Target Architecture
Code Generation

Generated code
- must be correct
- will be suboptimal (initially)
- will be optimized (eventually)
Consider applying the same template to a larger example:

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

This might be translated as:

LD R0, b
ADD R0, R0, c
ST a, R0
LD R0, a
ADD R0, R0, e
ST d, R0

This instruction is redundant: it is loading into R0 the value that is already there.
Basic blocks and flow graphs

To help us analyze the intermediate code we will group instructions from our program into "basic blocks":

"The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks."

[p 526]
Identifying leaders

for (i=1; i<=10; i=i+1) {
    for (j=1; j<=10; j=j+1) {
        a[i,j] = 0.0;
    }
}
for (i=1; i<=10; i=i+1) {
    a[i,i] = 1.0;
}

Leaders are:
1. first instruction
2. the target of any jump
3. the instruction immediately after any jump
Flow Graph

Figure 8.9 [p. 530]

Entry and exit nodes added.

Jump targets replaced by block names.

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

ENTRY

EXIT
8.4.2 Liveness and next-use

\[ i: \quad x = \ldots \]

assuming there are no assignments to \( x \) between \( i \) and \( j \)

\[ j: \quad \ldots = x \ op \ \ldots \]

If statement \( j \) uses \( x \), then \( x \) is live at \( i \). Since we need the value of \( x \) we should try to keep it in a register.
8.4.2 Liveness and next-use

i: \[ \ldots x \ldots \]

\[ \ldots \]

assuming there is no use of \( x \) between \( i \) and \( j \)

\[ j: \quad x = \ldots \]

Statement \( j \) overwrites old value of \( x \); we say \( x \) is dead at \( i \). This means we need not preserve that value in a register.
Algorithm 8.7 [p. 528]
Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block B of three address instructions. Assume the symbol table initially shows all non-temporary variables in B as being live on exit.

OUTPUT: At each statement i: x = y + z in B, we attach to i the liveness and next-use information for x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: x = y + z in B do the following:

1) attach to statement 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" and "no next use".
3) In the symbol table, set y and z to "live" and the next uses of y and z to instruction i.
Code Transformations on basic blocks

- Local optimizations can be performed on code inside basic blocks (e.g. local common subexpression elimination, dead code elimination, statement reordering)

- Represent code inside a basic block as a DAG. (6.1.2: value-number method for individual expressions.)

- The basic blocks will themselves be connected to form a flow graph.
Example 8.10 [p. 534]

If b is live on exit:

1) \(a = b + c\)
2) \(b = a - d\)
3) \(c = b + c\)
4) \(d = b\)
Example 8.10 [p. 534]

If b is not live on exit:

1) \( a = b + c \)
2) \( d = a - d \)
3) \( c = d + c \)
Phases of a compiler

Figure 1.6, page 5 of text
8.6 A Simple Code Generator [p. 542]

- algorithm focuses on generation of code for a single basic block
- generates code for each three address code instruction
- manages register allocations/assignment to avoid redundant loads/stores
Principal uses of registers

- operator operands must be in registers
- temporaries needed within block
- variables that span multiple blocks
- stack pointer
- function arguments
"We [...] assume that for each operator, there is exactly one machine instruction that takes the necessary operands in registers and performs that operation, leaving the result in a register. The machine instructions are of the form:

- LD reg, mem
- ST mem, reg
- OP reg, reg, reg" [p. 543]

x86 assembly resources (will add more as we go along)
man as   <--- at OS prompt
8.6.1 Register and Address Descriptors

A three-address instruction of the form:
\[ v = a \text{ op } b \]

we generate:
- \text{LD } Rx, a
- \text{LD } Ry, b
- \text{OP } Rx, Rx, Ry
- \text{ST } Rx, v
8.6.1 Register and Address Descriptors

A three-address instruction of the form:

\[ v = a \text{ op } b \]

we generate:

LD Rx, a
LD Ry, b
OP Rx, Rx, Ry
ST Rx, v

where \( a, b, \) and \( v \) are int

\[ v = a + b \]

\begin{align*}
\text{movl} & \quad -4(\%\text{rbp}), \%\text{edx} \\
\text{movl} & \quad -8(\%\text{rbp}), \%\text{eax} \\
\text{addl} & \quad \%\text{edx}, \%\text{eax} \\
\text{movl} & \quad \%\text{eax}, -12(\%\text{rbp})
\end{align*}
This results in many redundant loads and stores and may not make effective use of available registers.

