Compilation in a Nutshell

Abstract Syntax Tree:

```
If
  Eq
    b
  Assn
    0
    1
  None
```

Intermediate code:

```
11:
  %cnd = icmp eq i64 %b, 0
  br i1 %cnd, label %l2, label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:
```

Assembly Code

```
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  ...
```
label:
Move r1 3
Move r2 5
Cjump LT r1 0 ltrue

lfalse:
Move r3 0
Jump l2

ltrue:
Move r3 1

• What’s left to translate?
Code Generation

- Individual statements are still IR trees!

Source: \(a[i] := x\)

Block IR:

```
Move

Mem

Plus

Mem

Plus

Mem

Plus

fp

1

Mem

i

fp

2
```

Assembly:

```
r1 = fp + 1
r2 = [r1]
r3 = r2 + i
r4 = fp + 2
r5 = [r4]
[r3] = r5
```
Individual statements are still IR trees!

Source: \( a[i] := x \)

Block IR:

```
Move
  Mem
  Plus
    Mem
    i
  Mem
  Plus
    fp
    2
Plus
  fp
  1
```

Assembly:

```
r1 = fp + 1
r2 = [r1]
r3 = r2 + i
r4 = fp + 2
r5 = [r4]
[r3] = r5
```
Individual statements are still IR trees!

Source: 

\[
a[i] := x
\]

Block IR:

```
Move

Mem

Plus

Mem

i

fp

2

Plus

fp

1
```

Assembly:

```
r1 = fp + 1
r2 = [r1]
r3 = r2 + i
r4 = fp + 2
r5 = [r4]
[r3] = r5
```
Code Generation

- Individual statements are still IR trees!

**Source:**

\[ a[i] := x \]

**Assembly:**

\[ r1 = fp + 1 \]

\[ r2 = [r1] \]

\[ r3 = r2 + i \]

\[ r4 = fp + 2 \]

\[ r5 = [r4] \]

\[ [r3] = r5 \]
• Individual statements are still IR trees!

Source: $a[i] := x$

Block IR:

```
Move
    Mem
        Plus
            Mem
                i
            fp
        1
    Mem
        Plus
            Mem
                fp
            2
```

Assembly:

```
\text{r1} = \text{fp} + 1

\text{r2} = \text{[r1]}

\text{r3} = \text{r2} + i

\text{r4} = \text{fp} + 2

\text{r5} = \text{[r4]}
\text{[r3]} = \text{r5}
```
Code Generation

- Individual statements are still IR trees!

Source: \(a[i] := x\)

Block IR:

```
Move
  Mem
    Plus
      Mem
        i
      Mem
    fp
  Plus
    Mem
      fp
    2
```

Assembly:

- \(r1 = [fp + 1]\)
- \(r2 = r1 + i\)
- \([r2] = [fp + 2]\)
Instruction Selection: Tiling

- Individual statements are still IR trees!
- The process of choosing assembly instructions for each piece of the IR is called *instruction selection*
- We can do it by *tiling* the tree with segments that represent instructions in the target language

Assembly:

\[
\begin{align*}
    r1 &= [fp + 1] \\
    r2 &= r1 + i \\
    [r2] &= [fp + 2]
\end{align*}
\]
Instruction Selection: Tiling

- Inputs: a statement IR tree and a set of tiles for the target language

Instruction:

\[ r_1 = r_2 + r_3 \]

\[ r_1 = r_2 + c \]

\[ 3 + a \]

\[ r := 3 \]

\[ r_1 = [r_2] \]

\[ [r_1] = r_2 \]
Questions

Top
Instruction Selection: Tiling

- Inputs: a statement IR tree and a set of tiles for the target language

**Instruction:**

- \( r1 = r2 + r3 \)
- \( r1 = r2 + c \)
- \( r1 = [r2] \)
- \( [r1] = r2 \)

**Tile:**

- Plus
- Const
- Mem
- Move
Instruction Selection: Tiling

• Inputs: a statement IR tree and a set of tiles for the target language

Instruction:

$$[r1] = [r2 + c]$$

$$a[i] := b[3]$$

$$a[i] := b$$

$$a[i] := 3$$

Tile:
Instruction Selection: RISC vs. CISC

- Inputs: a statement IR tree and a set of tiles for the target language
- RISC (Reduced Instruction Set Computer) machines have a few simple instructions with small tiles, while CISC (Complex Instruction Set Computer) machines have many complicated instructions with large tiles
- Most newer processors are RISC, but Intel (x86) are CISC
- RISC is simple, easy to implement, easy to get right, while CISC allows lots of optimization for special cases
- Most compilers will target at least one of each! (e.g. x86 and ARM)
- Fortunately, we can use the same basic approach to tiling for both!
Even given a tree and a set of tiles, there might be multiple possible tilings!

