Basic Block Code Generation

- Generate code one basic block at a time.
- 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.

Next-Use Information I

- 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

X is \textit{live} at (5) because the value computed at (5) is used later in the basic block.

X’s \texttt{next use} at (5) is (14).

It is a good idea to keep X in a register between (5) and (14).

X is \textit{dead} at (12) because its value has no further use in the block.

Don’t keep X in a register after (12).

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 (i) X:=Y+Z stores next-use & live information. We set


Next-Use Algorithm II

1. Scan \texttt{forwards} over the basic block: Initialize the symbol table entry for each used variable, and the tuple data for each tuple.

2. Scan \texttt{backwards} over the basic block. For every tuple (i): x := y \textit{op} z do:

   1. Copy the live/next use-info from x, y, z’s symbol table entries into the tuple data for tuple (i).

   2. 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 Information III – Example

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

x, y, z are \textit{live on exit}, t_7 (a temporary) isn’t.
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) x:=y+z</td>
<td>F</td>
</tr>
<tr>
<td>(2) z:=x*5</td>
<td>F</td>
</tr>
<tr>
<td>(3) y:=z-7</td>
<td>F</td>
</tr>
<tr>
<td>(4) x:=z+y</td>
<td>F</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>(4) x := z+y</td>
<td>F</td>
</tr>
<tr>
<td>(3) y := z-7</td>
<td>F</td>
</tr>
<tr>
<td>(2) z := x*5</td>
<td>T</td>
</tr>
<tr>
<td>(1) x := y+z</td>
<td>F</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.

Register & Address Descriptors

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>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>0x2020</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 R0, R1, ..., Rn.
- MOV M, R Load variable M into register R.
- MOV R, M Store register R into variable M.
- OP M, R Compute R := R OP M, where OP is one of ADD, SUB, MUL, DIV.
- OP R2, R1 Compute R1 := R1 OP R2, where OP is one of ADD, SUB, MUL, DIV.

GenCode((i): X := Y 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.

1. L := GetReg(i: X := y 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): \text{X := Y})

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

- \textbf{IF Y only in mem. location \text{L} THEN}:
  - \text{R := GetReg(); gen(MOV Y, R);}
  - \text{AD: Y is now only in reg R.}
  - \text{RD: R now holds Y.}

- \textbf{IF Y is in register \text{R} THEN}:
  - \text{AD: X is now only in register \text{R}.}
  - \text{RD: R now holds X.}
  - \text{IF (i).Y\_next\_use=NONE THEN} \text{RD: No register holds Y.}

- At the end of the basic block:
  - Store all live variables (that are left in registers) in their memory locations.

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

- If we won’t be needing the value stored in \text{Y} after this instruction, we can reuse \text{Y}’s register.

1. \textbf{IF} \text{Y is in register \text{R} and \text{R} holds only \text{Y}}
   - (i).Y\_next\_use=NONE
   - \text{THEN RETURN R;}

2. \textbf{ELSIF} there’s an empty register \text{R} available \text{THEN}
   - \text{RETURN R;}

3. \textbf{ELSIF} \text{X} has a next use and there exists an occupied register \text{R}
   - \text{THEN Store R into its memory location and RETURN R;}

4. \textbf{OTHERWISE RETURN} the memory location of \text{X}.

### Code Generation Example I

<table>
<thead>
<tr>
<th>Interm. Code</th>
<th>Machine</th>
</tr>
</thead>
</table>
| (1) x := y + z | MOV y, r0  
ADD z, r0 |
| (2) z := x \ast 5 | MUL 5, r0 |
| (3) y := z - 7 | MOV r0, r1  
SUB 7, r1 |
| (4) x := z + y | MOV r0, z  
ADD r1, r0 |
|               | MOV r1, y  
MOV r0, x |

Note that \text{x} and \text{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. Code</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 \equiv x</td>
<td>x \equiv r0</td>
<td>T F T</td>
</tr>
<tr>
<td></td>
<td>ADD z, r0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z := x \ast 5</td>
<td>MUL 5, r0</td>
<td>r0 \equiv z</td>
<td>z \equiv r0</td>
<td>F T</td>
</tr>
<tr>
<td>y := z - 7</td>
<td>MOV r0, r1</td>
<td>r0 \equiv z</td>
<td>z \equiv r0</td>
<td>T T</td>
</tr>
<tr>
<td></td>
<td>SUB 7, r1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.