To better manage register use, employ two data structures:

- register descriptor
- address descriptor
register descriptor

"For each available register, a register descriptor keeps track of the variable names whose current value is in that register." [p. 543]
address descriptor

"For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found." [p. 543]
getReg function

"...getReg(I)...selects registers for each memory location associated with the three-address instruction I." [p. 544]

Note that I is an instruction, not a variable!
Example
(paraphrased from 8.6.2, page 544)

A three-address instruction of the form:

\[ v = a \text{ op } b \]

1. Use getReg\(v = a \text{ op } b\) to select registers for \(v\), \(a\) and \(b\): \(R_v\), \(R_a\), and \(R_b\) respectively
2. If \(a\) is not already in \(R_a\), generate LD \(R_a, a'\) (where \(a'\) is one of the possibly many current locations of \(a\))
3. Similarly for \(b\).
4. Generate OP \(R_v, R_a, R_b\)
copy instructions

\[ x = y \]

"We assume getReg will always choose the same register for both \( x \) and \( y \). If \( y \) is not already in that register \( R_y \), then generate the machine instruction \( LD \ R_y, y \). If \( y \) was already in \( R_y \), we do nothing. It is only necessary that we adjust the register descriptor for \( R_y \) so that it includes \( x \) as one of the values found there." [p. 544]
Writing back to memory at end of block

At the end of a basic block we must ensure that live variables are stored back into memory.

"...for each variable x whose address descriptor does not say that its value is located in the memory location for x, we must generate the instruction `ST x, R`, where R is a register in which x's value exists at the end of the block." [p. 545]
Updating register descriptors (RD) and address descriptors (AD)

1. LD R, x
   (a) Set RD of R to only x
   (b) Add R to AD of x

2. ST x, R
   (a) Add &x to AD of x

3. OP Rx, Ry, Rz for x = y op z
   (a) Set RD of Rx to only x
   (b) Set AD of x to only Rx (&x not in AD of x!)
   (c) Remove Rx from the AD of any variable other than x

4. "When we process a copy statement x = y, after generating the load for y into register Ry, if needed, and after managing descriptors as for all load statement (per rule 1):" [p. 545]
   (a) Add x to the RD of Ry
   (b) Set AD of x to only Ry
Example [p. 546]

\[ t = a - b \]
\[ u = a - c \]
\[ v = t + u \]
\[ a = d \]
\[ d = v + u \]

what does liveness and next use info looking like here?
Algorithm 8.7 [p. 528]
Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block B of three address instructions. Assume the symbol table initially shows all non-temporary variables in B as being live on exit.

OUTPUT: At each statement i: x = y + z in B, we attach to i the liveness and next-use information for x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: x = y + z in B do the following:

1) attach to statement 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" and "no next use".
3) In the symbol table, set y and z to "live" and the next uses of y and z to instruction i.
**Example [p. 546]**

INPUT: A basic block \( B \) of three address instructions. Assume the symbol table initially shows all non-temporary variables in \( B \) as being live on exit.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>u</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>v</td>
<td>t</td>
<td>u</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>d</td>
<td>v</td>
<td>u</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
1: t = a - b \\
2: u = a - c \\
3: v = t + u \\
4: a = d \\
5: d = v + u
\]
We start at the last statement in \( B \) and scan backwards to the beginning of \( B \). At each statement \( i: \)
\[
x = y + z
\]
in \( B \) do the following:
1) attach to statement \( i \) 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" and "no next use".
3) In the symbol table, set \( y \) and \( z \) to "live" and the next uses of \( y \) and \( z \) to instruction \( i \).
We start at the last statement in B and scan backwards to the beginning of B. At each statement i:

\[ x = y + z \]

in B do the following:

1) attach to statement 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" and "no next use".

3) In the symbol table, set y and z to "live" and the next uses of y and z to instruction i.
Example [p. 546]

We start at the last statement in B and scan backwards to the beginning of B. At each statement i:

\[ x = y + z \]

in B do the following:

1) attach to statement 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" and "no next use".

3) In the symbol table, set y and z to "live" and the next uses of y and z to instruction i.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>D</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>D</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>D</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>
We start at the last statement in B and scan backwards to the beginning of B. At each statement i:

\[ x = y + z \]

in B do the following:

1) attach to statement 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" and "no next use".

3) In the symbol table, set y and z to "live" and the next uses of y and z to instruction i.
We start at the last statement in B and scan backwards to the beginning of B. At each statement $i$: 

$x = y + z$

in B do the following:

1) attach to statement $i$ 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" and "no next use".
3) In the symbol table, set $y$ and $z$ to "live" and the next uses of $y$ and $z$ to instruction $i$. 

