Where are we?

- Analysis
  - Control Flow/Predicate
    - Treat basic blocks as a black box
    - Only look at branches
  - Dataflow
    - Look inside basic blocks
    - What is computed where?
- Transformations
  - Register Allocation
  - Optimization
  - Scheduling

Dataflow Analysis Motivation (Optimization)

Constant Propagation and Dead Code Elimination:

```
  r1 = 4
  r2 = r1 + 5

  r1 = 4
  r2 = r1 + 5
```

Needs dominator, liveness, and reaching definition information.

Dataflow Analysis Motivation (Register Allocation)

Register Allocation:

- Infinite number of registers (virtual registers) must be mapped to a limited number of real registers.
- Pseudo-assembly must be examined by *live variable analysis* to determine which virtual registers contain values which may be used later.
- Virtual registers which are not simultaneously *live* may be mapped onto the same real register.

1. \( r2 = r1 + 1 \)
2. \( r3 = M[r2] \)
3. \( r4 = r3 + 4 \)
4. ```LOAD r5 = M[r2 + r4]```
Dataflow Analysis Introduction

Dataflow analysis – Collection of information that summarizes the creation/destruction of values in a program. Used to identify legal optimization opportunities.

Pick an arbitrary point in the program
- Which VRs contain useful data values? (liveness or upward exposed uses)
- Which definitions may reach this point? (reaching definitions)
- Which definitions are guaranteed to reach this point? (available definitions)
- Which uses below are exposed? (downward exposed uses)

Iterative Dataflow Analysis Framework

- Iterative dataflow analysis equations are applied in an iterative fashion until \( \text{IN} \) and \( \text{OUT} \) sets do not change.
- Typically done in (FORWARD or REVERSE) topological sort order of CFG for efficiency.
- \( \text{IN} \) and \( \text{OUT} \) sets initialized to \( \emptyset \).

For each node \( n \) {
  \[
  \text{IN}[n] = \text{OUT}[n] = \emptyset;
  \]
}
Repeat {
  For each node \( n \) in forward/reverse topological order {
    \[
    \text{IN}'[n] = \text{IN}[n];
    \text{OUT}'[n] = \text{OUT}[n];
    \text{IN}[n], \text{OUT}[n] = \text{(Equations)};
    \]
  } until \( \text{IN}'[n] = \text{IN}[n] \) and \( \text{OUT}'[n] = \text{OUT}[n] \) for all \( n \).

Live Variable Analysis

Liveness Definitions:
- A source (RHS) register \( t \) is a use of \( t \).
- A destination (LHS) register \( t \) is a definition of \( t \).
- A register \( t \) is live on edge \( e \) if there exists a path from \( e \) to a use of \( t \) that does not go through a definition of \( t \).
- Register \( t \) is live-in at CFG node \( n \) if \( t \) is live on any in-edge of \( n \).
- Register \( t \) is live-out at CFG node \( n \) if \( t \) is live on any out-edge of \( n \).
Live Variable Analysis

Live Variable Analysis Equation:
- Set definition \((A[n]): USE[n]\) - the set of registers that \(n\) uses.
- Set definition \((B[n]): DEF[n]\) - the set of registers that \(n\) defines.
- Transfer function \(f(A, B, OUT))\): \(USE[n] \cup (OUT[n] - DEF[n])\)
- Confluence operator \((\lor): \lor\)
- Direction: REVERSE

\[
OUT[n] = \bigcup_{s \in \text{Succ}[n]} \text{IN}[s] \\
\text{IN}[n] = USE[n] \cup (OUT[n] - DEF[n])
\]

Live Variable Analysis Application 1:
Register Allocation
- In compiler, we assume unbounded number of registers.
- Virtual Registers - no limits
- Physical Registers exist in the target machine

Register allocation maps virtual to physical registers

Steps:
1. Perform live variable analysis.
2. Build *interference graph*.
3. Color interference graph with real registers.

Interference Graph
- Node \(t\) corresponds to virtual register \(t\).
- Edge \(<t_i, t_j>\) exists if registers \(t_i, t_j\) have overlapping live ranges.
- For some node \(n\), if \(DEF[n] = \{a\}\) and \(OUT[n] = \{b_1, b_2, ... b_k\}\), then add interference edges: \(<a, b_1>, <a, b_2>, <a, b_k>\)

Interference Graph For Example:

<table>
<thead>
<tr>
<th>Node</th>
<th>DEF</th>
<th>OUT</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r1</td>
<td>r1, r3</td>
<td>r3</td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2, r3</td>
<td>r1, r3</td>
</tr>
<tr>
<td>3</td>
<td>r3</td>
<td>r2, r3</td>
<td>r2, r3</td>
</tr>
<tr>
<td>4</td>
<td>r1</td>
<td>r1, r3</td>
<td>r2, r3</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>r1, r3</td>
<td>r1, r3</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>r3</td>
</tr>
</tbody>
</table>

Virtual registers \(r1\) and \(r2\) may be mapped to same real registers.
Live Variable Analysis Application 2: Dead Code Elimination

- Given statement $s$ with a definition and no side-effects:
  
  $r_1 = r_2 + r_3, r_1 = M[r_2], \text{ or } r_1 = r_2$

  If $r_1$ is not live at the end of $s$, then the $s$ is dead.

