THEORY OF COMPILATION

LECTURE 07

CODE GENERATION
From IR to ASM: Challenges

- Register allocation
- Mapping IR to ASM operations
  - what instruction(s) should be used to implement an IR operation?
  - how do we translate code sequences?
- Memory allocation
- Call/return of routines
  - managing activation records
- Architecture-specific optimizations
From IR to ASM: Challenges

• Register allocation
• Mapping IR to ASM operations
  – what instruction(s) should be used to implement an IR operation?
  – how do we translate code sequences?
• Memory allocation
• Call/return of routines
  – managing activation records
• Architecture-specific optimizations
From IR to ASM: Challenges

- Register allocation
- Mapping IR to ASM operations
  - what instruction(s) should be used to implement an IR operation?
  - how do we translate code sequences?
- Memory allocation
- Call/return of routines
  - managing activation records
- Architecture-specific optimizations
From IR to ASM: Challenges

• Register allocation
• Mapping IR to ASM operations
  – what instruction(s) should be used to implement an IR operation?
  – how do we translate code sequences?
• Memory allocation
• Call/return of routines
  – managing activation records
• Architecture-specific optimizations

this lecture
≈ week 10

way too complicated
Control Flow Graph

• Break the IR into **basic blocks**

• A **basic block** is a sequence of instructions with
  – **single entry** (to first instruction), no jumps to the middle of the block
  – **single exit** (last instruction)
  – code executes as a sequence from first instruction to last instruction without any jumps

• Edge from one basic block B1 to another block B2 when the last statement of B1 may jump to B2
Control Flow Graph: Example

B1
\[ t_1 := 4 * i \]
\[ t_2 := a[i] \]
if \( t_2 \leq 20 \) goto B3

B2
\[ t_3 := 4 * i \]
\[ t_4 := b[i] \]
goto B4

False

B3
\[ t_5 := t_2 \times t_4 \]
\[ t_6 := \text{prod} + t_5 \]
\[ \text{prod} := t_6 \]
goto B4

True

B4
\[ t_7 := i + 1 \]
i := t_2
goto B5
Control Flow Graph: Definition

- A directed graph $G = \langle V, E \rangle$
  - $V =$ basic blocks
  - $E =$ control flow
    - $(B_1, B_2) \in E$ when control from $B_1$ flows to $B_2$

```
t_1 := 4 \times i
i := 1
i := t_7
if i <= 20 goto B_2
```
Creating Basic Blocks

• **Input**: A sequence of three-address statements
• **Output**: A list of basic blocks with each three-address statement in exactly one block

• **Method**
  – Determine the set of **leaders** (first statement of a block)
    • The first statement is a leader
    • Any statement that is the target of a jump is a leader
    • Any statement that immediately follows a jump is a leader
  – For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program
Example

for i from 1 to 10 do
  for j from 1 to 10 do
    a[i, j] = 0.0;
  end for
  a[i, i] = 1.0;
end for

IR

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

CFG
Example

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i, j] = 0.0;
    end for
    a[i, i] = 1.0;
end for
```

source

IR

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 – 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i – 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

CFG

for i from 1 to 10 do
  for j from 1 to 10 do
    a[i, j] = 0.0;
  end for
end for

for i from 1 to 10 do
  a[i, i] = 1.0;
end for
Example

source

for i from 1 to 10 do
  for j from 1 to 10 do
    a[i, j] = 0.0;
  end for
end for

for i from 1 to 10 do
  a[i, i] = 1.0;
end for

IR

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 – 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i – 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

CFG

Example

for i from 1 to 10 do
  for j from 1 to 10 do
    a[i, j] = 0.0;
  end for
end for

for i from 1 to 10 do
  a[i, i] = 1.0;
end for

source

IR

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 – 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i – 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

CFG
Example

source

```plaintext
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i, j] = 0.0;
    for i from 1 to 10 do
        a[i, i] = 1.0;
```

IR

1) \( i = 1 \)
2) \( j = 1 \)
3) \( t1 = 10 \times i \)
4) \( t2 = t1 + j \)
5) \( t3 = 8 \times t2 \)
6) \( t4 = t3 - 88 \)
7) \( a[t4] = 0.0 \)
8) \( j = j + 1 \)
9) if \( j \leq 10 \) goto (3)
10) \( i = i + 1 \)
11) if \( i \leq 10 \) goto (2)
12) \( i = 1 \)
13) \( t5 = i - 1 \)
14) \( t6 = 88 \times t5 \)
15) \( a[t6] = 1.0 \)
16) \( i = i + 1 \)
17) if \( i \leq 10 \) goto (13)

CFG
Example

source

for i from 1 to 10 do
    for j from 1 to 10 do
        a[i, j] = 0.0;
    end
end

for i from 1 to 10 do
    a[i, i] = 1.0;
end

IR

1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

CFG
Example

Source

```c
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i, j] = 0.0;
    end for
    a[i, i] = 1.0;
end for
```

IR

