ECE 6775
High-Level Digital Design Automation
Fall 2023

Resource Sharing
Pipelining
Announcements

- Lab 3 is released (due Friday 10/6)
  - NO penalty for late submissions, *up to 6 days* past the deadline
  - Go through the CORDIC tutorial first

- HW 2 will be posted soon

- Jordan Dotzel (PhD TA) will give a tutorial on deep neural networks this Thursday

- Midterm on Thursday 10/19
  - In class, 75 mins
  - Open book, open notes, closed Internet
  - Coverage: Lectures 01~11 & 13
Agenda

- Resource sharing overview
  - Sub-problems: functional unit, register, and connectivity binding problems
  - Key concepts: compatibility and conflict graphs

- Introduction to pipelining
  - Parallel processing vs. Pipelining
  - Common forms in hardware accelerators
  - Throughput restrictions: resources and recurrences
Review: SDC-Based Scheduling

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

\[ s_i : \text{schedule variable for operation } i \]

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

- Timing constraints
  - Operation chaining is naturally supported

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

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

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

XLS: Accelerated HW Synthesis

What is XLS?

XLS implements a High Level Synthesis (HLS) toolchain which produces synthesizable designs (Verilog and SystemVerilog) from flexible, high-level descriptions of functionality. It is fully Open Source: Apache 2 licensed and developed via GitHub.

XLS (Accelerated HW Synthesis) aims to be the Software Development Kit (SDK) for the End of Moore’s Law (EoML) era. In this "age of specialization", software and hardware engineers must do more co-design across their domain boundaries — collaborate on shared artifacts, understand each other’s cost models, and share tooling/methodology. XLS attempts to leverage automation, software engineers, and machine cycles to accelerate this overall process.

https://github.com/google/xls/blob/main/xls/scheduling
Recap: A Typical HLS Flow

High-level Programming Languages (C/C++, OpenCL, SystemC, ...)

- Parsing
- Transformations
- Allocation
- Scheduling
- Binding

Intermediate Representation (IR)

Control data flow graph (CDFG)

```
if (condition) {
    ...
} else {
    t1 = a + b;
    t2 = c * d;
    t3 = e + f;
    t4 = t1 * t2;
    z = t4 - t3;
}
```

Finite state machines with datapath

3 cycles
Resource Sharing and Binding

- **Resource sharing** enables reuse of hardware resources to minimize cost, in resource usage/area/power
  - Typically carried out by binding in HLS
  - Other subtasks such allocation and scheduling greatly impact the resource sharing opportunities

- **Binding** maps operations, variables, and/or data transfers to the available resources
  - After scheduling: decide resource usage and detailed architecture (**focus of this lecture**)
  - Before scheduling: affect both area and delay
  - Simultaneous scheduling and binding: better result but more expensive
Binding Sub-problems

- Functional unit (FU) binding
  - Primary objective is to minimize the number of FUs
  - Considers connection cost

- Register binding
  - Primary objective is to minimize the number of registers
  - Considers connection cost

- Connectivity binding
  - Minimize connections by exploiting the commutative property of some operations / FUs
  - NP-hard
Sharing Conditions

- Functional units (registers) are shared by operations (variables) of same type whose *lifetimes* do not overlap.

- **Lifetime**: [birth-time, death-time)
  - Operation: The whole execution time (if unpipelined)
  - Variable: From the time this variable is defined to the time it is last used
### Operation Binding

#### Binding 1

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mul1</td>
<td>op1, op3</td>
</tr>
<tr>
<td>AddSub1</td>
<td>op2, op4</td>
</tr>
<tr>
<td>AddSub2</td>
<td>op5, op6</td>
</tr>
</tbody>
</table>

#### Binding 2

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mul1</td>
<td>op1, op3</td>
</tr>
<tr>
<td>AddSub1</td>
<td>op2, op4, op6</td>
</tr>
<tr>
<td>AddSub2</td>
<td>op5</td>
</tr>
</tbody>
</table>
Register Binding

Lifetimes crossing at least one clock edge

=> register(s) inferred

clock edge 1 2 3 4
Variable Lifetime Analysis

Variables v1, v2, and v3 can share the same register

<table>
<thead>
<tr>
<th></th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[1, 2)</td>
<td>[2, 3)</td>
<td>[3, 4)</td>
</tr>
</tbody>
</table>

Variable lifetimes [birth-time, death-time]
Compatibility and Conflict Graphs

- **Operation/variables compatibility**
  - Same type, non-overlapping lifetimes

- **Compatibility graph**
  - Vertices: operations/variables
  - Edges: compatibility relation

- **Conflict graph**: Complement of compatibility graph

A scheduled DFG (operations have the same type)

Compatibility graph

Conflict graph
Clique Cover Number and Chromatic Number

- Compatibility graph
  - Partition the graph into a **minimum number of cliques**
    - Clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge

- Conflict graph
  - Color the vertices by a **minimum number of colors** (chromatic number), where adjacent vertices cannot use the same color

Operations have same type

A scheduled DFG

**Clique partitioning** on compatibility graph

**Coloring** on conflict graph
Perfect Graphs

- Clique partitioning and graph coloring problems are NP-hard on general graphs, with the exception of perfect graphs.

