Basic Block Code Generation

- Generate code one basic block at a time.

Next-Use Information

- We don’t know which path through the flow-graph has taken us to this basic block. ⇒ We can’t assume that any variables are in registers.
- We don’t know where we will go from this block. ⇒ Values kept in registers must be stored back into their memory locations before the block is exited.
We want to keep variables in registers for as long as possible, to avoid having to reload them whenever they are needed.

When a variable isn’t needed any more we free the register to reuse it for other variables. We must know if a particular value will be used later in the basic block.

If, after computing a value $X$, we will soon be using the value again, we should keep it in a register. If the value has no further use in the block we can reuse the register.

Next-Use Information II(a)

- $X$ is live at (5) because the value computed at (5) is used later in the basic block.
- $X$’s next use at (5) is (14).
- It is a good idea to keep $X$ in a register between (5) and (14).

Next-Use Information II(b)

- $X$ is dead at (12) because its value has no further use in the block.
- Don’t keep $X$ in a register after (12).

Next-Use Information III – Example

<table>
<thead>
<tr>
<th>Intermediate Code</th>
<th>Live/Dead</th>
<th>Next Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$ := $y$+$z$</td>
<td>L D D</td>
<td>(2) - -</td>
</tr>
<tr>
<td>$z$ := $x$*5</td>
<td>D L</td>
<td>(3) -</td>
</tr>
<tr>
<td>$t_7$ := $z$+1</td>
<td>L L</td>
<td>(4) (4)</td>
</tr>
<tr>
<td>$y$ := $z$-$t_7$</td>
<td>L L D</td>
<td>(5) (5) -</td>
</tr>
<tr>
<td>$x$ := $z$+$y$</td>
<td>D D D</td>
<td>- - -</td>
</tr>
</tbody>
</table>

- $x$, $y$, $z$ are live on exit, $t_7$ (a temporary) isn’t.
Next-Use Algorithm I

- A two-pass algorithm computes next-use & liveness information for a basic block.
- In the first pass we scan over the basic block to find the end. Also:
  1. For each variable \( X \) used in the block we create fields \( X\.live \) and \( X\.next\_use \) in the symbol table. Set \( X\.live:=FALSE; \)
     \( X\.next\_use:=NONE \).
  2. Each tuple \( \langle i \rangle X:=Y+Z \) stores next-use & live information. We set
     \( \langle i \rangle .X\.live:=(\langle i \rangle .Y\.live:=(\langle i \rangle .Z\.live:=FALSE \)
     \( \langle i \rangle .X\.next\_use:=(\langle i \rangle .Y\.next\_use:=(\langle i \rangle .Z\.next\_use:=NONE \.

Next-Use Algorithm II

1. Scan **forwards** over the basic block:
   - Initialize the symbol table entry for each used variable, and the tuple data for each tuple.
2. Scan **backwards** over the basic block. For every tuple \( \langle i \rangle X:=Y \ op \ Z \) do:
   - Copy the live/next-use-info from \( X, Y, Z \)’s symbol table entries into the tuple data for tuple \( \langle i \rangle \).
   - Update \( X, Y, Z \)’s symbol table entries:
     \( X\.live := FALSE; \)
     \( X\.next\_use := NONE; \)
     \( Y\.live := TRUE; \)
     \( Z\.live := TRUE; \)
     \( Y\.next\_use := i; \)
     \( Z\.next\_use := i; \)

Next-Use Example I – Forward Pass

<table>
<thead>
<tr>
<th>SyTab-Info</th>
<th>Instr.-Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>live</td>
<td>next_use</td>
</tr>
<tr>
<td>( i )</td>
<td>( x )</td>
</tr>
<tr>
<td>1</td>
<td>( x:=y+z )</td>
</tr>
<tr>
<td>2</td>
<td>( z:=x*5 )</td>
</tr>
<tr>
<td>3</td>
<td>( y:=z-7 )</td>
</tr>
<tr>
<td>4</td>
<td>( x:=z+y )</td>
</tr>
</tbody>
</table>

Next-Use Example II – Backwards Pass

<table>
<thead>
<tr>
<th>SyTab-Info</th>
<th>Instr.-Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>live</td>
<td>next_use</td>
</tr>
<tr>
<td>( i )</td>
<td>( x )</td>
</tr>
<tr>
<td>1</td>
<td>( x := y+z )</td>
</tr>
<tr>
<td>2</td>
<td>( z := x*5 )</td>
</tr>
<tr>
<td>3</td>
<td>( y := z-7 )</td>
</tr>
<tr>
<td>4</td>
<td>( x := z+y )</td>
</tr>
</tbody>
</table>

- The data in each row reflects the state in the symbol table and in the data section of instruction \( i \) after \( i \) has been processed.
During code generation we need to keep track of what’s in each register (a Register Descriptor). One register may hold the values of several variables (e.g. after \( x := y \)).

We also need to know where the values of variables are currently stored (an Address Descriptor). A variable may be in one (or more) register, on the stack, in global memory; all at the same time.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Memory</td>
</tr>
<tr>
<td>----</td>
<td>-------</td>
</tr>
<tr>
<td>x</td>
<td>fp(16)</td>
</tr>
<tr>
<td>y</td>
<td>fp(20)</td>
</tr>
<tr>
<td>z</td>
<td>0x2020</td>
</tr>
<tr>
<td>t1</td>
<td></td>
</tr>
</tbody>
</table>

A Simple Code Generator

We have:

A flowgraph: We generate code for each individual basic block.

An Address Descriptor (AD): We store the location of each variable: in register, on the stack, in global memory.

A Register Descriptor (RD): We store the contents of each register.

Next-Use Information: We know for each point in the code whether a particular variable will be referenced later on.

