IR Generation

May 13, 2013
Why Even Google Will Embrace Cellphone Chips in the Data Center

BY CADE METZ 05.09.13  9:30 AM

Inside Google's Douglas County, Georgia data center. Photo: Google/Connie Zhou

Jason Mars is a rarity. He’s an outsider with regular access to Google’s data centers.

Mars is a professor of computer science at the University of California, San Diego, and about five years ago, during a conference for computer science researchers, he met a Googler named Robert Hundt. Among so many other things, Hundt is responsible for a set of tools that track the performance of Google’s massive computing facilities — widely regarded as the most advanced on the internet — and somewhere along the line, he asked Mars to help him sift through the reams of information produced by these tools.
Midterm Results
• Grades already in
• Repeal policy
  • Talk to TA by the end of the Tuesday
  • reserve the right to regrade the entire exam
CSE131 - Spring, 2013 - Midterm Statistics

Statistics

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Percent of Midterm</td>
<td>92.31%</td>
</tr>
<tr>
<td>Percent of Course</td>
<td>18.46%</td>
</tr>
<tr>
<td>Maximum Points</td>
<td>120</td>
</tr>
<tr>
<td>Total Number of Scores</td>
<td>61</td>
</tr>
<tr>
<td>Mean</td>
<td>107.54</td>
</tr>
<tr>
<td>Median</td>
<td>110.00</td>
</tr>
<tr>
<td>High Score</td>
<td>120 (4)</td>
</tr>
<tr>
<td>Low Score</td>
<td>0 (1)</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>16.33</td>
</tr>
</tbody>
</table>

Histogram

<table>
<thead>
<tr>
<th>Scores</th>
<th># of students</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;116 - 120</td>
<td>14</td>
</tr>
<tr>
<td>&gt;112 - 116</td>
<td>11</td>
</tr>
<tr>
<td>&gt;108 - 112</td>
<td>8</td>
</tr>
<tr>
<td>&gt;104 - 108</td>
<td>12</td>
</tr>
<tr>
<td>&gt;100 - 104</td>
<td>14</td>
</tr>
<tr>
<td>&gt;96 - 100</td>
<td>9</td>
</tr>
<tr>
<td>&gt;92 - 96</td>
<td>6</td>
</tr>
<tr>
<td>&gt;88 - 92</td>
<td>8</td>
</tr>
<tr>
<td>&gt;84 - 88</td>
<td>8</td>
</tr>
<tr>
<td>&gt;80 - 84</td>
<td>6</td>
</tr>
<tr>
<td>&gt;76 - 80</td>
<td>6</td>
</tr>
<tr>
<td>&gt;72 - 76</td>
<td>2</td>
</tr>
</tbody>
</table>
# CSE131 - Spring, 2013 - Mid extra credit Statistics

<table>
<thead>
<tr>
<th>Statistics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Percent of Midterm:</td>
<td>7.69%</td>
</tr>
<tr>
<td>Percent of Course:</td>
<td>1.54%</td>
</tr>
<tr>
<td>Maximum Points:</td>
<td>10</td>
</tr>
<tr>
<td>Total Number of Scores:</td>
<td>61</td>
</tr>
<tr>
<td>Mean:</td>
<td>2.13</td>
</tr>
<tr>
<td>Median:</td>
<td>0.00</td>
</tr>
<tr>
<td>High Score:</td>
<td>10 (13)</td>
</tr>
<tr>
<td>Low Score:</td>
<td>0 (48)</td>
</tr>
<tr>
<td>Standard Deviation:</td>
<td>4.13</td>
</tr>
</tbody>
</table>

## Histogram

<table>
<thead>
<tr>
<th>Scores</th>
<th># of students</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;0 - 2</td>
<td></td>
</tr>
<tr>
<td>&gt;2 - 6</td>
<td></td>
</tr>
<tr>
<td>&gt;6 - 10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>
WHAT DOESN'T KILL YOU

MAKES YOU STRONGER
- P1, P2, Midterm  (45%)
- P3, P4, Final  (55%)
Feedback

• Keep/Start:
  • slides online before class (condensed slides)
  • more discussion in class
  • more code gen/optimization/backend/research topics
  • homework
  • extra credit/extracurricular materials
  • more practice problems
  • talk about real world applications
  • podcast
  • meme
• Quit:
  • talking too fast
  • small voice
  • traveling
  • math
  • start code compilation errors
  • smoking?
Where We Are

