More Binding
Pipelining
Announcements

- Lab 1 grades and first batch of quiz scores will be released by the end of this week
Outline

- More resource sharing
  - Perfect graphs
  - Left-edge algorithms

- Introduction to pipelining
  - Common forms in hardware accelerators
  - Throughput restrictions
  - Dependence types
Review: Compatibility and Conflict Graphs

- **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

A scheduled DFG

- **Clique partitioning** on compatibility graph
- **Coloring** on conflict graph

Operations have same type
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]
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(nlogn) 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>
</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 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, op6</td>
</tr>
<tr>
<td>AddSub2</td>
<td>op5</td>
</tr>
</tbody>
</table>

Binding 2
Binding Summary

- 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 (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 (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, latency = 2)

```
+   +   ×   ×   −
C0  v_1  v_2  v_3  v_4  v_5  
C1
C2
C3
C4
C5
```
Loop Pipelining

- Loop pipelining is one of the most important optimizations for high-level synthesis
  - Key metric: **Initiation Interval (II)** in # cycles
  - Allows a new iteration to begin processing every II cycles, before the previous iteration is complete

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

Given a 100-iteration loop with the loop body taking 50 cycles to execute
   - If we pipeline the loop with II = 1, how many cycles do we need to complete execution of the entire loop? \(50 + 99 = 149\)
   - What about II = 2? \(50 + 99 \times 2 = 248\)
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)
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

Only one memory read port → 1 load / cycle

Assuming ‘a’ and ‘b’ are held in two different memories

for (i = 1; i < N; ++i)
  b[i] = a[i-1] + a[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>ld_1</td>
<td></td>
<td>+</td>
<td>st</td>
</tr>
<tr>
<td>1</td>
<td>ld_2</td>
<td>ld_1</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

Port conflict
Recurrence Restriction

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

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

<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>(\text{ld}_2)</td>
<td>+</td>
<td>st</td>
</tr>
<tr>
<td>(i = 1)</td>
<td>(\text{ld}_1)</td>
<td>(\text{ld}_2)</td>
<td>+</td>
<td>st</td>
</tr>
</tbody>
</table>

\(\text{ld} \rightarrow \text{Load}\)
\(\text{st} \rightarrow \text{Store}\)

Assume chaining is not possible on memory reads (i.e., ld) and writes (i.e., st) due to cycle time constraint
Type of Recurrences

- Types of dependences
  - True dependences, anti-dependences, output dependences
  - Intra-iteration vs. inter-iteration dependences

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

- 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
  - S1 $\rightarrow^t S2$
    - Statement S1 precedes statement S2 in the program and computes a value that S2 uses

Example:

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

Inter-iteration true dependence on A (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:

```c
for (… 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 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
Next Class

- More pipelining