- Definition of perfect graphs
  - For every induced subgraph, the size of the maximum (largest) clique equals the chromatic number of the subgraph.
  - Examples: bipartite graphs, chordal graphs, etc.
    - Chordal graphs: every cycle of four or more vertices has a chord, i.e., an edge between two vertices that are not consecutive in the cycle.
Interval Graph

- Intersection graphs of a (multi)set of intervals on a line
  - Vertices correspond to intervals
  - Edges correspond to interval intersection
  - A special class of chordal graphs

[Figure source: en.wikipedia.org/wiki/Interval_graph]
Left Edge Algorithm

- Problem statement
  - Given: Input is a group of intervals with starting and ending time
  - Goal: Minimize the number of colors of the corresponding interval graph

```
Repeat
create a new color group c

Repeat
  assign leftmost feasible interval to c
until no more feasible interval

until no more interval
```

Interval are sorted according to their left endpoints

Greedy algorithm, $O(n \log n)$ time
Left Edge Demonstration

Lifetime intervals with a given schedule

Assign colors (or tracks) using left edge algorithm

Corresponding colored conflict graph
Binding Impact on Multiplexer Network

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Operations</th>
<th>Functional Unit</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mul1</td>
<td>op1, op3</td>
<td>Mul1</td>
<td>op1, op3</td>
</tr>
<tr>
<td>AddSub1</td>
<td>op2, op4</td>
<td>AddSub1</td>
<td>op2, op4, op6</td>
</tr>
<tr>
<td>AddSub2</td>
<td>op5, op6</td>
<td>AddSub2</td>
<td>op5</td>
</tr>
</tbody>
</table>

**Binding 1**

**Binding 2**
Resource sharing directly impacts the complexity of the resulting datapath
- # of functional units and registers, multiplexer networks, etc.

Binding for resource usage minimization
- Left edge algorithm: greedy but optimal for DFGs
- NP-hard problem with the general form of CDFG
  • Polynomial-time algorithm exists for SSA-based register binding, although more registers are required

Connectivity binding problem (e.g., multiplexer minimization) is NP-Hard
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)

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

Dataflow of loop body

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

Pipelined schedule

- Here we assume multiplication (×) takes two cycles
Example: 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

```c
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

A coarse-grained pipeline for the optical flow algorithm
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

```plaintext
for (i = 1; i < N; ++i)
```

<table>
<thead>
<tr>
<th>i</th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
<th>cycle 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ld1</td>
<td>+</td>
<td>st</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ld2</td>
<td>ld1</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Port conflict

Assuming arrays A and B are held in two different SRAMs

Only one read port per SRAM $\rightarrow$ 1 load / cycle
Recurrence Restriction

- Recurrences restrict pipeline throughput
  - Computation of a component depends on a previous result from the same component

\[
\text{for } (i = 1; i < N; ++i) \\
\]

<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>(\text{ld}_1)</td>
<td>+</td>
<td>(\text{st})</td>
<td></td>
</tr>
<tr>
<td>(\text{ld}_2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| \(i = 1\) | \(\text{ld}_1\) | + |   | \(\text{st}\) |
| \(\text{ld}_2\) |   |   |   |

\(\parallel = 1\)

Id – Load
st – Store

Assume operation chaining is not allowed here due to cycle time constraint
Type of Recurrences

- **Recurrence** – if 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

- **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
  - $S_1 \rightarrow S_2$
    - Statement $S_1$ precedes statement $S_2$ in the program and computes a value that $S_2$ uses

**Example 1**
```
for (i = 0; i < N; i++)
```
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
  - Aka WAR (Write After Read) dependence
  - $S_1 \rightarrow^a S_2$
    - $S_1$ precedes $S_2$ and may read from a memory location that is later updated by $S_2$
  - Renaming (e.g., SSA) can resolve many of the WAR dependences

Example

```c
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

for (...) { 
    B[i] = A[i-1] + 1
    A[i] = B[i+1] + b
    B[i+2] = b - a
}
Next Lecture

- Tutorial on Deep Learning
Acknowledgements

- These slides contain/adapt materials developed by
  - Prof. Jason Cong (UCLA)
  - Prof. Deming Chen (UIUC)
  - Prof. Scott Mahlke (UMich)