- Dead statements can be deleted.

- Given statement $s$ without a definition or side-effects:
  
  $r_1 = \text{call\ FUN_NAME, } M[r_1] = r_2$

  Even if $r_1$ is not live at the end of $s$, it is not dead.

Example:

$$
\begin{align*}
  r_1 & = r_2 + 1 \\
  r_2 & = r_2 + 2 \\
  r_1 & = r_2 + 3 \\
  M[r_1] & = r_2
\end{align*}
$$

Class Problem

Reaching Definition Analysis ($rdefs$)

Determines whether definition of register $t$ directly affects use of $t$ at some point in program.

Reaching Definition Definitions:

- **unambiguous** - instruction explicitly defines register $t$.
- **ambiguous** - instruction may or may not define register $t$.
  - Global variables in a function call.
  - No ambiguous definitions in tiger since all globals are stored in memory.
- Definition of $d$ (of $t$) reaches statement $u$ if a path of CFG edges exists from $d$ to $u$ that does not pass through an unambiguous definition of $t$.
- One unambiguous and many ambiguous definitions of $t$ may reach $u$ on a single path.

Reaching Definition Analysis Equation:

- Set definition $(A[n]): GEN[n]$ - the set of definition id’s that $n$ creates.
- Set definition $(B[n]): KILL[n]$ - the set of definition id’s that $n$ kills.
  - $defs(t)$ - set of all definition id’s of register $t$.
- Transfer function $(f(A, B, IN))$: $GEN[n] \cup (IN[n] - KILL[n])$
- Confluence operator ($\vee$): $\cup$
- Direction: FORWARD

$$
\begin{align*}
  IN[n] &= \cup_{p \in PRED[n]} OUT[p] \\
  OUT[n] &= GEN[n] \cup (IN[n] - KILL[n])
\end{align*}
$$
Reaching Definitions Example

1: \( r_1 = r_2 + r_3 \)
2: \( r_6 = r_4 - r_5 \)
3: \( r_4 = 4 \)
4: \( r_6 = 8 \)
5: \( r_5 = r_2 + r_3 \)
6: \( r_7 = r_4 - r_5 \)

defs 1 and 2 reach this point

defs 1, 3, 4 reach this point

def 2 is killed by 4

defs 1, 3, 5, 6 reach this point
defs 2, 4 are killed by 5

Reaching Definitions Application 1: Constant Propagation

- Given Statement \( d: a = c \) where \( a \) is constant
- Given Statement \( w: t = a \oplus b \)
- If statement \( d \) reach \( w \) and no other definition of \( a \) reaches \( w \), then replace \( w \) by \( t = c \oplus b \).

An Aside of Sorts: Constant Folding

- Given Statement \( d: t = a \oplus b \)
- If \( a \) and \( b \) are constant, compute \( c \) as \( a \oplus b \), replace \( d \) by \( t = c \)
### Class Problem

**Reaching definitions**
- Calculate GEN/KILL for each BB
- Calculate IN/OUT for each BB

```
r1 = 3
r2 = r3
r3 = r4
```

```
r1 = r1 + 1
r7 = r1 * r2
```

```
r2 = 0
r2 = r2 + 1
r4 = r2 + r1
r9 = r4 + r8
```

### Some Things to Think About

- **Liveness and reaching defs are basically the same thing**
  - Dataflow analysis framework
  - Upward exposed uses
  - Downward exposed uses
  - Upward exposed definitions
  - Downward exposed definitions

- **Dataflow can be slow**
  - How to implement it efficiently?
  - How to represent the info?

- **Predicates**
  - Throw a monkey wrench into this stuff
  - So, how are predicates handled?

### Beyond Liveness or Upward Exposed Uses

- **Upward exposed definitions**
  - IN = GEN + (OUT − KILL)
  - OUT = Union(IN(successors))
  - Walk ops reverse order
    - GEN += dest; KILL += dest

- **Downward exposed uses**
  - IN = Union(OUT(predecessors))
  - OUT = GEN + (IN-KILL)
  - Walk ops forward order
    - GEN += src; KILL -= src;
    - GEN -= dest; KILL += dest;

- **Downward exposed definitions**
  - IN = Union(OUT(predecessors))
  - OUT = GEN + (IN-KILL)
  - Walk ops forward order
    - GEN += dest; KILL += dest;

### Example – Upward Exposed Definitions

```
BB1
r1 = MEM[r2+0]
r2 = r2 + 1
r3 = r1 * r4
```

```
BB2
r1 = r1 + 5
r3 = r5 − r1
r7 = r3 * 2
```

```
BB3
r2 = 0
r7 = 23
r1 = 4
```

```
BB4
r3 = r3 + r7
r1 = r3 − r8
r3 = r1 * 2
```
- Convenient way to access/use reaching definitions info
- Def-Use chains
  - Given a def, what are all the possible consumers of the operand produced
  - Maybe consumer
- Use-Def chains
  - Given a use, what are all the possible producers of the operand consumed
  - Maybe producer