Source Code

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

IR Optimization

Code Generation

Optimization

Machine Code
Overview for Today

• The Final Assignment
• Introduction to TAC:
  • TAC for simple expressions.
  • TAC for functions and function calls.
  • TAC for objects.
  • TAC for arrays.
• Generating TAC.
• A few low-level details.
The Final Assignment

- **Goal:** Generate TAC IR for Decaf programs.
- We provide a code generator to produce MIPS assembly.
  - You can run your programs using *spim*, the MIPS simulator.
- You must also take care of some low-level details:
  - Assign all parameters, local variables, and temporaries positions in a stack frame.
  - Assign all global variables positions in the global memory segment.
  - Assign all fields in a class an offset from the base of the object.
- You **should not** need to know MIPS to do this; all details will be covered in lecture.
- If you have any questions on MIPS, please feel to ask!
An Important Detail

• When generating IR at this level, you do not need to worry about optimizing it.
• It's okay to generate IR that has lots of unnecessary assignments, redundant computations, etc.
• We'll see how to optimize IR code later this week and at the start of next week.
  • It's tricky, but extremely cool!
Three-Address Code

- Or “**TAC**”
- The IR that you will be using for the final programming project.
- High-level assembly where each operation has at most three operands.
- Uses explicit runtime stack for function calls.
- Uses vtables for dynamic dispatch.
Sample TAC Code

```c
int x;
int y;

int x2 = x * x;
int y2 = y * y;
int r2 = x2 + y2;
```
int x;
int y;

int x2 = x * x;
int y2 = y * y;
int r2 = x2 + y2;
Sample TAC Code

int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
Sample TAC Code

```c
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

```c
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;

_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
Temporary Variables

• The “three” in “three-address code” refers to the number of operands in any instruction.

• Evaluating an expression with more than three subexpressions requires the introduction of temporary variables.

• This is actually a lot easier than you might think; we'll see how to do it later on.
Sample TAC Code

```c
int a;
int b;

a = 5 + 2 * b;
```
int a;
int b;
a = 5 + 2 * b;

_t0 = 5;
_t1 = 2 * b;
a = _t0 + _t1;
Sample TAC Code

```c
int a;
int b;
a = 5 + 2 * b;
```

TAC allows for instructions with two operands.

```c
_t0 = 5;
_t1 = 2 * b;
a = _t0 + _t1;
```
Simple TAC Instructions

- **Variable assignment** allows assignments of the form
  - `var = constant;`
  - `var_1 = var_2;`
  - `var_1 = var_2 op var_3;`
  - `var_1 = constant op var_2;`
  - `var_1 = var_2 op constant;`
  - `var = constant_1 op constant_2;`
- Permitted operators are `+`, `-`, `*`, `/`, `%`.
- How would you compile `y = -x;`?
Simple TAC Instructions

- **Variable assignment** allows assignments of the form
  
  - `var = constant;`
  - `var₁ = var₂;`
  - `var₁ = var₂ op var₃;`
  - `var₁ = constant op var₂;`
  - `var₁ = var₂ op constant;`
  - `var = constant₁ op constant₂;`

- Permitted operators are `+`, `-`, `*`, `/`, `%`.

- How would you compile `y = -x;`?
  
  `y = 0 - x;`  
  `y = -1 * x;`
One More with bools

```c
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```
One More with bools

```cpp
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```cpp
_t0 = x + x;
_t1 = y;
_b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
_b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
_b3 = _t5 < _t4;
```
TAC with **bools**

- Boolean variables are represented as integers that have zero or nonzero values.
- In addition to the arithmetic operator, TAC supports `<`, `==`, `||`, and `&&`.
- How might you compile \[ b = (x \leq y) \]?
TAC with bools

- Boolean variables are represented as integers that have zero or nonzero values.
- In addition to the arithmetic operator, TAC supports $<$, $==$, $||$, and $&&$.
- How might you compile $b = (x \leq y)$?

```c
_t0 = x < y;
_t1 = x == y;
b = _t0 || _t1;
```
Control Flow Statements

```c
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;
z = z * z;
Control Flow Statements

```c
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```c
_t0 = x < y;
IfZ _t0 Goto _L0;
    z = x;
Goto _L1;

_L0:
    z = y;
_L1:
    z = z * z;
```
Control Flow Statements

```c
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```assembly
_t0 = x < y;
IfZ _t0 Goto _L0;
z = x;
Goto _L1;

_L0:
z = y;

_L1:
z = z * z;
```
Labels

