Control Flow Graph
Announcements

- First problem set (HW1) is posted
  - Due Monday 9/17

- Lab 2 will be released on Thursday
Revisiting Some Important Graph Definitions

▸ What is a strongly connected component (SCC)?

▸ Is a DAG strongly connected?

▸ Does topological search apply to an SCC?

▸ What is the time complexity of a topological search (in terms of |V| and |E|)?
Graph Traversal

- **Purpose**: visit all the vertices in a particular order, check/update their properties along the way

- **Algorithms**
  - Depth-first search (DFS)
  - Breadth-first search (BFS)
  - Either can be used to realize topological sort

DFS order = a → ?

BFS order = a → ?
Outline

▸ More on BDDs and static timing analysis

▸ Basics of control data flow graph
  – Basic blocks
  – Control flow graph

▸ Dominance relation
  – Finding loops
Review: Two-Input Gates in BDD

Name the logic gates represented by the following BDDs

(a) 

(b)
More Virtues of BDDs

- There are many, but to list a few more:
  - Can represent an exponential number of paths with a DAG
  - Can evaluate an $n$-ary Boolean function in at most $n$ steps
    - By tracing paths to the 1 node, we can count or enumerate all solutions to equation $f = 1$
  - Every BDD node (not just root) represent some Boolean function in a canonical way
    - A BDD can be multi-rooted representing multiple Boolean functions sharing subgraphs
BDD Application on Functional Verification

Either prove equivalence
Or find counterexample(s)
  - Input values (x, y) for which these programs produce different results

```
bool P(bool x, bool y) { return ~(~x & ~y); }
```

```
bool Q(bool x, bool y) { return x ^ y; }
```

Does (P == Q) hold true?

Straight-line evaluation

<table>
<thead>
<tr>
<th>v1</th>
<th>~x</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td>~y</td>
</tr>
<tr>
<td>v3</td>
<td>v1 &amp; v2</td>
</tr>
<tr>
<td>P</td>
<td>~v3</td>
</tr>
<tr>
<td>Q</td>
<td>x ^ y</td>
</tr>
<tr>
<td>Is (P == Q) true?</td>
<td></td>
</tr>
</tbody>
</table>
BDD-based Equivalence Checking

\[ P = \neg v3 = \neg(\neg x \& \neg y) = x \lor y \]

OR gate

\[ Q = x \oplus y \]

XOR gate

\[ T = (P == Q) \]

Counterexample
Setting \( x = 1 \) & \( y = 1 \) leads to false output

Hence \( P \) does not equal \( Q \)
FPGA LUT Mapping Revisited

- Cone $C_v$: a subgraph rooted on a node $v$
  - K-feasible cone: $\#\text{inputs}(C_v) \leq K$ (Can occupy a K-input LUT)
  - K-feasible cut: The set of input nodes of a K-feasible $C_v$

Another 3-feasible cone with an associated cut = \{a, b, c\}

A 3-feasible cone with a cut = \{c, e, f\}
Timing Analysis with LUT Mapping

- Assumptions
  - K=3
  - All inputs arrive at time 0
  - Unit delay model: 3-LUT delay = 1; Zero edge delay

- Question: **Minimum arrival time** (AT) of each gate output?

  - AT(a) = 1
  - AT(d) = 1
  - AT(f) = ?
  - Associated cut?
  - AT(g) = ?
  - AT(h) = ?
  - AT(b) = 1
  - AT(e) = 1
  - AT(c) = 1
FPGA Design Flow with HLS

High-level Programming Languages

Compilation

Scheduling/ Pipelining, Binding

RTL

Logic Synth., Tech. Mapping, P&R, STA

Bitstream

Untimed high-level description

Timed design

FPGA

if (condition) {
    ...
} else {
    t_1 = a + b;
    t_2 = c * d;
    t_3 = e + f;
    t_4 = t_1 * t_2;
    z = t_4 - t_3;
}
A Typical HLS Flow

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

Parsing

Transformations

Intermediate Representation (IR)

Scheduling

Binding

RTL generation

if (condition) {
  ...
} else {
  t_1 = a + b;
  t_2 = c * d;
  t_3 = e + f;
  t_4 = t_1 * t_2;
  z = t_4 - t_3;
}

Control data flow graph (CDFG)

Finite state machines with datapath

3 cycles
Intermediate Representation (IR)

- Purposes of creating and operating on an IR
  - Encode the behavior of the program
  - Facilitate analysis
  - Facilitate optimization
  - Facilitate retargeting

- The IR we will focus on is control data flow graph (CDFG)
Program Flow Analysis

- Control flow analysis: determine control structure of a program and build control flow graphs (CFGs)

- Data flow analysis: determine the flow of data values and build data flow graphs (DFGs)
Basic Blocks

- **Basic block**: a sequence of consecutive intermediate language statements in which flow of control can only enter at the beginning and leave at the end

  - Only the last statement of a basic block can be a branch statement and only the first statement of a basic block can be a target of a branch
Partitioning a Program into Basic Blocks

- Each basic block begins with a leader statement

- Identify leader statements (i.e., the first statements of basic blocks) by using the following rules:
  
  - (i) The **first statement** in the program is a leader
  
  - (ii) Any statement that is the **target of a branch statement** is a leader (for most intermediate languages these are statements with an associated label)
  