We need:

GenCode(i: x := y op z): Generate code for the i:th intermediate code instruction.

GetReg(i: x := y op z): Select a register to hold the result of the operation.
Machine Model

- We will generate code for the address-register machine described in the book. It is a CISC, not a RISC; it is similar to the x86 and MC68k.
- The machine has \( n \) general purpose registers \( R_0, R_1, \ldots, R_n \).

```plaintext
MOV M, R
Load variable M into register R.
```

```plaintext
MOV R, M
Store register R into variable M.
```

```plaintext
OP M, R
Compute \( R := R \text{ OP } M \), where \text{OP} \text{ is one of } \text{ADD}, \text{SUB}, \text{MUL}, \text{DIV}.
```

```plaintext
OP R2, R1
Compute \( R1 := R1 \text{ OP } R2 \), where \text{OP} \text{ is one of } \text{ADD}, \text{SUB}, \text{MUL}, \text{DIV}.
```

GenCode((i): \( X := Y \text{ OP } Z \))

- \( L \) is the location in which the result will be stored. Often a register.
- \( Y' \) is the most favorable location for \( Y \). I.e. a register if \( Y \) is in a register, \( Y \)'s memory location otherwise.

```plaintext
1 L := GetReg(i: \( X := Y \text{ OP } Z \)).
2 Y' := “best” location for \( Y \). IF \( Y \) is not in \( Y' \) THEN gen(MOV \( Y' \), L).
3 Z' := “best” location for \( Z \).
4 gen(OP \( Z' \), L)
5 Update the address descriptor: \( X \) is now in location \( L \).
6 Update the register descriptor: \( X \) is now only in register \( L \).
7 IF (i).Y.next use=NONE THEN update the register descriptor: \( Y \) is not in any register. Same for \( Z \).
```

GenCode((i): \( X := Y \))

- Often we won’t have to generate any code at all for the tuple \( X := Y \); instead we just update the address and register descriptors (AD & RD).

```plaintext
IF Y only in mem. location L THEN
  R := GetReg(); gen(MOV Y, R);
  AD: \( Y \) is now only in reg R.
  RD: R now holds \( Y \).
ELSE IF Y is in register R THEN
  AD: \( X \) is now only in register R.
  RD: R now holds X.
  IF (i).Y.next use=NONE THEN RD: No register holds \( Y \).
ELSE IF \( X \) has a next use and there exists an occupied register R THEN Store R into its memory location and RETURN R;
ELSE RETURN the memory location of \( X \).
```

GetReg(i: \( X := Y \text{ OP } Z \))

- If we won’t be needing the value stored in \( Y \) after this instruction, we can reuse \( Y \)'s register.

```plaintext
IF
  Y is in register R and R holds only \( Y \)
  (i).Y.next use=NONE
THEN RETURN R;
ELSIF there’s an empty register R available THEN RETURN R;
ELSIF
  \( X \) has a next use and there exists an occupied register R THEN Store R into its memory location and RETURN R;
OTHERWISE RETURN the memory location of \( X \).
```
Code Generation Example I

<table>
<thead>
<tr>
<th>Interm. Code</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := y + z</td>
<td>MOV y, r0</td>
</tr>
<tr>
<td></td>
<td>ADD z, r0</td>
</tr>
<tr>
<td>z := x * 5</td>
<td>MUL 5, r0</td>
</tr>
<tr>
<td>y := z - 7</td>
<td>MOV r0, r1</td>
</tr>
<tr>
<td></td>
<td>SUB 7, r1</td>
</tr>
<tr>
<td>x := z + y</td>
<td>MOV r0, z</td>
</tr>
<tr>
<td></td>
<td>ADD r1, r0</td>
</tr>
</tbody>
</table>

Note that x and y are kept in registers until the end of the basic block. At the end of the block, they are returned to their memory locations.

Code Generation Example II

<table>
<thead>
<tr>
<th>Interm.</th>
<th>Machine</th>
<th>RD</th>
<th>AD</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := y + z</td>
<td>MOV y, r0</td>
<td>r0 ≡ x</td>
<td>x ≡ r0</td>
<td>T</td>
</tr>
<tr>
<td>z := x * 5</td>
<td>MUL 5, r0</td>
<td>r0 ≡ z</td>
<td>z ≡ r0</td>
<td>F</td>
</tr>
<tr>
<td>y := z - 7</td>
<td>MOV r0, r1</td>
<td>r0 ≡ z</td>
<td>z ≡ r0</td>
<td>T</td>
</tr>
</tbody>
</table>

Code Generation Example III

<table>
<thead>
<tr>
<th>Interm.</th>
<th>Machine</th>
<th>RD</th>
<th>AD</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := z + y</td>
<td>MOV r0, z</td>
<td>r0 ≡ z</td>
<td>z ≡ mem</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r1 ≡ y</td>
<td>z ≡ r1</td>
<td></td>
</tr>
<tr>
<td>ADD r1, r0</td>
<td>r0 ≡ x</td>
<td>x ≡ r0</td>
<td>y ≡ r1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r1 ≡ y</td>
<td></td>
<td>z ≡ mem</td>
<td></td>
</tr>
<tr>
<td>M ambit y</td>
<td>MOV r1, y</td>
<td>y ≡ mem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M ambit  x</td>
<td>MOV r0, x</td>
<td>x ≡ mem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary
• This lecture is taken from the Dragon book:
  Next-Use Information  534–535
  Simple Code Generation  535–541.
  Address & Register Descriptors  537

• Register allocation requires next-use information, i.e. for each reference to \( x \) we need to know if \( x \)'s value will be used further on in the program.
• We also need to keep track of what's in each register. This is sometimes called register tracking.
• We need a register allocator, a routine that picks registers to hold the contents of intermediate computations.