- TAC allows for **named labels** indicating particular points in the code that can be jumped to.
- There are two control flow instructions:
  - `Goto label;`
  - `IfZ value Goto label;`
- Note that **IfZ** is always paired with `Goto`. 
Control Flow Statements

```c
int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;
```
Control Flow Statements

```c
int x;
int y;

while (x < y) {
    x = x * 2;
}

y = x;
```

```c
_L0:
    _t0 = x < y;
    IfZ _t0 Goto _L1;
    x = x * 2;
    Goto _L0;
_L1:
    y = x;
```
A Complete Decaf Program

```c
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}

main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
    _L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
    _L1:
    EndFunc;
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}

main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
    _L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
    _L1:
    EndFunc;
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}

main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
    _L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
    _L1:
    EndFunc;
A Complete Decaf Program

void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}

main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
    _L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
    _L1:
    EndFunc;
Compiling Functions

• Decaf functions consist of four pieces:
  • A **label** identifying the start of the function.
    - *(Why?)*
  • A **BeginFunc N;** instruction reserving N bytes of space for locals and temporaries.
  • The body of the function.
  • An **EndFunc;** instruction marking the end of the function.
    - When reached, cleans up stack frame and returns.
A Logical Decaf Stack Frame

Stack frame for function \( f(a, \ldots, n) \)

- Param \( N \)
- Param \( N - 1 \)
- ...
- Param 1
- Storage for Locals and Temporaries
A Logical Decaf Stack Frame

Stack frame for function $f(a, ..., n)$

- Param $N$
- Param $N - 1$
- ...
- Param 1
- Storage for Locals and Temporaries
- Param $M$
A Logical Decaf Stack Frame

Stack frame for function \( f(a, ..., n) \)

- Param N
- Param \( N - 1 \)
- \( ... \)
- Param 1
- Storage for Locals and Temporaries
- Param M
- \( ... \)
A Logical Decaf Stack Frame

Stack frame for function f(a, ..., n)

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries
  - Param M
  - ...
  - Param 1
A Logical Decaf Stack Frame

Stack frame for function \( f(a, \ldots, n) \)
A Logical Decaf Stack Frame

Stack frame for function $f(a, ..., n)$

- Param $N$
- Param $N - 1$
- ...
- Param 1
- Storage for Locals and Temporaries

Stack frame for function $g(a, ..., m)$

- Param $M$
- ...
- Param 1
- Storage for Locals and Temporaries
A Logical Decaf Stack Frame

Stack frame for function \( f(a, \ldots, n) \)
A Logical Decaf Stack Frame

Stack frame for function $f(a, ..., n)$

- Param $N$
- Param $N - 1$
-...
- Param 1
- Storage for Locals and Temporaries
  - Param $M$
  - ...
  - Param 1
A Logical Decaf Stack Frame

Stack frame for function \( f(a, ..., n) \)

- Param N
- Param \( N - 1 \)
- ...
- Param 1
- Storage for Locals and Temporaries
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}

_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;
Compiling Function Calls

void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}

_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
Compiling Function Calls

void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}
Compiling Function Calls

```c
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}

_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}

_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
Compiling Function Calls

```c
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}

void main() {
    SimpleFunction(137);
}
```

```assembly
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;

main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
```
Stack Management in TAC

- The **BeginFunc** $N$; instruction only needs to reserve room for local variables and temporaries.
- The **EndFunc**; instruction reclaims the room allocated with **BeginFunc** $N$;
- A single parameter is pushed onto the stack by the caller using the **PushParam** $\text{var}$ instruction.
- Space for parameters is reclaimed by the caller using the **PopParams** $N$; instruction.
  - $N$ is measured in *bytes*, not number of arguments.
## A Logical Decaf Stack Frame

A stack frame for function $f(a, ..., n)$

<table>
<thead>
<tr>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Param $N$</td>
</tr>
<tr>
<td>Param $N - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Param 1</td>
</tr>
<tr>
<td><strong>Storage for Locals and Temporaries</strong></td>
</tr>
</tbody>
</table>
A Logical Decaf Stack Frame

Stack frame for function \( f(a, \ldots, n) \)

<table>
<thead>
<tr>
<th>Param N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Param N - 1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Param 1</td>
</tr>
<tr>
<td>Storage for Locals and Temporaries</td>
</tr>
<tr>
<td>Param M</td>
</tr>
</tbody>
</table>

PushParam \( \text{var}; \)
A Logical Decaf Stack Frame

Stack frame for function \( f(a, \ldots, n) \)

- Param \( N \)
- Param \( N - 1 \)
- ...
- Param 1
- Storage for Locals and Temporaries
- Param \( M \)
- ...

