Three-Address Code IR
Where We Are

Source Code

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code
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”

• 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;
```
Sample TAC Code

int x;
int y;

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

x2 = x * x;
y2 = y * y;
r2 = x2 + y2;
int a;
int b;
int c;
int d;

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

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

_t0 = 5;
_t1 = 2 * b;
a = _t0 + _t1;

TAC allows for instructions with two operands.
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;`?
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;`?
  - `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

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

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

```c
_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 <= 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 \( b = (x \leq y) \) ?

\[
\begin{align*}
_t0 &= x < y; \\
_t1 &= x == y; \\
b &= _t0 \text{ || } _t1;
\end{align*}
\]
Control Flow Statements

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;
```

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

_L0:
    z = y;
_L1:
    z = z * z;
```
int x;
int y;
int z;

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

z = z * z;

_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;
```

```c
_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

```plaintext
int x;
int y;

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

y = x;
```
Control Flow Statements

```
int x;
int y;

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

y = x;
```

```
_L0:
_t0 = x < y;
IfZ _t0 Goto _L1;
x = x * 2;
Goto _L0;

_L1:
y = x;
```
A Logical Call Stack

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

- Param N
- Param N - 1
- ...
- Param 1
- Storage for Locals and Temporaries
A Logical Call Stack

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

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

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

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

[Diagram of a call stack with labels for parameters and storage areas.]
A Logical Call Stack

Stack frame for function f(a, ..., n)
Stack frame for function $f(a, \ldots, n)$
A Logical Call Stack

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

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

- Param $N$
- Param $N - 1$
- ...
- Param 1
  - Storage for Locals and Temporaries
- Param $M$
- ...
- Param 1
  - Storage for Locals and Temporaries
Stack frame for function \( f(a, \ldots, n) \)

A Logical Call Stack

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

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

- 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() {
    SimpleFn(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;
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);
}
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;
```
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;
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;
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 var` instruction.
- Space for parameters is reclaimed by the caller using the `PopParams N;` instruction.
  - `N` is measured in bytes, not number of arguments.
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>Storage for Locals and Temporaries</td>
</tr>
</tbody>
</table>
Stack frame for function $f(a, \ldots, n)$

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

PushParam var;
Param N
Param N - 1
...
Param 1
Storage for Locals and Temporaries
Param M
...

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

PushParam var;
PushParam var;
Stack frame for function f(a, ..., n)

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

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

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

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

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

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

PushParam \( var \);
PushParam \( var \);
PushParam \( var \);
BeginFunc \( N \);
Stack frame for function $f(a, \ldots, n)$
Stack frame for function \( f(a, \ldots, n) \)

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

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

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

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

- 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
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
# The Frame Pointer

<table>
<thead>
<tr>
<th>Frame Pointer</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>
<tr>
<td>✔️ Param M</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>✔️ Param 1</td>
</tr>
</tbody>
</table>
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>
</tr>
</thead>
<tbody>
<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>
Logical vs Physical Stack Frames

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

| Param N | Param N - 1 | ... | Param 1 | fp of caller | Storage for Locals and Temporaries |
Logical vs Physical Stack Frames

Frame Pointer

Storage for Locals and Temporaries

Param N
Param N - 1
...
Param 1

fp of caller

Storage for Locals and Temporaries

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

- Frame Pointer
- Frame Pointer of caller
- Storage for Locals and Temporaries
- Param 1
- ...
(Mostly) Physical Stack Frames

![Diagram showing Frame Pointer and stack frame structure]

- Param N
- ...
- Param 1
- **fp of caller**
- Storage for Locals and Temporaries
- Param N
- ...
- Param 1

Frame Pointer
(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

"fp of caller"

Storage for Locals and Temporaries

Param N
...
Param 1

"fp of caller"
(Mostly) Physical Stack Frames

Frame Pointer

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

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

- Frame Pointer
- fp of caller
- Storage for Locals and Temporaries
  - fp of caller
  - Param 1
    - ...
(Mostly) Physical Stack Frames

- Param N
- ...
- Param 1

Storage for Locals and Temporaries

- fp of caller

Frame Pointer

- Param N
- ...
- Param 1
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

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

Frame Pointer
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

- Param N
- ...
- Param 1
- \textit{fp of caller}
- \textit{ra of caller}
- Locals and Temporaries
- Param N
- ...
- Param 1
- \textit{fp of caller}

Frame Pointer
Physical Stack Frames