```c
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

CFG

```
B1  | i = 1
B2  | j = 1
B3  | t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
a[t4] = 0.0
j = j + 1
if j <= 10 goto B3
B4  | i = i + 1
if i <= 10 goto B2
B5  | i = 1
B6  | t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B6
```
Example

for i from 1 to 10 do
    for j from 1 to 10 do
        a[i, j] = 0.0;
    for i from 1 to 10 do
        a[i, i] = 1.0;
Simple Code Generation

• Registers
  – used as operands of instructions
  – can be used to store temporary results
  – can (should) be used as loop indexes due to frequent reads and increments
  – used to manage administrative info (e.g., runtime stack)

• Number of registers is limited
  ⇒ Need to allocate them in a clever way
Simple Code Generation

• Assume machine instructions of the form
  ‣ LD reg, mem
  ‣ ST mem, reg
  ‣ OP reg, reg, reg

• Assume that we have all registers available for our use
  – Ignore registers allocated for stack management
  – Treat all registers as general-purpose
Simple Code Generation

- Translate each 3AC instruction separately

- For each register, a **register descriptor** records the variable name(s) whose current value is in that register
  - we use only those registers that are available for local use within a basic block; we assume that initially, all register descriptors are empty
  - As code generation progresses, **each register will hold the value of zero or more variables**

- For each program variable, an **address descriptor** records the location(s) where the current value of the variable can be found
  - The location may be a register, a memory address, a stack offset, or some set of these
  - Information can be stored in the symbol-table entry for that variable
Simple Code Generation

For each three-address statement $x := y \text{ op } z,$

1. Invoke $\textit{getreg}(x := y \text{ op } z)$ to select registers $R_x, R_y,$ and $R_z$
2. If $R_y$ does not contain $y,$ issue: $\text{LD } R_y, y'$ for a location $y'$ of $y$
3. If $R_z$ does not contain $z,$ issue: $\text{LD } R_z, z'$ for a location $z'$ of $z$
4. Issue the instruction $\text{OP } R_x, R_y, R_z$

The function $\textit{getreg}$ is not defined yet — for now think of it as an oracle that gives us a register per variable
Updating the Descriptors

1. For the instruction **LD  R, x**
   a) Change register descriptor for R so it holds **only** x
   b) Change address descriptor for x by **adding** R as an additional location

2. For the instruction **ST  x, R**
   a) change the address descriptor for x to include its own memory location

3. For an operation such **OP  R_x, R_y, R_z**, originating form a 3AC instruction
   x = y op z
   a) Change register descriptor for R_x so that it holds **only** x
   b) Change address descriptor for x so that its **only** location is R_x. Note that the memory location for x is **not** now in the address descriptor for x
   c) Remove R_x from the address descriptor of any variable other than x

4. When we process a **copy statement x  =  y**, 
   a) Generate load for y into register R_y and update descriptors accordingly (rule 1)
   b) Add x to the register descriptor for R_y.
   c) Change the address descriptor for x so that its only location is R_y.
Example

t = A – B
u = A – C
v = t + u
A = D
D = v + u

A B C D = variables in function scope
t u v = temporaries in block
Example

t = A – B
u = A – C
v = t + u
A = D
D = v + u

A B C D = variables in function scope
t u v = temporaries in block
Example

\[
t = A - B \\
u = A - C \\
v = t + u \\
A = D \\
D = v + u
\]

A B C D = variables in function scope

t u v = temporaries in block
Example

\[ t = A - B \]
\[ u = A - C \]
\[ v = t + u \]
\[ A = D \]
\[ D = v + u \]

A B C D = variables in function scope

15
Example

\[ t = A - B \]
\[ u = A - C \]
\[ v = t + u \]
\[ A = D \]
\[ D = v + u \]

A B C D = variables in function scope

15
Example

\[ t = A - B \]
\[ u = A - C \]
\[ v = t + u \]
\[ A = D \]
\[ D = v + u \]

R1   R2   R3
\[ \begin{array}{c}
A \\
B \\
C \\
D \\
t \\
u \\
v
\end{array} \]

A B C D = variables in function scope

A,B,C,D = temporaries in block
Example

\[ t = A - B \]
\[ u = A - C \]
\[ v = t + u \]
\[ A = D \]
\[ D = v + u \]

\[ t = A - B \]
LD R1, A
LD R2, B
SUB R2, R1, R2

\[ u = A - C \]
LD R3, C
SUB R1, R1, R3

\[ v = t + u \]
ADD R3, R2, R1

\[ A \quad B \quad C \quad D \quad t \quad u \quad v \]