PushParam \( \text{var}; \)
PushParam \( \text{var}; \)
A Logical Decaf Stack Frame

Stack frame for function f(a, ..., n)

<table>
<thead>
<tr>
<th>Param N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Param N - 1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Param 1</td>
</tr>
<tr>
<td>Storage for Locals and Temporaries</td>
</tr>
<tr>
<td>Param M</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Param 1</td>
</tr>
</tbody>
</table>

PushParam var;
PushParam var;
PushParam var;
A Logical Decaf Stack Frame

Stack frame for function \( f(a, ..., n) \)

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries
- Param M
- ...
- Param 1
- Storage for Locals and Temporaries

PushParam var;
PushParam var;
PushParam var;
BeginFunc N;
A Logical Decaf Stack Frame

Stack frame for function \( f(a, \ldots, n) \)

Stack frame for function \( g(a, \ldots, m) \)

- \( \text{BeginFunc } N; \)
- \( \text{PushParam } \text{var}; \)
- \( \text{PushParam } \text{var}; \)
- \( \text{PushParam } \text{var}; \)

- \( \text{Storage for Locals and Temporaries} \)

- \( \text{Param } N \)
- \( \text{Param } N - 1 \)
- \( \ldots \)
- \( \text{Param } 1 \)

- \( \text{Storage for Locals and Temporaries} \)

- \( \text{Param } M \)
- \( \ldots \)
- \( \text{Param } 1 \)
A Logical Decaf Stack Frame

Stack frame for function $f(a, \ldots, n)$
A Logical Decaf Stack Frame

Stack frame for function \( f(a, \ldots, n) \)

- Param \( N \)
- Param \( N - 1 \)
- ... (omitted)
- Param 1
- Storage for Locals and Temporaries
- Param \( M \)
- ... (omitted)
- Param 1
- Storage for Locals and Temporaries

EndFunc;
### A Logical Decaf Stack Frame

Stack frame for function $f(a, \ldots, n)$

<table>
<thead>
<tr>
<th>Param $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Param $N - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Param 1</td>
</tr>
<tr>
<td>Storage for Locals and Temporaries</td>
</tr>
<tr>
<td>Param $M$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Param 1</td>
</tr>
</tbody>
</table>
A Logical Decaf Stack Frame

Stack frame for function \( f(a, \ldots, n) \)

- Param \( N \)
- Param \( N - 1 \)
- ...;
- Param 1
- Storage for Locals and Temporaries
  - Param \( M \)
  - ...;
  - Param 1

PopParams \( N; \)
A Logical Decaf Stack Frame

Stack frame for function $f(a, \ldots, n)$

- Param $N$
- Param $N - 1$
- ...
- Param 1
- Storage for Locals and Temporaries
Storage Allocation

- As described so far, TAC does not specify where variables and temporaries are stored.
- For the final programming project, you will need to tell the code generator where each variable should be stored.
- This normally would be handled during code generation, but Just For Fun we thought you should have some experience handling this. 😊
The Frame Pointer

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries
The Frame Pointer

- Param N
- Param N - 1
- ...
- Param 1

Storage for Locals and Temporaries

Frame Pointer
The Frame Pointer

Param N
Param N - 1
...
Param 1
Storage for Locals and Temporaries
Param M
...
Param 1

Frame Pointer
The Frame Pointer

Frame Pointer

Param N
Param N - 1
...
Param 1
Storage for Locals and Temporaries
Param M
...
Param 1
Storage for Locals and Temporaries
The Frame Pointer

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries
- Param M
- ...
- Param 1
- Storage for Locals and Temporaries

Frame Pointer
The Frame Pointer

Frame Pointer

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries
- Param M
- ...
- Param 1
The Frame Pointer

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries

Frame Pointer
The Frame Pointer

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries

**Frame Pointer**
## Logical vs Physical Stack Frames

<table>
<thead>
<tr>
<th>Param N</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Param N - 1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Param 1</td>
<td></td>
</tr>
<tr>
<td><strong>Storage for Locals and Temporaries</strong></td>
<td></td>
</tr>
</tbody>
</table>
# Logical vs Physical Stack Frames

<table>
<thead>
<tr>
<th>Logical Stack Frame</th>
<th>Physical Stack Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Param N</td>
<td>Param N</td>
</tr>
<tr>
<td>Param N - 1</td>
<td>Param N - 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Param 1</td>
<td>Param 1</td>
</tr>
<tr>
<td>Storage for Locals</td>
<td>Storage for Locals</td>
</tr>
<tr>
<td>and Temporaries</td>
<td>and Temporaries</td>
</tr>
</tbody>
</table>

