More Pipelining
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

<table>
<thead>
<tr>
<th>Meeting</th>
<th>Schedule (am)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9:00~11:00</td>
</tr>
<tr>
<td>B</td>
<td>9:30~10:00</td>
</tr>
<tr>
<td>C</td>
<td>10:00~11:00</td>
</tr>
<tr>
<td>D</td>
<td>11:00~11:30</td>
</tr>
</tbody>
</table>

Conflict graph (chromatic number)

Compatibility graph (clique cover)

Gantt chart
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
  - $S_1 \rightarrow S_2$ : $S_1$ precedes $S_2$ in the program execution and computes a value that $S_2$ uses

**Example 1**

```c
for (i = 0; i < N; i++)
```

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

**Example 2**

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

Inter-iteration true dependence on “sum” (distance = 1)
for (i = 1; i < N; i++) {
    A[i-1] = b - a;
    B[i] = A[i] + 1
}
Output Dependences

- **Output dependence**
  - Also known as *Write After Write (WAW) dependence*
  - $S_1 \rightarrow^0 S_2$: $S_1$ precedes $S_2$ in the program execution and may write to a memory location that is later (over)written by $S_2$
  - Renaming (e.g., SSA) can resolve many WAW dependences

**Example**

```c
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)
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
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

[Diagram showing dependence graph with operations LD (Load), - (Subtraction), and + (Addition)]

Schedule of the body

[Diagram showing schedule with iterations 0 to 3 and operations LD and ST (Store)]

**Steady state determines both performance and resource usage**
Heuristics for Modulo Scheduling

- A common, iterative scheme of heuristic algorithms
  - Find a lower bound on \( M_{II} \): \( M_{II} = \max (\text{ResMII}, \text{RecMII}) \)
  - 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?  Yes

No

Increase II
```
Calculating Lower Bound of Initiation Interval

- **Minimum possible II (MII)**
  - $MII = \max (\text{ResMII}, \text{RecMII})$
  - A lower bound, not necessarily achievable

- **Resource constrained MII (ResMII)**
  - $\text{ResMII} = \max_i \left[ \frac{\text{OPs}(r_i)}{\text{Limit}(r_i)} \right]$
    
    OPs($r$): number of operations that use resource of type $r$
    
    Limit($r$): number of available resources of type $r$

- **Recurrence constrained MII (RecMII)**
  - $\text{RecMII} = \max_i \left[ \frac{\text{Latency}(c_i)}{\text{Distance}(c_i)} \right]$
    
    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

\[ \text{ResMII} = \max_i \left[ \frac{\text{OPs}(r_i)}{\text{Limit}(r_i)} \right] \]

- Take the max ratio among all resource types

<table>
<thead>
<tr>
<th>Dependence</th>
<th>Resource Allocation &amp; Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>a0</td>
<td>i0, i1, i2, i3, i4, i5</td>
</tr>
<tr>
<td>a1</td>
<td>i0, i1, i2, i3</td>
</tr>
<tr>
<td>a2</td>
<td>i0, i1, i2</td>
</tr>
<tr>
<td>a3</td>
<td>i0, i1</td>
</tr>
</tbody>
</table>

- 4 adders (a0~a3) | 2 adders (a0,a1)

- 0, 1, 2, 3, 4, 5 : time (cycles)
- a0, a1, a2, a3 : available adders
- i0, i1, i2, ... : loop iterations

- 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

\[ \text{RecMII} = \max_i \left[ \frac{\text{Latency}(c_i)}{\text{Distance}(c_i)} \right] \]

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

Analyze the MII for pipelining the above DFG

- 3 AddSub units available
- 2 Multipliers available
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 $u \rightarrow v$ with $\text{Distance}(u, v) = K$

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

$K = \text{Dist}(v_5, v_1) = 2$
Modeling Loop-Carried Dependence with SDC

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

for $(i = 0; i < N-2; i++)$
{
    $B[i] = A[i] \times C[i]$;
}

$s_5 \leq s_1 + 2 \times \text{II}$
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.

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

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

...
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

Prefix Sum:

\[
\begin{array}{c}
\text{cycle 1} & \text{cycle 2} & \text{cycle 3} & \text{cycle 4} \\
\hline
i = 0 & \text{ld}_1 & + & \text{st} \\
i = 1 & \text{ld}_1 & \text{ld}_2 & + & \text{st} \\
\end{array}
\]

out[0] = in[0];

\textbf{for} (\text{int} \ i = 1; \ i < N; \ i++) 
\text{out}[i] = \text{out}[i-1] + \text{in}[i];

\text{ld} – \text{Load} \\
\text{st} – \text{Store}
Prefix Sum: Code Optimization

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

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

<table>
<thead>
<tr>
<th></th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
<th>cycle 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0$</td>
<td>ld</td>
<td>+</td>
<td>st</td>
<td></td>
</tr>
<tr>
<td>$i = 1$</td>
<td>ld</td>
<td>+</td>
<td>st</td>
<td></td>
</tr>
</tbody>
</table>

Id – Load
st – Store
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)