A B C D = variables in function scope
\t u v = temporaries in block
Example

\[ t = A - B \]
\[ u = A - C \]
\[ v = t + u \]
\[ A = D \]
\[ D = v + u \]

\[ t = A - B \]
LD R1, A
LD R2, B
SUB R2, R1, R2

\[ u = A - C \]
LD R3, C
SUB R1, R1, R3

\[ v = t + u \]
ADD R3, R2, R1

A B C D = variables in function scope

R1  R2  R3  A  B  C  D  t  u  v

R1  R2  R3  A  t
A, R1  B  C  D  R2

R1  R2  R3  u  t  C
A  B  C, R3  D  R2  R1

R1  R2  R3  u  t  v
A  B  C  D  R2  R1  R3
Example

\[
\begin{align*}
  t &= A - B \\
  u &= A - C \\
  v &= t + u \\
  A &= D \\
  D &= v + u
\end{align*}
\]

A B C D = variables in function scope

\( t \ u \ v = \) temporaries in block
Example

A = D
LD R2, D

t = A – B
u = A – C
v = t + u
A = D
D = v + u

A B C D = variables in function scope
t u v = temporaries in block
Example

A = D
LD R2, D

t = A – B
u = A – C
v = t + u
A = D
D = v + u

A B C D = variables in function scope
t u v = temporaries in block
Example

A = D
LD R2, D

t = A – B
u = A – C
v = t + u

D = v + u
ADD R1, R3, R1

A B C D = variables in function scope
t u v = temporaries in block
Example

A = D
LD R2, D

t = A – B
u = A – C
v = t + u
A = D
D = v + u
ADD R1,R3,R1

A B C D = variables in function scope
t u v = temporaries in block
Example

\[
\begin{align*}
A &= D \\
LD & R2, D \\
t &= A - B \\
u &= A - C \\
v &= t + u \\
A &= D \\
D &= v + u \\
ADD & R1,R3,R1 \\
exit \\
ST & A, R2 \\
ST & D, R1
\end{align*}
\]

A B C D = variables in function scope

\[
\begin{align*}
t & u \\
u & A,D \\
v & D
\end{align*}
\]

A B C D = variables in function scope

t u v = temporaries in block
Example

A = D
   LD R2, D

t = A – B
u = A – C
v = t + u
A = D
D = v + u
   ADD R1, R3, R1

exit
   ST A, R2
   ST D, R1

A B C D = variables in function scope
t u v = temporaries in block
Variable Liveness

• A statement $x = y + z$:
  ▶ defines $x$
  ▶ uses $y$ and $z$

• A variable $x$ is live at a program point if its value is used at a later point, prior to any (re-)definition of $x$

<table>
<thead>
<tr>
<th>$y = 42$</th>
<th>x undefined, y live, z undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z = 73$</td>
<td>x undefined, y live, z live</td>
</tr>
<tr>
<td>$x = y + z$</td>
<td>x is live, y dead, z dead</td>
</tr>
<tr>
<td>print($x$)</td>
<td>x is dead, y dead, z dead</td>
</tr>
</tbody>
</table>

(showing state after the statement)
Computing Liveness Information

• Between basic blocks – data-flow analysis (next lecture)

• Within a single basic block?

• **Idea**
  – use symbol table to record next-use information
  – scan basic block backwards
  – update next-use for each variable
Computing Liveness Information

- **INPUT**: A basic block $B$ of three-address statements. Symbol table initially shows all non-temporary variables in $B$ as being live on exit.

- **OUTPUT**: For each statement $i$: $x = y + z$ in $B$, liveness and next-use information of $x$, $y$, and $z$ at *(just after)* $i$.

- Start at the last statement in $B$ and scan backwards
  - At each statement $i$: $x = y + z$ in $B$, we do the following:
    1. Attach to $i$ the information currently found in the symbol table regarding the next use and liveness of $x$, $y$, and $z$.
    2. In the symbol table, set $x$ to “not live”.
    3. In the symbol table, set $y$ and $z$ to “live” and their next use to $i$. 
Computing Liveness Information

1: \( y = 42 \)
2: \( z = 73 \)
3: \( x = y + z \)
4: \( \text{print}(x) \) x dead, y dead, z dead
Computing Liveness Information

1: \( y = 42 \)
2: \( z = 73 \)
3: \( x = y + z \)  \( x \) live (4), \( y \) dead, \( z \) dead
4: \( \text{print}(x) \)  \( x \) dead, \( y \) dead, \( z \) dead
Computing Liveness Information

1: y = 42
2: z = 73
3: x = y + z
4: print(x)

x dead, y live (3), z live (3)
x live (4), y dead, z dead
x dead, y dead, z dead
Computing Liveness Information

1: \( y = 42 \)  \hspace{1cm} \text{x dead, y live (3), z dead}

2: \( z = 73 \)  \hspace{1cm} \text{x dead, y live (3), z live (3)}

3: \( x = y + z \)  \hspace{1cm} \text{x live (4), y dead, z dead}

4: \texttt{print(x)}  \hspace{1cm} \text{x dead, y dead, z dead}
Computing Liveness Information