- fp of caller

<table>
<thead>
<tr>
<th>Logical Stack Frame</th>
<th>Physical Stack Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage for Locals</td>
<td>Storage for Locals</td>
</tr>
<tr>
<td>and Temporaries</td>
<td>and Temporaries</td>
</tr>
</tbody>
</table>
Logical vs Physical Stack Frames

<table>
<thead>
<tr>
<th>Logical Stack Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Param N</td>
</tr>
<tr>
<td>Param N - 1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Param 1</td>
</tr>
<tr>
<td>Storage for Locals and Temporaries</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Stack Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Param N</td>
</tr>
<tr>
<td>Param N - 1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Param 1</td>
</tr>
<tr>
<td>fp of caller</td>
</tr>
<tr>
<td>Storage for Locals and Temporaries</td>
</tr>
</tbody>
</table>

Frame Pointer
(Mostly) Physical Stack Frames

Frame Pointer

Param N
...
Param 1
\textit{fp of caller}
Storage for Locals and Temporaries
(Mostly) Physical Stack Frames

Frame Pointer

<table>
<thead>
<tr>
<th>fp of caller</th>
</tr>
</thead>
</table>

Storage for Locals and Temporaries

<table>
<thead>
<tr>
<th>Param N</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Param 1

Param N

Param 1
(Mostly) Physical Stack Frames

- Param N
- ...
- Param 1

Storage for Locals and Temporaries

- fp of caller
- Param N
- ...
- Param 1

fp of caller

Frame Pointer
(Mostly) Physical Stack Frames

Frame Pointer

- Param N
- ... 
- Param 1
- \textit{fp of caller}
- Storage for Locals and Temporaries
- Param N
- ... 
- Param 1
- \textit{fp of caller}
(Mostly) Physical Stack Frames

- Frame Pointer
  - fp of caller
  - Storage for Locals and Temporaries

- Param 1
  - fp of caller
  - Storage for Locals and Temporaries

- ...
(Mostly) Physical Stack Frames

Frame Pointer

fp of caller

Storage for Locals and Temporaries

Param N

...

Param 1

...

Param 1

fp of caller
(Mostly) Physical Stack Frames

Frame Pointer

(fp of caller)

Storage for Locals and Temporaries

(fp of caller)

Param 1

...

Param N

...

Param 1

...

Param N
(Mostly) Physical Stack Frames

Frame Pointer

Param N
...
Param 1

Storage for Locals and Temporaries

fp of caller

Param N
...
Param 1
(Mostly) Physical Stack Frames

- Param N
- ...
- Param 1
- \texttt{fp of caller}
- Storage for Locals and Temporaries

Frame Pointer
The Stored Return Address

- Internally, the processor has a special register called the **program counter** (PC) that stores the address of the next instruction to execute.
- Whenever a function returns, it needs to restore the PC so that the calling function resumes execution where it left off.
- The address of where to return is stored in MIPS in a special register called **ra** (“return address.”)
- To allow MIPS functions to call one another, each function needs to store the previous value of **ra** somewhere.
Physical Stack Frames

Frame Pointer

Param N
...
Param 1
fp of caller
ra of caller
Locals and Temporaries
Physical Stack Frames

- Param N
- ...
- Param 1
- fp of caller
- ra of caller
- Locals and Temporaries
- Param N
- ...
- Param 1

Frame Pointer
Physical Stack Frames

Frame Pointer

- fp of caller
- ra of caller
- Locals and Temporaries
- Param N
- ...
- Param 1
- ...
- Param N
- Param 1
- fp of caller
Physical Stack Frames

Param N
...
Param 1

.fp of caller
.ra of caller
Locals and Temporaries
Param N
...
Param 1

.fp of caller
.ra of caller

Frame Pointer
Physical Stack Frames

- Param N
- ...
- Param 1
- fp of caller
- ra of caller
- Locals and Temporaries
- Param N
- ...
- Param 1
- fp of caller
- ra of caller

Frame Pointer
Physical Stack Frames

- Param N
- ...
- Param 1
- fp of caller
- ra of caller
- Locals and Temporaries
- Param N
- ...
- Param 1
- fp of caller
- ra of caller
- Locals and Temporaries
So What?

