Control Data Flow Graph
Announcements

- Lab 2 released (Due Nov 10th)
Agenda

- HLS compilation flow

- Intermediate representation (IR)
  - Control data flow graph

- Control flow analysis
  - Basic blocks
  - Dominance relation: Finding loops

- Dataflow analysis
  - Static single assignment (SSA)
FPGA Design Flow with HLS

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)

Allocation

Scheduling

Binding

RTL generation

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

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

Basic Blocks:

B1

B2

B3

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

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

```plaintext
(1) p = 0
(2) i = 1
(3) t1 = 4 * i
(4) t2 = a[t1]
(5) t3 = 4 * i
(6) t4 = b[t3]
(7) t5 = t2 * t4
(8) t6 = p + t5
(9) p = t6
(10) t7 = i + 1
(11) i = t7
(12) if i <= 20 goto (3)
(13) j = ...
```
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 $q$.
  - Each node dominates itself by definition; thus $q \in \text{DOM}(q)$.
**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:**
  - $1 <_d 2$, $2 <_d 3$, …
  - $\text{IDOM}(2) = 1$, $\text{IDOM}(3) = 2$, …
Dominance Relationship: True or False

- Does a node strictly (or properly) dominate itself?

- Does a predecessor of a node B always dominate B?

- Suppose a node A dominates all of B’s predecessors. Does A also dominate B?
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 syntax of the input language.
  - Create a 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
Is it a Loop?

- 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 these nodes 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.
Finding Loops (1)

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

(9,1)
Finding Loops (1)

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

(9,1) Entire graph
Finding Loops (2)

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

(9,1) Entire graph
(10,7)
Intuition of Dominance Relation

Imagine a source of light at the entry node, and that the edges are optical fibers.

To find which nodes are dominated by a given node, place an opaque barrier at that node and observe which nodes become dark.
Finding Loops (2)

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

(9,1) Entire graph
(10,7)
Finding Loops (2)

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

(9,1)    Entire graph
(10,7)   {7,8,10}
Finding Loops (3)

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

(9,1)  Entire graph
(10,7)  {7,8,10}
(7,4)
Finding Loops (3)

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

(9,1) Entire graph
(10,7) {7,8,10}
(7,4)
Finding Loops (3)

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

(9,1)   Entire graph
(10,7)  {7,8,10}
(7,4)   {4,5,6,7,8,10}
Data Flow Analysis with SSA

- Static single assignment (SSA) form is a restricted IR where
  - Each variable definition has a **unique** name
  - Each variable use refers to a single definition

- SSA simplifies **data flow analysis** & many compiler optimizations
  - Eliminates artificial dependences (on scalars)
    - Write-after-write
    - Write-after-read
SSA within a Basic Block

- Assign each variable definition a unique name
- Update the uses accordingly

Original code

\[
\begin{align*}
x &= \text{read()} \\
x &= x \times 5 \\
x &= x + 1 \\
y &= x \times 9
\end{align*}
\]

SSA form

\[
\begin{align*}
x_0 &= \text{read()} \\
x_1 &= x_0 \times 5 \\
x_2 &= x_1 + 1 \\
y &= x_2 \times 9
\end{align*}
\]

Corresponding data flow graph

\[
\begin{align*}
x_0 &\rightarrow x \\x_1 &\rightarrow 1 \\x_2 &\rightarrow 9 \\
y &\rightarrow \text{Corresponding data flow graph}
\end{align*}
\]
SSA with Control Flow

- Consider a situation where two control-flow paths merge
  - e.g., due to an if-then-else statement or a loop

```
x = read()
if (x > 0)
y = 5
else
  y = 10
x = y
```

```
x0 = read()
if (x0 > 0)
y0 = 5
y1 = 10
x1 = y
```

should this be $y_0$ or $y_1$?
Introducing $\phi$-Node

- Inserts special join functions (called $\phi$-nodes or PHI nodes) at points where different control flow paths converge

Note: $\phi$ is not an executable function!
To generate executable code from this form, appropriate copy statements need to be generated in the predecessors (in other words, reversing the SSA process for code generation)
SSA in a Loop

- Insert $\phi$-nodes in the loop header block

```plaintext
x = 0
i = 1
while (i < 10) {
    x = x + i
    i = i + 1
}
```
SSA Applications

▶ SSA form simplifies data flow analysis and many code transformations
  – Primarily due to explicit & simplified (sparse) def-use chains

▶ Here we show a simple yet useful transformation
  – Dead code elimination
Dead Code in CDFG

- Dead code is either
  - Unreachable code
  - Definitions never used

- Dead statements?

```
x = a + b
y = c + d
x = a - b
y = c - d
z = x + y
z = z + 1
f(x, y)
```
Dead Code Elimination with SSA

Iteratively remove unused definitions:
remove $y_1$, $z_2$ (and B4) $\rightarrow$ then remove $z_1$
Summary

- Importance of compilers
  - Essential component of software development flow
  - Essential component of high-level synthesis

- A good intermediate representation (IR) enables efficient and effective analysis and optimization
  - Dominance relation helps effective CFG analysis
  - SSA form facilitates efficient IR-level optimization
Next Lecture

- Scheduling algorithms
Acknowledgements

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