Example [p. 546]

1: $t = a - b$
2: $u = a - c$
3: $v = t + u$
4: $a = d$
5: $d = v + u$
We start at the last statement in B and scan backwards to the beginning of B. At each statement i:
\[ x = y + z \]
in B do the following:
1) attach to statement 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" and "no next use".
3) In the symbol table, set y and z to "live" and the next uses of y and z to instruction i.
We start at the last statement in B and scan backwards to the beginning of B. At each statement i:

\[ x = y + z \]

in B do the following:

1) attach to statement 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" and "no next use".
3) In the symbol table, set y and z to "live" and the next uses of y and z to instruction i.
We start at the last statement in B and scan backwards to the beginning of B. At each statement i:

\[ x = y + z \]

in B do the following:

1) attach to statement 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" and "no next use".

3) In the symbol table, set y and z to "live" and the next uses of y and z to instruction i.
We start at the last statement in B and scan backwards to the beginning of B. At each statement i:
\[ x = y + z \]
in B do the following:
1) attach to statement 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" and "no next use".
3) In the symbol table, set y and z to "live" and the next uses of y and z to instruction i.
Example [p. 546]

1: \( t = a - b \)
2: \( u = a - c \)
3: \( v = t + u \)
4: \( a = d \)
5: \( d = v + u \)

We start at the last statement in B and scan backwards to the beginning of B. At each statement \( i \):

\[ x = y + z \]

in B do the following:

1) attach to statement \( i \) 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" and "no next use".
3) In the symbol table, set \( y \) and \( z \) to "live" and the next uses of \( y \) and \( z \) to instruction \( i \).
Register descriptor

Assume just 3 registers are available for the sake of this example.

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
</table>

t = a - b
u = a - c
v = t + u
a = d
da = v + u

Address descriptor

Variables t, u, and v are compiler-generated temporary variables.
t = a - b
u = a - c
v = t + u
a = d
d = v + u

At start of block, assume the values of variables a, b, c, and d are in main memory.

Variables t, u, and v are compiler-generated temporary variables.
\[
t = a - b
\]

LD R1, a
LD R2, b
SUB R2, R1, R2
No registers are in use - pick the first two available for a and b. Choose to put t in R2 because b is not used again in this block.
\[ t = a - b \]

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>

LD R1, a
LD R2, b
SUB R2, R1, R2

\[ u = a - c \]

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>c</td>
</tr>
</tbody>
</table>

LD R3, c
SUB R1, R1, R3
\[ t = a - b \]

- \( a \) is already in \( R1 \), so no load needed.
- \( t \) is used later, so don't overwrite \( R2 \).
- Load \( c \) into \( R3 \).
- Put result into \( R1 \) since \( a \) is not needed again in this block.

\[ u = a - c \]

LD \( R3, c \)

SUB \( R1, R1, R3 \)
<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = a - b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a, R1</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R2, b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R2, R1, R2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u = a - c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c, R3</td>
</tr>
<tr>
<td>a, R1</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>v = t + u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R3, R2, R1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\[ t = a - b \]

LD R1, a
LD R2, b
SUB R2, R1, R2

\begin{align*}
&u = a - c \\
&v = t + u
\end{align*}

ADD R3, R2, R1

\[ t \text{ and } u \text{ are already in registers - no loads needed.} \]

\[ \text{Perform addition, putting the result into R3; c is no longer needed in this block.} \]
<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>t</td>
<td>v</td>
</tr>
</tbody>
</table>

The state of the system remains the same as at the end of the previous slide.
$a = d$

LD R2, d
Load d into R2, attach a to R2 as well.
\[
\begin{align*}
a &= d \\
d &= v + u
\end{align*}
\]
u and v are in registers, so no loads needed.

Cannot destroy a (exists only in R2) without storing back to memory, so use R1 for result.

Move d to R1 from R2.
<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>t</td>
<td>v</td>
</tr>
</tbody>
</table>

\(a = d\)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>R2</td>
<td>R1</td>
<td>R3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>a,d</td>
<td>v</td>
</tr>
</tbody>
</table>

\(d = v + u\)

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>a</td>
<td>v</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>R2</td>
<td>R1</td>
<td>R3</td>
</tr>
</tbody>
</table>

ADD R1, R3, R1

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>a</td>
<td>v</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>b</td>
<td>c</td>
<td>R1</td>
<td></td>
<td></td>
<td>R3</td>
</tr>
</tbody>
</table>

exit

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R2</td>
<td>R1</td>
</tr>
</tbody>
</table>

ST a, R2
ST d, R1
We're at the end of the block. Make sure that values of R1 and R2 are stored back to memory (d and a respectively). Value of R3 can be lost - it is a temporary of only this block.