- In your code generator, you must assign each local variable, parameter, and temporary variable its own location.
- These locations occur in a particular stack frame and are called **fp-relative**.
- Parameters begin at address **fp + 4** and grow upward.
- Locals and temporaries begin at address **fp - 8** and grow downward.
Location* location =
    new Location(fpRelative, +4, locName);
Location* location =
    new Location(fpRelative, +4, locName);
Location* location =
    new Location(fpRelative, +4, locName);
And One More Thing...

```c
int globalVariable;

int main() {
    globalVariable = 137;
}
```
And One More Thing...

```c
int globalVariable;

int main() {
    globalVariable = 137;
}
```
And One More Thing...

```c
int globalVariable;

int main() {
    globalVariable = 137;
}
```

Where is this stored?
The Global Pointer

- MIPS also has a register called the global pointer (gp) that points to globally accessible storage.
- Memory pointed at by the global pointer is treated as an array of values that grows upward.
- You must choose an offset into this array for each global variable.
From Your Perspective

Location* global =
    new Location(gpRelative, +8, locName);
From Your Perspective

Location* global =
    new Location(gpRelative, +8, locName);
Summary of Memory Layout

- Most details abstracted away by IR format.

- Remember:
  - Parameters start at \( fp + 4 \) and grow upward.
  - Locals start at \( fp - 8 \) and grow downward.
  - Globals start at \( gp + 0 \) and grow upward.

- You will need to write code to assign variables to these locations.
class A {
    void fn(int x) {
        int y;
        y = x;
    }
}

int main() {
    A a;
    a.fn(137);
}
class A {
    void fn(int x) {
        int y;
        y = x;
    }
}

int main() {
    A a;
    a.fn(137);
}
class A {
    void fn(int x) {
        int y;
        y = x;
    }
}

int main() {
    A a;
    a.fn(137);
}
A Reminder: Object Layout

- Vtable*
  - Field 0
  - ...
  - Field N

- Method 0
  - Method 1
  - ...
  - Method K

- Vtable*
  - Field 0
  - ...
  - Field N
  - Field 0
  - ...
  - Field M

- Method 0
  - Method 1
  - ...
  - Method K
  - Method 0
  - ...
  - Method L
class A {
    int y;
    int z;
    void fn(int x) {
        y = x;
        x = z;
    }
}

int main() {
    A a;
    a.fn(137);
}
TAC for Objects, Part II

class A {
    int y;
    int z;
    void fn(int x) {
        y = x;
        x = z;
    }
}

int main() {
    A a;
    a.fn(137);
}
class A {
    int y;
    int z;
    void fn(int x) {
        y = x;
        x = z;
    }
}

int main() {
    A a;
    a.fn(137);
}
TAC for Objects, Part II

class A {
    int y;
    int z;
    void fn(int x) {
        y = x;
        x = z;
    }
}

int main() {
    A a;
    a.fn(137);
}
Memory Access in TAC

• Extend our simple assignments with memory accesses:
  • \( \text{var}_1 = *\text{var}_2 \)
  • \( \text{var}_1 = *(\text{var}_2 + \text{constant}) \)
  • \( *\text{var}_1 = \text{var}_2 \)
  • \( *(\text{var}_1 + \text{constant}) = \text{var}_2 \)

• You will need to translate field accesses into relative memory accesses.
class Base {
    void hi() {
        Print("Base");
    }
}

class Derived extends Base{
    void hi() {
        Print("Derived");
    }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
class Base {
    void hi() {
        Print("Base");
    }
}

class Derived extends Base{
    void hi() {
        Print("Derived");
    }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
class Base {
    void hi() {
        Print("Base");
    }
}

class Derived extends Base{
    void hi() {
        Print("Derived");
    }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
class Base {
    void hi() {
        Print("Base");
    }
}

class Derived extends Base{
    void hi() {
        Print("Derived");
    }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
class Base {
    void hi() {
        Print("Base");
    }
}

class Derived extends Base{
    void hi() {
        Print("Derived");
    }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
class Base {
    void hi() {
        Print("Base");
    }
}

class Derived extends Base{
    void hi() {
        Print("Derived");
    }
}

int main() {
    Base b;
    b = new Derived;
    b.hi();
}
TAC for Objects, Part III

class Base {
  void hi() {
    Print("Base");
  }
}

class Derived extends Base{
  void hi() {
    Print("Derived");
  }
}

int main() {
  Base b;
  b = new Derived;
  b.hi();
}

main:
  BeginFunc 20;
  _t0 = 4;
  PushParam _t0;
  b = LCall _Alloc;
  PopParams 4;
  _t1 = Derived;
  *b = _t1;
  _t2 = *b;
  _t3 = *_t2;
  PushParam b;
  ACall _t3;
  PopParams 4;
  EndFunc;

