More Binding
Pipelining
Logistics

- Lab 3 due Friday 10/6
  - No late penalty for this assignment (up to 3 days late)

- HW 2 will be posted tomorrow
Agenda

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

- Introduction to pipelining
  - Common forms in hardware synthesis
  - 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]
# Example: Meeting Scheduling

<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>

**Interval graph**

**Conflict graph**
- chromatic number = 2
- max clique size = 2

**Compatibility graph**
- max clique size = 3
- chromatic number = 3

---

**Gantt chart**
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

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 Algorithms to Optimize MUX Network

▸ The connectivity binding problem is NP-Hard
  – Exact ILP formulations available but not scalable

▸ Graph-based heuristic algorithms
  – Clique partitioning [Tseng CAD’86] [Paulin DAC’86]
  – Bipartite [Huang DAC’90]
  – Min-cost network-flow [Chang DAC’95] [Chen ASPDAC’04] [Chen DAC’06]

▸ Meta-heuristics using simulated annealing, evolutionary algorithm, etc.
  – Pros: Consider multiple optimization parameters together for globally better results
  – Cons: Run-time and scalability
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 is intractable
Parallelization Techniques

- Parallel processing
  - Emphasizes concurrency by **replicating** a hardware structure several times
    - 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)
Loop Pipelining

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

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

**Pipeline schedule**

- **ld** – Load
- **st** – Store

**II = 1**
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?
  - 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)
A coarse-grained pipeline for the optical flow algorithm

```c
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       }
262   }
263 }
```
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

```cpp
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>ld₁</td>
<td></td>
<td></td>
<td>st</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>ld₂</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

Port conflict

**Only one memory read port → 1 load / cycle**

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

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

for (i = 1; i < N; ++i)
    a[i] = a[i-1] + a[i];

Assume chaining is not possible on memory reads (i.e., \text{ld}) and writes (i.e., \text{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)
True Dependences

- True dependence
  - Aka flow or RAW (Read After Write) dependence
  - $S_1 \rightarrow^t S_2$
    - Statement $S_1$ precedes statement $S_2$ in the program and computes a value that $S_2$ 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
  - \( \text{S1 precedes S2 and may write to a memory location that is later (over)written by S2} \)
  - Renaming (e.g., SSA) can resolve many of the WAW dependences

Example:

```plaintext
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
Before Next Class

- Next lecture: More pipelining (modulo scheduling)