1: y = 42  x dead, y live (3), z dead
2: z = 73  x dead, y live (3), z live (3)
3: x = y + z x live (4), y dead, z dead
4: print(x) x dead, y dead, z dead
Computing Liveness Information

• Start at the last statement in B and scan backwards
  – At each statement i: x = y + z in B, we do the following:
    1. Attach to i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
    2. In the symbol table, set x to “not live”.
    3. In the symbol table, set y and z to “live” and their next use to i.

\[
\begin{align*}
x &= 1 \\
y &= x + 3 \\
z &= x \times 3 \\
x &= x \times z
\end{align*}
\]

can we change the order between steps 2 and 3 in the algorithm?
Local Register Allocation

( There are many design choices — we show one possible design )

• Simple rules
  ‣ If y is currently in a register, pick a register already containing y as R_y. No need to load this register
  ‣ If y is not in a register, but there is a register that is currently empty, pick one such register as R_y and load it

• Complicated case
  ‣ y is not in a register, but there is no free register
Local Register Allocation

• Instruction: \( x = y + z \)
• Consider the case where
  – \( y \) is not in a register, no free register
  – \( R \) is an occupied register holding value of a variable \( v \)
• Possibilities
  – if the value \( v \) is available somewhere other than \( R \), we can allocate \( R \) to be \( R_y \)
  – if \( v \) is \( x \), the value computed by the instruction, we can use it as \( R_y \) (it is going to be overwritten anyway)
  – if \( v \) is not used later, we can use \( R \) as \( R_y \)
  – otherwise: spill the value to memory by \( ST \ v, R \)
Global Register Allocation

• So far we assumed that register values are **written back to memory at the end of every basic block**

• Want to save load/stores by keeping frequently accessed values in registers (e.g., loop counters)
Global Register Allocation

• Idea: compute “weight” for each variable
  ‣ each use of \( v \) in \( B \) prior to any definition of \( v = 1 \) point
  ‣ each variable \( v \) computed in \( B \) and live on exit = 2 points
    (as we save the store/load between blocks)

\[
\text{cost}(v) = \sum_B \text{use}(v, B) + 2 \cdot \text{live}(v, B)
\]

– after computing weights, allocate registers to the “heaviest” values
Global Register Allocation

• Idea: compute “weight” for each variable
  ‣ each use of $v$ in B prior to any definition of $v = 1$ point
  ‣ each variable $v$ computed in B and live on exit = 2 points
    (as we save the store/load between blocks)

$$\text{cost}(v) = \sum_{B} \text{use}(v, B) + 2 \cdot \text{live}(v, B)$$

– after computing weights, allocate registers to the “heaviest” values
Global Register Allocation

- Idea: compute “weight” for each variable
  - each use of \(v\) in \(B\) prior to any definition of \(v = 1\) point
  - each variable \(v\) computed in \(B\) and live on exit = 2 points
    (as we save the store/load between blocks)

\[
\text{cost}(v) = \sum_{B} \text{use}(v, B) + 2 \cdot \text{live}(v, B)
\]

- after computing weights, allocate registers to the “heaviest” values
Example

\[
\begin{align*}
\text{B1:} & \quad a = b + c \\
& \quad d = d - b \\
& \quad e = a + f \\
\text{B2:} & \quad f = a - d \\
\text{B3:} & \quad b = d + f \\
& \quad e = a - c \\
\text{B4:} & \quad b = d + c
\end{align*}
\]

\[
\text{cost}(v) = \sum_B \text{use}(v, B) + 2 \cdot \text{live}(v, B)
\]
Example

\[a = b + c\]
\[d = d - b\]
\[e = a + f\]

\[f = a - d\]

\[b = d + c\]

\[b = d + f\]
\[e = a - c\]

\[b, d, e, f\] live

\[\text{cost}(v) = \sum_B \text{use}(v, B) + 2 \cdot \text{live}(v, B)\]
Example

2 uses of b prior to def

B1

a = b + c
d = d – b
e = a + f

acdef

B2

f = a – d

cdef

B3

b = d + f
e = a – c

acdf

B4

b = d + c

bcdef

b,c,d,e,f live

\[ \text{cost}(v) = \sum_B \text{use}(v, B) + 2 \cdot \text{live}(v, B) \]
Example

\[ a = b + c \]
\[ d = d - b \]
\[ e = a + f \]

\[ b = d + f \]
\[ e = a - c \]

\[ b, c, d, e, f \text{ live} \]

\[ \text{cost}(v) = \sum_B \text{use}(v, B) + 2 \cdot \text{live}(v, B) \]
### Example

**Equations:**
- \( a = b + c \)
- \( d = d - b \)
- \( e = a + f \)

**Variable Definitions:**
- \( f = a - d \)
- \( b = d + f \)

**Cost Calculation:**
\[
\text{cost}(v) = \sum_B \text{use}(v, B) + 2 \cdot \text{live}(v, B)
\]

<table>
<thead>
<tr>
<th>( v )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{cost}(v) )</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Diagram:**
- **B1:** \( a = b + c \), \( d = d - b \), \( e = a + f \)
- **B2:** \( f = a - d \)
- **B3:** \( b = d + f \), \( e = a - c \)
- **B4:** \( b = d + c \)