What's going on here?
Dissecting TAC

```c
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = * _t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```
Dissecting TAC

int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *__t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*b = _t1;
_t2 = *b;
_t3 = *_t2;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
Dissecting TAC

int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*b = _t1;
_t2 = *b;
_t3 = *_t2;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
* _t2 = _t1;
* _t3 = * _t2;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*(_t2 = *b;
_t3 = *__t2;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
int main() {
  Base b;
  b = new Derived;
  b.hi();
}

main:
  BeginFunc 20;
  _t0 = 4;
  PushParam _t0;
  b = LCall _Alloc;
  PopParams 4;
  _t1 = Derived;
  *b = _t1;
  _t2 = *b;
  _t3 = *__t2;
  PushParam b;
  ACall _t3;
  PopParams 4;
  EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;  
b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *__t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
_b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*_{b} = _t1;
_{t2} = *_{b};
_{t3} = *_{t2};
PushParam _b;
ACall _t3;
ACall _t1;
P_CPP: 4
}
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
```c
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*_b = _t1;
_t2 = *b;
_t3 = *_t2;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*_{b} = _t1;
_t2 = *_{b};
_t3 = *_{t2};
PushParam _b;
ACall _t3;
PopParams 4;
EndFunc;
int main()
{
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
Dissecting TAC

```c
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*t2 = _t1;
_t3 = *t2;
PUSHParam b;
ACall _t3;
PopParams 4;
EndFunc;
```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
* _t2 = _t1;
_t3 = _t2;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
Dissecting TAC

int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*_{t0} = _t1;
_t2 = *_{t0};
_t3 = *_{t2};
ACall _t3;
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    PushParam b;
    ACall _t3;
    PopParams 4;
    EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
* _t2 = * _t3;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*_t2 = _t1;
PushParam b;
ACall _t3; 
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
* _t2 = _t1;
_t3 = * _t2;
ACall _t3;
PopParams 4;
EndFunc;
```c
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```
Dissecting TAC

```c
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
main:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t3 = *b;
    b = _t3;
    EndFunc;
```
Dissecting TAC

```c
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```c
main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
_b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*_{t1} = _t2 = *
_t2;
_t3 = _t3 = *
PopFunc;
EndFunc;
```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

begin:
    BeginFunc 20;
    _t0 = 4;
    PushParam _t0;
    b = LCall _Alloc;
    PopParams 4;
    _t1 = Derived;
    *b = _t1;
    _t2 = *b;
    _t3 = *_t2;
    ACall _t3;
    PopParams 4;
    EndFunc;
Dissecting TAC

int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*_b = _t1;
_t2 = *b;
_t3 = *__t2;
ACall _t3;
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
* _b = _t1;
_t2 = * _b;
_t3 = * _t2;
Load Function

VTable*
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*(_t2 = *b) = _t1;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*_b = _t1;
_t2 = *_b;
_t3 = *_t2;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
_*b = _t1;
_t2 = *b;
_t3 = *_*t2;
PushParam _t3;
ACall _t3;
PopParams 4;
EndFunc;
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

main:
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*_b = _t1;
_t2 = *b;
_t3 = *_t2;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
```cpp
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

```
BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
_pop3 = *_t2;
PramParam b;
ACall _t3;
PopParams 4;
EndFunc;
```
int main() {
    Base b;
    b = new Derived;
    b.hi();
}

int main:

BeginFunc 20;
_t0 = 4;
PushParam _t0;
b = LCall _Alloc;
PopParams 4;
_t1 = Derived;
*tb = _t1;
_t2 = *tb;
_t3 = **_t2;
PushParam _t3;
ACall _t3;
PopParams 4;
EndFunc;
OOP in TAC

- The address of an object's vtable can be referenced via the name assigned to the vtable (usually the object name).
  - e.g. \_t0 = Base;
- When creating objects, you must remember to set the object's vtable pointer or any method call will cause a crash at runtime.
- The `ACall` instruction can be used to call a method given a pointer to the first instruction.
Generating TAC
TAC Generation

• At this stage in compilation, we have
  • an AST,
  • annotated with scope information,
  • and annotated with type information.

• To generate TAC for the program, we do (yet another) recursive tree traversal!
  • Generate TAC for any subexpressions or substatements.
  • Using the result, generate TAC for the overall expression.
TAC Generation for Expressions

• Define a function \texttt{cgen}(expr) that generates TAC that computes an expression, stores it in a temporary variable, then hands back the name of that temporary.

