ECE 5775 (Fall’17)
High-Level Digital Design Automation

More Pipelining
Announcements

- HW 2 due Monday 10/16 (no late submission)

- Second round paper bidding @ 5pm tomorrow on Piazza

- Talk by Prof. Margaret Martonosi (Princeton) @ 4:30pm today PHL 203 on “End of Moore’s Law Challenges and Opportunities: Computer Architecture Perspectives”
  - 5pm instructor office hour cancelled

- Midterm logistics
  - Exam time: Thursday 10/19 in class (75 mins)
    - Open book, open notes, closed Internet
  - In-class recitation on Tuesday 10/17
  - Instructor office hour next Thursday moved to Wednesday 10/18, 4:30-5:30pm (Rhodes 320)
Agenda

- Midterm coverage
- Modulo scheduling
- Case studies on pipelining
Midterm (20%)

Topics covered
  – Specialized computing
  – FPGAs
  – C-based synthesis
  – Control flow graph and SSA
  – Scheduling
  – Resource sharing
  – Pipelining
Key Topics (1)

▸ Algorithm basics
  – Time complexity, esp. big-O notation
  – Graphs
    • Trees, DAGs, topological sort
    • BDDs, timing analysis

▸ FPGAs
  – LUTs and LUT mapping

▸ C-based synthesis
  – Arbitrary precision and fixed-point types
  – Basic C-to-RTL mapping
  – Key HLS optimizations to improve design performance
Key Topics (2)

- Compiler analysis
  - Control data flow graph
  - Dominance relation
  - SSA

- Scheduling
  - TCS & RCS algorithms: ILP, list scheduling, SDC
    - Operation chaining, frequency/latency/resource constraints
  - Ability to devise a simple scheduling algorithm to optimize certain design metric
Key Topics (3)

- **Resource sharing**
  - Conflict and compatibility graphs
  - Left edge algorithm
  - Ability to determine minimum resource usage in # of functional units and/or registers, given a fixed schedule

- **Pipelining**
  - Dependence types
  - Ability to determine minimum II given a code snippet
    - Modulo scheduling concepts: MII, RecMII, ResMII
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
Pipelining through Modulo Scheduling

Dependence graph:

Schedule:

Steady state determines both performance resource usage
Modulo Scheduling

▸ A regular form of software pipelining technique
  – Also applies to loop (/ function) pipelining for hardware synthesis
  – Loop iterations (/ function) use the same schedule, which are initiated at a constant rate

▸ Advantages of modulo scheduling
  – Easy to analyze
    • Steady state determines performance and resource usage
  – Cost efficient
    • No code or hardware replication

▸ Optimization objective: (1) Minimize II under resource constraints or (2) minimize resource usage under II constraint
  – NP-hard in general
  – Optimal polynomial time solution exists without recurrences or resource constraints
Algorithmic Scheme for Modulo Scheduling

- Common scheme of heuristic algorithms
  - Find $MII$ (a lower bound on $II$) = $\max \{ \text{ResMII}, \text{RecMII} \}$
  - Look for a schedule with given $II$
  - If a feasible schedule not found, increase $II$ and try again
Calculating Lower Bound of Initiation Interval

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

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

- **Recurrence constrained MII (RecMII)**
  - \( \text{RecMII} = \max_i \left[ \frac{\text{Latency}(c_i)}{\text{Distance}(c_i)} \right] \)
    - \( \text{Latency}(c_i) \): total latency in dependence circuit \( c_i \)
    - \( \text{Distance}(c_i) \): total distance in dependence circuit \( c_i \)
Minimum II due to Resource Limits (ResMII)

Compute ResMII: Max among all types of resources
- \[ \text{ResMII} = \max_i \left\lceil \frac{\text{OPs}(r_i)}{\text{Limit}(r_i)} \right\rceil \]
Compute Recurrence Minimum II \((RecMII)\):
- Max among all circuits of:
  \[RecMII = \max_i \left[ \frac{\text{Latency}(c_i)}{\text{Distance}(c_i)} \right]\]
  - \(\text{Latency}(c)\) : sum of operation latencies along circuit \(c\)
  - \(\text{Distance}(c)\) : sum of dependence distances along circuit \(c\)
SDC-Based Modulo Scheduling