**Legend:**
- Defined earlier:
- Live on exit:

**Cost Matrix:**

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B1 )</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Notes:**
- Variables \( a, b, c, d, e, f \) live on exit.
Example

LD R1, b
LD R2, d

LD R3, c
ADD R0, R1, R3
SUB R2, R2, R1
LD R3, f
ADD R3, R0, R3
ST e, R3

B1

SUB R3, R0, R2
ST f, R3

B2

LD R3, f
ADD R1, R2, R3
LD R3, c
SUB R3, R0, R3
ST e, R3

B3

ST b, R1
ST a, R2

LD R3, c
ADD R1, R2, R3

B4

ST b, R1
ST d, R2
Register Allocation by Graph Coloring

• Address register allocation by
  ‣ Liveness analysis
  ‣ Reduction to graph coloring

• Main idea
  ‣ Register allocation = coloring of an interference graph
  ‣ Every node is a variable
  ‣ Edge between variables that “interfere” = are both live at the same time
  ‣ Number of colors = number of registers
Example

interference graph

life span

time
a = read();
b = read();
c = read();
a = a + b + c;
if (a<10) {
    d = c + 8;
    print(c);
} else if (a<20) {
    e = 10;
    d = e + a;
    print(e);
} else {
    f = 12;
    d = f + a;
    print(f);
}
print(d);
a = read();
b = read();
c = read();
a = a + b + c;
if (a<10) {
    d = c + 8;
    print(c);
} else if (a<20) {
    e = 10;
    d = e + a;
    print(e);
} else {
    f = 12;
    d = f + a;
    print(f);
}
print(d);
Example: Interference Graph

Example code:

```c
a = read();
b = read();
c = read();
a = a + b + c;
if (a<10) goto B2 else goto B3

b = a + c;
d = c + 8;
print(c);
if (a<20) goto B4 else goto B5

e = 10;
d = e + a;
print(e);
f = 12;
d = f + a;
print(f);
```

Intermediate code:

```c
print(d);
```
Register Allocation by Graph Coloring

- Variables that interfere with each other cannot be allocated the same register

- Graph coloring
  - classic problem: how to color the nodes of a graph with the lowest possible number of colors such that no two adjacent nodes have the same color
  - bad news: problem is NP-complete
  - good news: there are pretty good heuristic approaches
Heuristic Graph Coloring

• **Idea**: color nodes one by one, coloring the “easiest” node last

• “Easiest nodes” are ones that have lowest degree
  – fewer conflicts

• **Algorithm at high-level**
  ‣ find a lowest-degree node
  ‣ remove lowest-degree node from the graph
  ‣ color the reduced graph **recursively**
  ‣ re-attach the that node
Heuristic Graph Coloring

stack: ε
Heuristic Graph Coloring

stack: ε

stack: b
Heuristic Graph Coloring

stack: $\varepsilon$

stack: b

stack: cb
Heuristic Graph Coloring

stack: ε

stack: cb

stack: b

stack: acb
Heuristic Graph Coloring

stack: acb
Heuristic Graph Coloring

stack: acb

stack: eacb
Heuristic Graph Coloring

stack: acb

stack: eacb

stack: deacb

35
Heuristic Graph Coloring

```
stack: acb
```

```
stack: eachb
```

```
stack: deacb
```

```
stack: fdeacb
```
Heuristic Graph Coloring
Heuristic Graph Coloring

stack: acb

stack: eacb

stack: deacb

stack: fdeacb

stack: deacb

stack: eachb
Heuristic Graph Coloring

stack: acb
stack: eacb
stack: deacb
stack: fdeacb
stack: deacb
stack: eacb
stack: acb
Heuristic Graph Coloring

stack: acb

stack: eachb

stack: deacb

stack: fdeacb

stack: deacb

stack: eachb

stack: acb

stack: cb
Heuristic Graph Coloring

stack: cb
Heuristic Graph Coloring

stack: cb

stack: b
Heuristic Graph Coloring
Heuristic Graph Coloring

Result:
3 registers for 6 variables
Can we do with 2 registers?
Heuristic Graph Coloring