• Define \texttt{cgen} directly for atomic expressions (constants, \texttt{this}, identifiers, etc.).

• Define \texttt{cgen} recursively for compound expressions (binary operators, function calls, etc.)
cgen for Basic Expressions
cgen for Basic Expressions

cgen(k) = { // k is a constant
    Choose a new temporary t
    Emit( t = k );
    Return t
}

**cgen for Basic Expressions**

\[
cgen(k) = \begin{cases} 
& // k \text{ is a constant} \\
& \text{Choose a new temporary } t \\
& \text{Emit}( t = k ); \\
& \text{Return } t 
\end{cases}
\]

\[
cgen(id) = \begin{cases} 
& // id \text{ is an identifier} \\
& \text{Choose a new temporary } t \\
& \text{Emit}( t = id ) \\
& \text{Return } t 
\end{cases}
\]
cgen for Binary Operators
cgen for Binary Operators

cgen(e_1 + e_2) = {

Choose a new temporary t
Let t_1 = cgen(e_1)
Let t_2 = cgen(e_2)
Emit( t = t_1 + t_2 )
Return t

}
An Example

cgen(5 + x) = {
    Choose a new temporary t
    Let t₁ = cgen(5)
    Let t₂ = cgen(x)
    Emit (t = t₁ + t₂)
    Return t
}

An Example

cgen(5 + x) = {
  Choose a new temporary t
  Let t₁ = {
    Choose a new temporary t
    Emit( t = 5 )
    return t
  }
  Let t₂ = cgen(x)
  Emit (t = t₁ + t₂)
  Return t
}
An Example

cgen(5 + x) = {
    Choose a new temporary $t$
    Let $t_1 =$
        Choose a new temporary $t$
        Emit($t = 5$)
        return $t$
    return $t$
}
Let $t_2 =$
    Choose a new temporary $t$
    Emit($t = x$)
    return $t$

Emit ($t = t_1 + t_2$)
Return $t$
}
An Example

cgen(5 + x) = {
    Choose a new temporary \( t \)
    Let \( t_1 = \) {
        Choose a new temporary \( t \)
        Emit( \( t = 5 \) )
        return \( t \)
    }

    Let \( t_2 = \) {
        Choose a new temporary \( t \)
        Emit( \( t = x \) )
        return \( t \)
    }

    Emit \( (t = t_1 + t_2) \)
    Return \( t \)
}

\_t0 = 5
\_t1 = x
\_t2 = \_t0 + \_t1
cgen for Statements

- We can extend the cgen function to operate over statements as well.
- Unlike cgen for expressions, cgen for statements does not return the name of a temporary holding a value.
  - (Why?)
cgen for Simple Statements
cgen for Simple Statements

cgen(expr;) = {
    cgen(expr)
}
cgen for while loops
cgen for while loops

cgen(while (expr) stmt) = {

Let L before be a new label.
Let L after be a new label.
Emit( L before:)
Let t = cgen(expr)
Emit(IfZ t Goto L after)
cgen(stmt)
Emit(Goto L before)
Emit( L after:)

}
cgen for while loops

cgen(while (expr) stmt) = {
    Let L_{before} be a new label.
    Let L_{after} be a new label.
}
cgen for while loops

cgen(while (expr) stmt) = {
    Let $L_{before}$ be a new label.
    Let $L_{after}$ be a new label.
    Emit( $L_{before}$ : )
    Emit( $L_{after}$ : )
}

cgen for while loops

cgen(while (expr) stmt) = {
  Let L_{before} be a new label.
  Let L_{after} be a new label.
  Emit( L_{before} : )
  Let t = cgen(expr)
  Emit( IfZ t Goto L_{after} )
  Emit( L_{after} : )
}

cgen for while loops

cgen(while  (expr) stmt) = {
    Let L_{before} be a new label.
    Let L_{after} be a new label.
    Emit( L_{before} : )
    Let t = cgen(expr)
    Emit( IfZ t Goto L_{after} )
    cgen(stmt)

    Emit( L_{after} : )
}

cgen for while loops

cgen(while (expr) stmt) = {
    Let $L_{before}$ be a new label.
    Let $L_{after}$ be a new label.
    Emit($L_{before}$ :)
    Let $t = \text{cgen}(expr)$
    Emit( IfZ $t$ Goto $L_{after}$ )
    cgen(stmt)
    Emit( Goto $L_{before}$ )
    Emit( $L_{after}$ : )
}


Next Time

• **Intro to IR Optimization**
  • Basic Blocks
  • Control-Flow Graphs
  • Local Optimizations