- SDC-based modulo scheduling
  - Unifies intra-iteration and inter-iteration scheduling constraints in a single SDC
  - Iterative algorithm with efficient incremental SDC update

---

Flowchart Image:
Example: Modeling Loop-Carried Dependence

- The dependence between two operations from different iterations is termed inter-iteration (loop-carried) dependence
  - Loop-carried dependence \( u \rightarrow v \) with \( \text{Dist}(u, v) = K \)
    \[ s_u + \text{Lat}_u \leq s_v + K^* \| \]

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

\[ \text{Dist}(v_5, v_1) = 2 \]
The constraint inequalities form a \textit{system of difference constraints} $SDC(X, C)$
\begin{itemize}
\item $X$ denotes the variable set
\item $C$ denotes the constraint set
\end{itemize}

The \textit{constraint graph} $G^c(V^c, E^c)$ of $SDC(X, C)$ is a weighted directed graph
\begin{itemize}
\item $x_i \in X : v_i \in V^c$
\item $x_i - x_j \leq b_k \in C : e(v_i, v_j) \in E^c$ and \text{weight}(e) = b_k
\end{itemize}
Feasibility Checking

**SDC**

$X: \{x_1, x_2, x_3, x_4, x_5\}$

- $x_1 - x_2 \leq 0$
- $x_1 - x_5 \leq -1$
- $x_2 - x_5 \leq 1$
- $x_3 - x_1 \leq -4$
- $x_1 - x_4 \leq 3$
- $x_4 - x_3 \leq -1$

- An SDC($X, C$) is feasible if and only if its constraint graph $G_c$ does not contain any negative cycles
  - Feasibility check can be done by solving a single-source shortest path problem

> Contradiction!

\[ x_3 - x_1 \leq -4 \]
\[ x_1 - x_4 \leq 3 \]
\[ x_4 - x_3 \leq -1 \]
\[ \rightarrow 0 \leq -2 \]

Contradiction!
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=?
    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 (i.e., ld) and writes (i.e., 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];
```

<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>ld₁</td>
<td>+</td>
<td>st</td>
<td></td>
</tr>
<tr>
<td>i = 1</td>
<td>ld₂</td>
<td></td>
<td>st</td>
<td></td>
</tr>
</tbody>
</table>

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

int tmp = 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>// = 1</td>
<td>ld</td>
<td>+</td>
<td>st</td>
</tr>
</tbody>
</table>

ld – Load
st – Store
Case Study: Convolution for Image Processing

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

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

![Input image frame](image1)

![3x3 convolution](image2)

![Output image frame](image3)
Achieving High Throughput with Pipelining

```c
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 Il=1 for 3x3 Convolution

Push three pixels into shift register window: one new pixel plus two pixels from line buffer.

Pixels in **line buffer** (2 lines stored)

New pixel fetched from input stream or frame buffer in off-chip memory

Output pixel produced by one convolution operation
Resulting Specialized Memory Hierarchy

- Memory architecture customized for convolution

Input pixel stream → Flip-Flops → Convolve → Output pixel stream

Processing window
Line buffers
Frame buffers
On-chip SRAMs
Off-chip DDR

24
HLS Code Snippet

```cpp
LineBuffer<2,C,pixel_t> linebuf;
Window<3,3,pixel_t> window;
for (int r = 1; r < R+1; r++) {
    for (int c = 1; c < C+1; c++) {
        #pragma HLS pipeline II=1
        pixel_t new_pixel = img[r][c];
        // Update shift window
        window.shift_left();
        if (r < R && c < C) {
            for (int i = 0; i < 2; i++)
                window.insert(buf[i][c]);
        } else { // zero padding
            for (int i = 0; i < 2; i++)
                window.insert(0);
        }
        window.insert(new_pixel);
        // Update line buffer
        linebuf.shift_up(c);
        if (r < R && c < C)
            linebuf[1].insert(c, new_pixel);
        else // Zero padding
            linebuf[1].insert(c, 0);
    }
    // Perform 3x3 convolution
    out[r-1][c-1] = convolve(window, weights);
}
```
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)