• Two sources of non-determinism in the algorithm
  – choosing which of the (possibly many) nodes of lowest degree should be detached
  – choosing a free color from the available colors
Heuristic Graph Coloring

stack: ε
Heuristic Graph Coloring

stack: ε  

stack: f
Heuristic Graph Coloring

stack: ε

stack: f

stack: ef
Heuristic Graph Coloring

stack: ε

stack: f

stack: ef

stack: def
Heuristic Graph Coloring

stack: def
Heuristic Graph Coloring

stack: def

stack: adef
Heuristic Graph Coloring

Stack: def

Stack: adef

Stack: cadef
Heuristic Graph Coloring
Heuristic Graph Coloring

stack: def
stack: adef
stack: cadef
stack: bcadef

stack: cadef

stack: cadef

b1
Heuristic Graph Coloring

stack: def

stack: adef

stack: cadef

stack: bcadef

stack: cadef

stack: adef

stack: bcadef
Heuristic Graph Coloring

stack: def

stack: adef

stack: cadef

stack: bcadef

stack: cadef

stack: adef

stack: def
Heuristic Graph Coloring
Heuristic Graph Coloring
Heuristic Graph Coloring

stack: ef
Heuristic Graph Coloring

stack: ef

stack: f
Heuristic Graph Coloring

stack: ef

stack: f

stack: ε
Heuristic Graph Coloring

Result: 3 registers for 6 variables (as before)
Now for some Assembly Code
Intel IA-32 Assembly

- Going from Assembly to Binary...
  - Assembling
  - Linking

- AT&T syntax vs. Intel syntax

- We will use AT&T syntax
  - matches GNU assembler (GAS)
## AT&T vs. Intel Syntax

<table>
<thead>
<tr>
<th>Attribute</th>
<th>AT&amp;T</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter order</td>
<td>Source, destination</td>
<td>Destination, source</td>
</tr>
<tr>
<td>Parameter width</td>
<td>Mnemonics are suffixed with a letter</td>
<td>Derived from the name of the register that is used</td>
</tr>
<tr>
<td></td>
<td>indicating the size of the operands</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(e.g., “b” for byte, “w” for word,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“l” for double-word, “q” for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>quad-word)</td>
<td></td>
</tr>
<tr>
<td>Symbols</td>
<td>Immediate values prefixed by “$”,</td>
<td>The assembler automatically detects the type of symbols; i.e., if</td>
</tr>
<tr>
<td></td>
<td>registers prefixed by “%”</td>
<td>they are registers, immediates, or constants.</td>
</tr>
<tr>
<td>Effective addresses</td>
<td>Special syntax:</td>
<td>Use variables, and need to be in square brackets; additionally, size</td>
</tr>
<tr>
<td></td>
<td>(\text{DISP}\text{(BASE,INDEX,SCALE)})</td>
<td>keywords like byte, word, or dword have to be used.</td>
</tr>
</tbody>
</table>

```assembly
movl mem_location(%ebx,%ecx,4), %eax
```

```assembly
mov eax, dword [ebx + ecx*4 + mem_location]
```
IA-32 Registers

- Eight 32-bit general-purpose registers
  - EAX – accumulator for operands and result data. Used to return value from function calls.
  - EBX – pointer to data. Often used as array-base address
  - ECX – counter for string and loop operations
  - EDX – I/O pointer (GP for us)
  - ESI – GP and source pointer for string operations
  -EDI – GP and destination pointer for string operations
  - EBP – stack frame (base) pointer
  - ESP – stack pointer
- EFLAGS register
- EIP (instruction pointer) register
- Six 16-bit segment registers
- … (ignore the rest for our purposes)
Not all registers are born equal

- **EAX**
  - Required operand of MUL,IMUL,DIV and IDIV instructions
  - Contains the result of these operations
- **EDX**
  - Stores remainder of a DIV or IDIV instruction
    (EAX stores quotient)
- **ESI, EDI**
  - ESI – required source pointer for string instructions
  - EDI – required destination pointer for string instructions
- **Destination Registers of Arithmetic operations**
  - EAX, EBX, ECX, EDX
- **EBP** – stack frame (base) pointer
- **ESP** – stack pointer
IA-32 Addressing Modes

- Machine-instructions take zero or more operands
  - Source operand
    - Immediate
    - Register
    - Memory location
  - Destination operand
    - Register
    - Memory location
Immediate and Register Operands

• Immediate
  – Value specified in the instruction itself
  – GAS syntax – immediate values preceded by $ 
  – add $4, %esp

• Register
  – Register name is used
  – GAS syntax – register names preceded with %
  – mov %esp, %ebp
Memory and Base Displacement Operands

- **Memory operands**
  - Value at given address
  - GAS syntax — parentheses
    - `mov (%eax), %eax`

- **Base displacement**
  - Value at computed address
  - Address computed out of base register, offset register, scale factor, displacement
    - `address = base + (offset*scale) + disp`
  - Examples:
    - `movl $42, $2(%eax)`
    - `movl $42, $1(%eax,%ecx,4)`
Base Displacement Addressing