```
Frame Pointer

param N
...
param 1

fp of caller
ra of caller
locals and temporaries
param N
...
param 1
fp of caller
ra of caller
```
Physical Stack Frames

Frame Pointer

Param N
...
Param 1

fp of caller
ra of caller

Locals and Temporaries

Param N
...
Param 1

fp of caller
ra of caller
Physical Stack Frames

<table>
<thead>
<tr>
<th>Frame Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
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.
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?
• 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.
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.
TAC for Objects, Part I

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
- Method 0
- Method 1
  - ...
- Method K
- Method L
- Field 0
  - ...
- Field M
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);
}

_A.fn:
    BeginFunc 4;
    *(this + 4) = x;
    x = *(this + 8);
    EndFunc;

main:
    BeginFunc 8;
    _t0 = 137;
    PushParam _t0;
    PushParam a;
    LCall _A.fn;
    PopParams 8;
    EndFunc;
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);
}
Memory Access in TAC

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

- 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();
}
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();
}
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();
}
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;
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();
}
```

```c
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;
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;
* _t1 = _t0;
* _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;
*_{b} = _t1;
_t2 = *_{b};
_t3 = *_{t2};
PushParam \text{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);
*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;
_t2 = *b;
_t3 = *__t2;
PushParam b;
ACall _t3;
PopParams 4;
EndFunc;
```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;
*t_b = _t1;
_t2 = *t_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;
```c
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```

### Code for Derived.hi

<table>
<thead>
<tr>
<th>fp of caller</th>
<th>ra of caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>_t0</td>
</tr>
<tr>
<td>_t1</td>
<td>_t2</td>
</tr>
<tr>
<td>_t3</td>
<td>b</td>
</tr>
</tbody>
</table>

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

(fp of caller)

(ra of caller)
Allocate Object

(raw memory)
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;

<table>
<thead>
<tr>
<th>fp of caller</th>
<th>ra of caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>_t0</td>
</tr>
<tr>
<td></td>
<td>_t1</td>
</tr>
<tr>
<td></td>
<td>_t2</td>
</tr>
<tr>
<td></td>
<td>_t3</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
</tbody>
</table>

Allocate Object

(raw memory)

Dissecting TAC
```plaintext
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

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

```
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;
*b = _t1;
_t2 = _t2;
_t3 = _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;
*_{t0} = _t1;
_t2 = *b;
_t3 = *_{t2};
ACall _t3;
PopParams 4;
EndFunc;

Derived Vtable

Code for Derived.hi

fp of caller
ra of caller
4
_t0
_t1
_t2
_t3
b

VTable*

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;
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();
}
```
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 = *_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;
_*b = _t1;
_t2 = *b;
_*t2 = *_*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};
ACall _t3;
PushParam _b;
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;
```c
int main() {
    Base b;
    b = new Derived;
    b.hi();
}
```
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. 

- 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 `cgen(expr)` that generates TAC that computes an expression, stores it in a temporary variable, then hands back the name of that temporary.
- Define `cgen` directly for atomic expressions (constants, `this`, identifiers, etc.).
- Define `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

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

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

cgen(e₁ + e₂) = {
    Choose a new temporary \( t \)
    Let \( t₁ = \text{cgen}(e₁) \)
    Let \( t₂ = \text{cgen}(e₂) \)
    Emit( \( t = t₁ + t₂ \) )
    Return \( t \)
}
An Example

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

\texttt{cgen}(5 + x) = \{ 
  \text{Choose a new temporary } t \\
  \text{Let } t_1 = \{ \\
    \text{Choose a new temporary } t \\
    \text{Emit}( t = 5 ) \\
    \text{return } t \\
  \} \\
  \text{Let } t_2 = \{ \\
    \text{Choose a new temporary } t \\
    \text{Emit}( t = x ) \\
    \text{return } t \\
  \} \\
  \text{Emit} (t = t_1 + t_2) \\
  \text{Return } t \\
\}
An Example

\texttt{cgen}(5 + x) = \{ \\
\quad \text{Choose a new temporary } t \\
\quad \text{Let } t_1 = \{ \\
\quad\quad \text{Choose a new temporary } t \\
\quad\quad \text{Emit}( t = 5 ) \\
\quad\quad \text{return } t \\
\quad \} \\
\quad \text{Let } t_2 = \{ \\
\quad\quad \text{Choose a new temporary } t \\
\quad\quad \text{Emit}( t = x ) \\
\quad\quad \text{return } t \\
\quad \} \\
\quad \text{Emit} \ ( t = t_1 + t_2 ) \\
\quad \text{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

\[
cgen(expr;) = \{ \\
\quad cgen(expr) \\
\}
\]
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 $L_{\text{before}}$ be a new label.
  Let $L_{\text{after}}$ be a new label.
  Emit( $L_{\text{before}}$ : )

  Emit( $L_{\text{after}}$ : )
}

**cgen for while loops**

\[ \text{cgen(while (expr) stmt) = \{ \} } \]

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

Emit( \( L_{after} : \) )
**cgen for while loops**

\[ \text{cgen(while (expr) stmt)} = \{ \]

Let \( L_{\text{before}} \) be a new label.

Let \( L_{\text{after}} \) be a new label.

Emit( \( L_{\text{before}} : \) )

Let \( t = \text{cgen(expr)} \)

Emit( \text{IfZ} t \text{ Goto } L_{\text{after}} )

\[ \text{cgen(stmt)} \]

Emit( \( L_{\text{after}} : \) )

\}
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( Goto L_{before} )
    Emit( L_{after} : )
}

Next Time

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