**Assembly:**

\[
\begin{align*}
    r1 &= [fp + 1] \\
    r2 &= r1 + i \\
    [r2] &= [fp + 2]
\end{align*}
\]
Even given a tree and a set of tiles, there might be multiple possible tilings!

```
Move
  Mem
    Plus
      Mem
        i
    fp
  fp
```

Assembly:

\[
\begin{align*}
  r1 &= fp + 1 \\
  r2 &= [r1] + i \\
  [r2] &= [fp + 2]
\end{align*}
\]
Even given a tree and a set of tiles, there might be multiple possible tilings!

Assembly:

\[ r_1 = \text{fp} + 0 \]
\[ r_2 = 1 + 0 \]
\[ r_3 = r_1 + r_2 \]
\[ \ldots \]
Instruction Selection: Tiling

• Given a tree and a set of tiles, there might be multiple possible tilings
• There always exists at least one tiling, with one tile per node (so we’ll never get stuck)
• Which tiling do we actually want?
• Probably one with the fewest/cheapest instructions.
  – We can assign a cost (time, space, etc.) to each instruction, and use it to compute the cost of each tile
• Optimum tiling: the cheapest possible tiling for a given tree
• Optimal tiling: can’t be made cheaper with a local change
Questions

Top
Instruction Selection: Maximal Munch

- Inputs: a statement IR tree and a set of tiles for the target language
- Output: a tiling that is optimal in the number of tiles
- A classic greedy algorithm: starting from the root, choose the biggest tile for the current node

\[
\begin{align*}
r1 &= r3 + 3 \\
[r1] &= r2
\end{align*}
\]
Instruction Selection: Maximal Munch

• Inputs: a statement IR tree and a set of tiles for the target language
• Output: a tiling that is optimal in the number of tiles
• A classic greedy algorithm: starting from the root, choose the biggest tile for the current node

\[
\begin{align*}
\text{Plus} & \quad \text{Move} \\
\text{Mem} & \quad \text{Mem} \\
\text{Plus} & \quad \text{Plus} \\
\text{Mem} & \quad \text{fp} \\
\text{fp} & \quad 2 \\
3 & \\
\text{Plus} & \quad \text{Mem} \\
\text{fp} & \quad 1 \\
\text{fp} & \\
\end{align*}
\]

\[
\begin{align*}
r5 &= fp + 2 \\
r2 &= [r5] \\
r4 &= fp + 1 \\
r3 &= [r4] \\
r1 &= r3 + 3 \\
[r1] &= r2
\end{align*}
\]
Instruction Selection: Maximal Munch

- Inputs: a statement IR tree and a set of tiles for the target language
- Output: a tiling that is optimal in the number of tiles
- A classic greedy algorithm: starting from the root, choose the biggest tile for the current node
Instruction Selection: Maximal Munch

- Implementation: go through all the patterns until we find one that fits

```c
void munch(T_stm s){
    switch(s->kind){
        case move:
            if(s->left->kind == mem){
                munch(s->left->child);
                emit([r1] <- r2);
            }
            else emit(r1 <- r2);
        case plus: ...
    }
}
```
Instruction Selection: Maximal Munch

• Implementation: go through all the patterns until we find one that fits

```c
void munch(T_stm s) {
  switch (s->kind) {
    case move:
      switch (s->left->kind) {
        case mem:
          switch (s->right->kind) {
            case plus: ... 
            case const: ...
          }
      }
  }
}
```
Instruction Selection: Tiling

• Given a tree and a set of tiles, there might be multiple possible tilings.

• There always exists at least one tiling, with one tile per node (so we’ll never get stuck).

• Which tiling do we actually want?

• Probably one with the fewest/cheapest instructions.
  – We can assign a cost (time, space, etc.) to each instruction, and use it to compute the cost of each tile.

• *Optimum* tiling: the cheapest possible tiling for a given tree.

• *Optimal* tiling: can’t be made cheaper with a local change.

• Especially for CISC targets, optimal tilings may be significantly more expensive than optimum ones.
• We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.
We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.
We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.
We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.

![Diagram of tree structure with computations](image)

- best tile for this node + 1 + 2 = 4
Instruction Selection: Optimum Tiling

- We can compute an optimum tiling using dynamic programming: working from the bottom up, compute the optimum tiling for each subtree.

```
best tile for this node + 1 + 2 = 4
best left 2-node tile + 2 = 3
```
We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.

- best tile for this node + 1 + 2 = 4
- best left 2-node tile + 2 = 3
- best right 2-node tile + 1 = 2
We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.