Array Base Reference

\[
\begin{array}{cccccccccc}
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
7 & 0 & 2 & 4 & 5 & 6 & 7 & 1 \\
\end{array}
\]

\[
\text{mov} \ (\%ecx,\%ebx,4), \ %eax
\]

base = \%ecx
offset = \%ebx
scale = 4
Base Displacement Addressing

<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Array Base Reference

\[
\text{mov} \ (\%ecx,\%ebx,4), \ %eax
\]

address = base + (index*scale) + displacement

address = %ecx + (3*4) + 0 = %ecx + 12

base = %ecx
offset = %ebx
scale = 4
Base Displacement Addressing

Array Base Reference

\[
\text{(%ecx, %ebx, 4)} = \text{mem[ecx + ebx \times 4]}
\]

\[
\text{mov (%ecx, %ebx, 4), %eax}
\]

address = base + (index\times scale) + displacement

address = %ecx + (3\times 4) + 0 = %ecx + 12

base = %ecx
offset = %ebx
scale = 4
Instruction Selection

• How do we choose which instructions to emit from our IR?
• Greatly depends on how we design our IR. We see two examples in this lecture:
  ‣ Tree-structured IR — Tiling
  ‣ 3AC sequence — Peephole Matching
Canonical IR — Tiling

Example: 

\[ x = x + 1; \]
Canonical IR — Tiling

Example:

\[ x = x + 1; \]

\[
\begin{align*}
\text{mov} & \ t1, \ [bp+x] \\
\text{mov} & \ t2, \ t1 \\
\text{add} & \ t2, \ 1 \\
\text{mov} & \ [bp+x], \ t2
\end{align*}
\]
Tiles

- Capture compiler’s understanding of the instruction set
- Tile = sequence of instructions that update a fresh temporary (may need extra `mov`'s) and associated IR tree
- Each outgoing edge represents a temporary

\[
\text{mov } t2, t1 \\
\text{add } t2, 1
\]
CISC vs. RISC Tiling

(e.g. ARM) RISC

(e.g. x86) CISC

MOVE
MEM
+ @x
FP
MEM
+ 1
FP @x

add [bp+x], 1
CISC vs. RISC Tiling

(e.g. ARM) RISC

CISC (e.g. x86)

mov t1, [bp+x]
mov t2, t1
add t2, 1
mov [bp+x], t2

add [bp+x], 1
CISC vs. RISC Tiling

(e.g. ARM)  RISC

MOVE

MEM

+ 1

FP @x

CISC  (e.g. x86)

MOVE

MEM

+ 1

FP @x

mov t1, [bp+x]
mov t2, t1
add t2, 1
mov [bp+x], t2

r/m32 +
CONST(k)
Example Tiles for “+”

Based on *Intel Architecture Manual*, Vol 2, 3-17
Computing a Tiling

- **Maximal Munch** — a **greedy** approach
  - Start at statement root
  - Find largest tile covering node, matching all children and outgoing edges
  - Invoke recursively on temporaries dangling from tile
  - Generate code for tile
    - (code for children will have been generated in recursive calls)
Timing Model

- **Idea**: associate cost with each tile (proportional to # cycles to execute)
  - sum of costs approximates execution time

Total cost = 5
Computing Optimum Tiling

- **Goal**: find minimum total cost tiling of tree
- **Algorithm**: for every node, find minimum total cost tiling of that node and sub-tree.
- **Lemma**: once minimum cost tiling for all descendants of a node is known, one can find minimum cost tiling of the node (sub-tree) by trying out all possible tiles matching the top

⇒ start from leaves, work upward to top node
Dynamic Programming

Example:

\[ a[i] \]

\[
\begin{align*}
\text{MEM} & \quad + \\
\text{MEM} & \quad + \\
\text{FP} & \quad \text{CONST (@a)} \\
\text{CONST (4)} & \quad \text{MEM} \\
\text{FP} & \quad \text{CONST (@i)}
\end{align*}
\]
Dynamic Programming

Example:

\[
a[i]
\]

\[
\text{MEM} + \text{MEM} \ast \text{CONST}(4) + \text{FP} + \text{MEM} + \text{FP} \ast \text{CONST}(\text{@i})
\]
Dynamic Programming

Example:

\[ a[i] \]
Example:

\[ a[i] \]

\[ \text{MEM} \]

\[ + \]

\[ \text{MEM} \]

\[ + \]

\[ \text{FP} \]

\[ \text{CONST}(@a) \]

\[ \times \]

\[ \text{CONST}(4) \]

\[ + \]

\[ \text{MEM} \]

\[ + \]

\[ \text{FP} \]

\[ \text{CONST}(@i) \]
Dynamic Programming

Example:

\[ a[i] \]
Dynamic Programming

Example:

```
\[ a[i] \]
```
Dynamic Programming

Example:

\[ a[i] \]
Example:

\[ a[i] \]

```
movl a(%ebp), t1
movl i(%ebp), t2
movl (t1,t2,4), t3
```
Peephole Matching

- **Basic idea**: discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a small sliding window (“peephole”) over code and search for improvement

- Classic examples:

  $$
  *p := R1
  R15 := *p
  \downarrow
  *p := R1
  R15 := R1
  \text{store followed by load}
  \]

  $$
  R7 := R2 + 0
  R10 := R4 * R7
  \downarrow
  R10 := R4 * R2
  \text{algebraic identities}
  \]

  $$
  \text{goto L}_{10}
  \downarrow
  \text{L}_{10}: \text{goto L}_{11}
  \text{jump to jump}
  \]
Peephole Matching

• How to implement it?