  - (iii) Any statement that **immediately follows a branch or return** statement is a leader
Example: Forming the Basic Blocks

| (1) | p = 0         |
| (2) | i = 1         |
| (3) | \( t1 = 4 \times i \) |
| (4) | \( t2 = a[t1] \) |
| (5) | \( t3 = 4 \times i \) |
| (6) | \( t4 = b[t3] \) |
| (7) | \( t5 = t2 \times t4 \) |
| (8) | \( t6 = p + t5 \) |
| (9) | \( p = t6 \) |
| (10) | \( t7 = i + 1 \) |
| (11) | \( i = t7 \) |
| (12) | if \( i \leq 20 \) goto (3) |
| (13) | \( j = \ldots \) |

Basic Blocks:

B1

| (1) | \( p = 0 \) |
| (2) | \( i = 1 \) |

B2

| (3) | \( t1 = 4 \times i \) |
| (4) | \( t2 = a[t1] \) |
| (5) | \( t3 = 4 \times i \) |
| (6) | \( t4 = b[t3] \) |
| (7) | \( t5 = t2 \times t4 \) |
| (8) | \( t6 = p + t5 \) |
| (9) | \( p = t6 \) |
| (10) | \( t7 = i + 1 \) |
| (11) | \( i = t7 \) |
| (12) | if \( i \leq 20 \) goto (3) |

B3

| (13) | \( j = \ldots \) |

Leader statement is:
(1) the first in the program
(2) any that is the target of a branch
(3) any that immediately follows a branch
Control Flow Graph (CFG)

- A **control flow graph** (CFG), or simply a flow graph, is a directed graph in which:
  - (i) the nodes are basic blocks; and
  - (ii) the edges are induced from the possible flow of the program

- The basic block whose leader is the first intermediate language statement is called the **entry node**

- In a CFG we assume no information about data values
  - an edge in the CFG means that the program **may** take that path
Example: Control Flow Graph Formation

1. \( p = 0 \)
2. \( i = 1 \)
3. \( t_1 = 4 \times i \)
4. \( t_2 = a[t_1] \)
5. \( t_3 = 4 \times i \)
6. \( t_4 = b[t_3] \)
7. \( t_5 = t_2 \times t_4 \)
8. \( t_6 = p + t_5 \)
9. \( p = t_6 \)
10. \( t_7 = i + 1 \)
11. \( i = t_7 \)
12. if \( i \leq 20 \) goto (3)
13. \( j = \ldots \)
Dominators

- A node $p$ in a CFG **dominates** a node $q$ if every path from the entry node to $q$ goes through $p$. We say that node $p$ is a **dominator** of node $q$.

- The **dominator set** of node $q$, $\text{DOM}(q)$, is formed by all nodes that dominate $q$.
  - By definition, each node dominates itself therefore, $q \in \text{DOM}(q)$. 
  

Dominance Relation

- **Definition:** Let $G = (N, E, s)$ denote a CFG, where
  - $N$: set of nodes
  - $E$: set of edges
  - $s$: entry node and
  let $p \in N$, $q \in N$
  - $p$ dominates $q$, written $p \leq q$
    - $p \in \text{DOM}(q)$
  - $p$ properly (strictly) dominates $q$, written $p < q$ if $p \leq q$ and $p \neq q$
  - $p$ immediately (or directly) dominates $q$, written $p <_d q$ if $p < q$ and there is no $t \in N$ such that $p < t < q$
    - $p = \text{IDOM}(q)$
Example: Dominance Relation

- **Dominator sets:**
  - $\text{DOM}(1) = \{1\}$
  - $\text{DOM}(2) = \{1, 2\}$
  - $\text{DOM}(3) = \{1, 2, 3\}$
  - $\text{DOM}(10) = \{1, 2, 10\}$

- **Immediate domination:**
  - $1 \preceq_d 2, 2 \preceq_d 3, \ldots$
  - $\text{IDOM}(2) = 1, \text{IDOM}(3) = 2 \ldots$
Assume that node $P$ is an immediate dominator of node $Q$

Question: Is $P$ necessarily an immediate predecessor of $Q$ in the CFG?

Answer: **NO**
Identifying Loops

- **Motivation**: Programs spend most of the execution time in loops, therefore there is a larger payoff for optimizations that exploit loop structure.

- **Goal**: Identify loops in a CFG, not sensitive to input syntax.
  - Create an uniform treatment for program loops written using different syntactical constructs (e.g. while, for, goto).

- **Approach**: Use a general approach based on analyzing graph-theoretical properties of the CFG.
Loop Definition

- Definition: A **strongly connected component** (SCC) of the CFG, with
  - a single entry point called the **header** which dominates all nodes in the SCC
Question: In the CFG below, nodes 2 and 3 form an SCC; but **do they form a loop?**
Finding Loops

- Loop identification algorithm
  - Find an edge $B \rightarrow H$ where $H$ dominates $B$; This edge is called a **back-edge**
  
  - Find all nodes that (1) are dominated by $H$ and (2) can reach $B$ through nodes dominated by $H$; add them to the loop
    - $H$ and $B$ are naturally included
Finding Loops

Find all back edges in this graph and the natural loop associated with each back edge.
Summary

- **Basic Blocks**
  - Group of statements that execute atomically

- **Control Flow Graphs**
  - Model the control dependences between basic blocks

- **Dominance relations**
  - Shows control dependences between BBs
  - Used to determine natural loops
Next Class

- Next lecture: Static single assignment
Acknowledgements

- These slides contain/adapt materials developed by
  - Forrest Brewer (UCSB)
  - Ryan Kastner (UCSD)
  - Prof. José Amaral (Alberta)