- best tile for this node + 1 + 2 = 4
- best left 2-node tile + 2 = 3
- best right 2-node tile + 1 = 2
- best 3-node tile = 3
Instruction Selection: Optimum Tiling

- We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.

\[
\begin{align*}
\text{best tile for this node} & + 1 + 2 = 4 \\
\text{best left 2-node tile} & + 2 = 3 \\
\text{best right 2-node tile} & + 1 = 2 \\
\text{best 3-node tile} & = 3
\end{align*}
\]

\[
\text{cost of using a tile at this node} = \text{cost of the tile} + \text{best cost for the roots of the uncovered subtrees}
\]
We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.

\[
\text{fancy tile} + 1 + 2 + 1 = X
\]

Cost of using a tile at this node = cost of the tile + best cost for the roots of the uncovered subtrees.
We can compute an optimum tiling using dynamic programming: working from the bottom up, compute the optimum tiling for each subtree.

- fancy tile + 1 + 2 + 1 = X
- simpler tile + 2 + 1 = Y

Cost of using a tile at this node = cost of the tile + best cost for the roots of the uncovered subtrees.
Instruction Selection: Optimum Tiling

- We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.

  cost of using a tile at this node = cost of the tile + best cost for the roots of the uncovered subtrees

Once we choose the best tile for the root, put that instruction at the end of the list, then recursively emit instructions for the roots of the uncovered subtrees.
Instruction Selection: Optimum Tiling

- We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.

Once we choose the best tile for the root, put that instruction at the end of the list, then recursively emit instructions for the roots of the uncovered subtrees.

```
instr1
```

10

6 3

5 2

2 1 1 1

2 1 2
Instruction Selection: Optimum Tiling

- We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree.

Once we choose the best tile for the root, put that instruction at the end of the list, then recursively emit instructions for the roots of the uncovered subtrees.

```
instr3
instr2
instr1
```
Instruction Selection: Optimum Tiling

• We can compute an optimum tiling using *dynamic programming*: working from the bottom up, compute the optimum tiling for each subtree

Once we choose the best tile for the root, put that instruction at the end of the list, then recursively emit instructions for the roots of the uncovered subtrees

instr5
instr4
instr3
instr2
instr1
Questions
Instruction Selection: Summary

• Input: list of IR trees, one for each statement
• Output: list of assembly instructions (with infinite registers)
• Translate trees into instructions by tiling each tree with corresponding tiles
• Each target language has different sizes and shapes of tiles
• We can easily compute optimal tilings, and optimum ones take just a little more effort
Instruction Selection: Summary

- Input: list of IR trees, one for each statement
- Output: list of assembly instructions *(with infinite registers)*
- Translate trees into instructions by tiling each tree with corresponding tiles
- Each target language has different sizes and shapes of tiles
- We can easily compute optimal tilings, and optimum ones take just a little more effort

- We’ll still need to do register allocation to get real assembly
Representing Abstract Assembly

• Output: list of assembly instructions (with infinite registers)
• Essentially a very low-level, machine-specific IR

Real instruction: Effect:

add $3, $1, $2 $3 = $1 + $2
addi $5, $4, 77 $5 = $4 + 77
sw $6, 16($7) [$7 + 16] = $6
j L1 jump L1

Representing Abstract Assembly

- Output: list of assembly instructions **(with infinite registers)**
- Essentially a very low-level, machine-specific IR

<table>
<thead>
<tr>
<th>Real instruction:</th>
<th>Effect:</th>
<th>Abstract instruction:</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $3, $1, $2</td>
<td>$3 = $1 + $2</td>
<td>add <code>d0, </code>s0, `s1</td>
</tr>
<tr>
<td>addi $5, $4, 77</td>
<td>$5 = $4 + 77</td>
<td>add <code>d0, </code>s0, 22</td>
</tr>
<tr>
<td>sw $6, 16($7)</td>
<td>[$7 + 16] = $6</td>
<td>sw <code>s0, 16(</code>d0)</td>
</tr>
<tr>
<td>j L1</td>
<td>jump L1</td>
<td>j `j0</td>
</tr>
</tbody>
</table>

Abstract Assembly in Tiger

• In Tiger, this is implemented by:

\[
\text{AS\_instr AS\_Oper(string a, Temp\_tempList d,}
\]
\[
\text{Temp\_tempList s, AS\_targets j);}
\]

• Or AS\_Move if it’s a move that does no computation

\[
\text{add t4, t6, t9 becomes}
\]
\[
\text{AS\_Oper("add `d0, `s0, `s1",}
\]
\[
\text{<list containing t4>, <list containing t6 and t9>, NULL)}
\]

• assem.c implements a string processor that replaces `d0, `s0, etc. with values from the lists
void munch(T_stm s) {
    switch(s->kind) {
        case move:
            if(s->left->kind == mem) {
                munch(s->left->child);
                emit([r1] <- r2);
            }
            ...
        }
    }
}
void munch(T_stm s) {
    switch(s->kind) {
        case move:
            if(s->left->kind == mem) {
                munch(s->left->child);
                emit(AS_Oper("sw `s0, 0 (`d0 )"),
                    L(r2, NULL), L(r1, NULL), NULL);
            }
        ...
    }
}
Questions