• Modern instruction selectors break problem into three tasks:  
  \((Davidson, 1989)\)
Peephole Matching

Expander

- Turns IR code into a low-level IR (LLIR) such as RTL
- Operation-by-operation, template-driven rewriting
- LLIR form includes all direct effects (e.g., setting cc)
- Significant, albeit constant, expansion of size

IR → Expander → LLIR → Simplifier → LLIR → Matcher → ASM
Peephole Matching

Simplifier

- Looks at LLIR through window and rewrites is
- Uses forward substitution, algebraic simplification, local constant propagation, and dead-effect elimination
- Performs local optimization within window
Peephole Matching

Matcher

• Compares simplified LLIR against a library of patterns
• Picks low-cost pattern that captures effects
• Must preserve LLIR effects, may add new ones \((e.g.,\ set\ cc)\)
• Generates the assembly code output

IR → Expander → LLIR → Simplifier → LLIR → Matcher → ASM
# Peephole Matching — Example

## Original IR Code

<table>
<thead>
<tr>
<th>op</th>
<th>arg₁</th>
<th>arg₂</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>2</td>
<td>y</td>
<td>t₁</td>
</tr>
<tr>
<td>−</td>
<td>x</td>
<td>t₁</td>
<td>w</td>
</tr>
</tbody>
</table>

\[ t₁ = r_{14} \]

\[ w = r_{20} \]

## Expand

\[ r_{10} := 2 \]
\[ r_{11} := @y \]
\[ r_{12} := bp + r_{11} \]
\[ r_{13} := *(r_{12}) \]
\[ r_{14} := r_{10} * r_{13} \]
\[ r_{15} := @x \]
\[ r_{16} := bp + r_{15} \]
\[ r_{17} := *(r_{16}) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ r_{19} := @w \]
\[ r_{20} := bp + r_{19} \]
\[ *(r_{20}) := r_{18} \]
Peephole Matching — Example

LLIR Code

\[
\begin{align*}
r_{10} &:= 2 \\
r_{11} &:= \@y \\
r_{12} &:= \text{bp} + r_{11} \\
r_{13} &:= *(r_{12}) \\
r_{14} &:= r_{10} * r_{13} \\
r_{15} &:= \@x \\
r_{16} &:= \text{bp} + r_{15} \\
r_{17} &:= *(r_{16}) \\
r_{18} &:= r_{17} - r_{14} \\
r_{19} &:= \@w \\
r_{20} &:= \text{bp} + r_{19} \\
*(r_{20}) &:= r_{18}
\end{align*}
\]

Simplify

LLIR Code

\[
\begin{align*}
r_{13} &:= *(\text{bp} + \@y) \\
r_{14} &:= 2 * r_{13} \\
r_{17} &:= *(r_{\text{arp}} + \@x) \\
r_{18} &:= r_{17} - r_{14} \\
*(\text{bp} + \@w) &:= r_{18}
\end{align*}
\]
Peephole Matching — Example

LLIR Code

\[ r_{13} := *(bp + @y) \]
\[ r_{14} := 2 * r_{13} \]
\[ r_{17} := *(r_{\text{arp}} + @x) \]
\[ r_{18} := r_{17} - r_{14} \]
\[ *(bp + @w) := r_{18} \]

Assembly (ILOC) Code

loadAI \( r_{\text{arp}}, @y \) \( \Rightarrow r_{13} \)
multi \( 2, r_{13} \) \( \Rightarrow r_{14} \)
loadAI \( r_{\text{arp}}, @x \) \( \Rightarrow r_{17} \)
sub \( r_{17}, r_{14} \) \( \Rightarrow r_{18} \)
storeAI \( r_{18} \) \( \Rightarrow r_{\text{arp}}, @w \)
Making Peephole Matching Work

Details.

• LLIR is largely machine independent
  (RTL)
• Target machine described as LLIR → ASM pattern
• Actual pattern matching
  ‣ Use a hand-coded pattern matcher
    (gcc)
  ‣ Turn patterns into grammar & use LR parser
    (VPO)
• Several important compilers use this technology
• It seems to produce good portable instruction selectors

Key strength appears to be late low-level optimization
Summary

• Register allocation
  – “Local” = with a basic block
  – “Global” = across blocks (in a single function)

• Instruction selection
  ‣ Use Tiling to match portions of IR tree
    ◦ Greedy approach is fast
    ◦ Dynamic programming more costly, but can find optimal tiling w.r.t. timing model
  ‣ Use Peephole matching directly on 3AC
Coming Up

Static Analysis