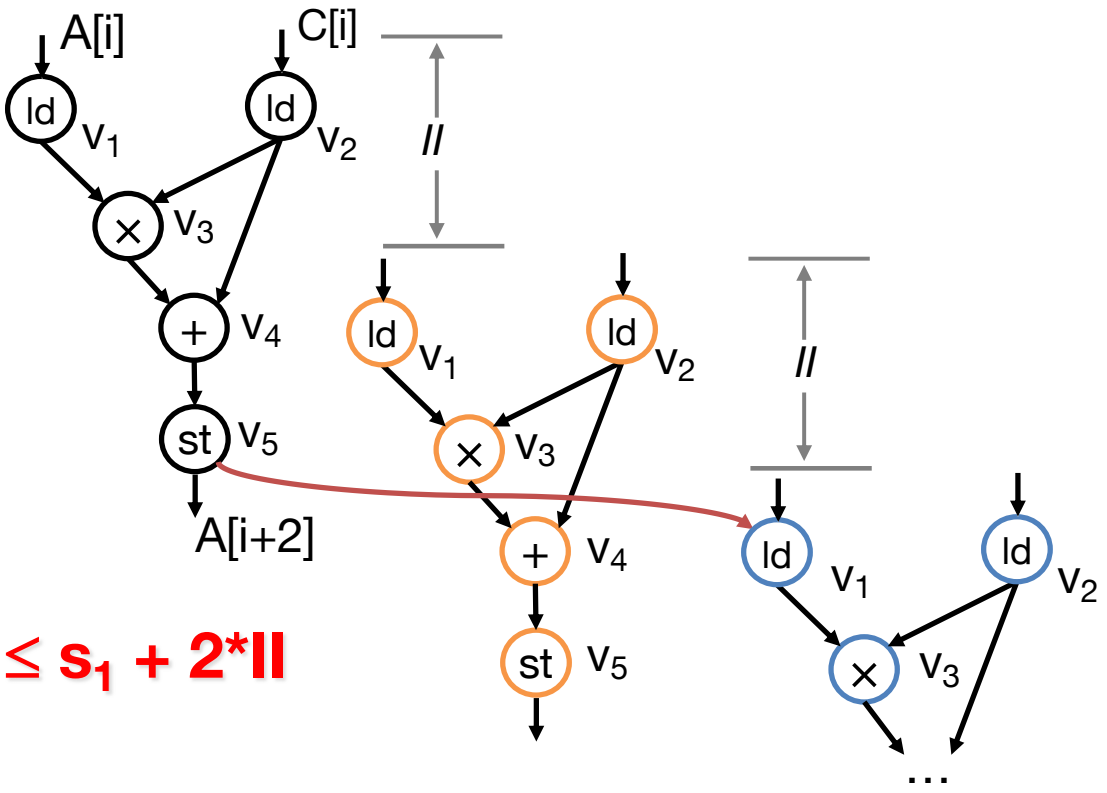


K = Dist(v_5, v_1) = 2

Modeling Loop-Carried Dependence with SDC

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

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



$s_5 \leq s_1 + 2*II$

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 which supports both nonpipelined and pipelined scheduling

Next Lecture

- More Pipelining
- Resource Sharing

Acknowledgements

- These slides contain/adapt materials developed by
  - Prof. Scott Mahlke (UMich)