CSc 553

Principles of Compilation

X8 : Code Generation II

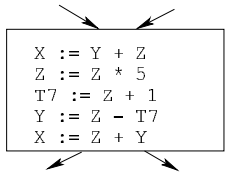
Department of Computer Science
University of Arizona

collberg@gmail.com

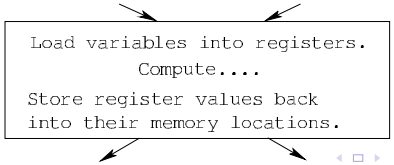
Copyright © 2011 Christian Collberg

# Next-Use Information

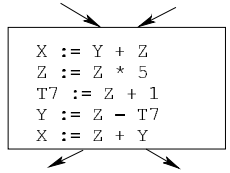
## Basic Block Code Generation




- 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 **live** at (5) _

```
(5)   X := ···
          ...  (no ref to X) ...
(14)  ··· := ··· X ···
```

- 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).

X is **dead** at (12)

```
(12)  ··· := ··· X ···
          ...  (no ref to X) ...
(25)  X := ···
```

- 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

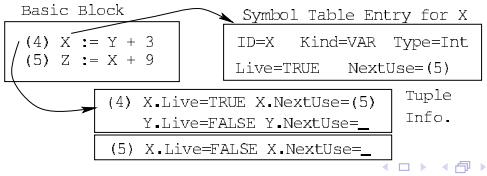
| Intermediate Code | | | | Live/Dead | | | | Next Use | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | x | y | z | $t_7$ | x | y | z | $t_7$ |
| (1) | x | := | y+z | L | D | D | | (2) | _ | _ | |
| (2) | z | := | x*5 | D | | L | | _ | | (3) | |
| (3) | $t_7$ | := | z+1 | | | L | L | | | (4) | (4) |
| (4) | y | := | z-$t_7$ | | L | L | D | | (5) | (5) | _ |
| (5) | x | := | z+y | D | D | D | | _ | _ | _ | |

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

(i).X.live:=(i).Y.live:=(i).Z.live:=FALSE and
(i).X.next_use:=(i).Y.next_use:= (i).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 `(i): x := y 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 Example I – Forward Pass

| | SyTab-Info | | | | | | Instr.-Info | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | live | | | next_use | | | live | | | next_use | | |
| i | x | y | z | x | y | z | x | y | z | x | y | z |
| (1) x:=y+z | F | F | F | | | | F | F | F | | | |
| (2) z:=x*5 | F | F | F | | | | F | F | F | | | |
| (3) y:=z-7 | F | F | F | | | | F | F | F | | | |
| (4) x:=z+y | F | F | F | | | | F | F | F | | | |

## Next-Use Example II – Backwards Pass

| | SyTab-Info | | | | | | Instr.-Info | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | live | | | next_use | | | live | | | next_use | | |
| i | x | y | z | x | y | z | x | y | z | x | y | z |
| (4) x := z+y | F | T | T | | 4 | 4 | F | F | F | | | |
| (3) y := z-7 | F | F | T | | | 3 | F | T | T | | 4 | 4 |
| (2) z := x*5 | T | F | F | 2 | | | F | F | T | | | 3 |
| (1) x := y+z | F | T | T | | 1 | 1 | T | F | F | 2 | | |

- 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

## 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.

| Address Descr. | | |
|---|---|---|
| Id | Memory | Regs. |
| x | fp(16) | {r0} |
| y | fp(20) | {} |
| z | 0x2020 | {r1, r3} |
| t1 | | {r0} |

| Reg. Descr. | |
|---|---|
| Reg | Contents |
| r0 | {x, t1} |
| r1 | {z} |
| r2 | {} |
| r3 | {z} |

# A Simple Code Generator

## 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, ..., R`*n*.

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

## GetReg(i: X := Y op Z)

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

1. **IF**
   - `Y` is in register `R` and `R` holds only `Y`
   - `(i).Y.next_use=NONE`

   **THEN RETURN** `R;`
2. **ELSIF** there's an empty register `R` available **THEN RETURN** `R;`
3. **ELSIF**
   - `X` has a next use and there exists an occupied register `R`

   **THEN** Store `R` into its memory location and **RETURN** `R;`
4. **OTHERWISE RETURN** the memory location of `X`.

## Code Generation Example I

| | Interm. Code | Machine |
|---|---|---|
| (1) | `x := y + z` | `MOV y, r0`<br>`ADD z, r0` |
| (2) | `z := x * 5` | `MUL 5, r0` |
| (3) | `y := z - 7` | `MOV r0, r1`<br>`SUB 7, r1` |
| (4) | `x := z + y` | `MOV r0, z`<br>`ADD r1, r0` |
| | | `MOV r1, y`<br>`MOV r0, x` |

- 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

| Interm. | Machine | RD | AD | Live | | |
|---|---|---|---|---|---|---|
| | | | | x | y | z |
| `x := y + z` | `MOV y, r0`<br>`ADD z, r0` | `r0` ≡ `x` | `x` ≡ `r0` | T | F | T |
| `z := x * 5` | `MUL 5, r0` | `r0` ≡ `z` | `z` ≡ `r0` | F | | T |
| `y := z - 7` | `MOV r0, r1`<br>`SUB 7, r1` | `r0` ≡ `z`<br>`r1` ≡ `y` | `z` ≡ `r0`<br>`y` ≡ `r1` | | T | T |

## Code Generation Example III

| Interm. | Machine | RD | AD | Live | | |
|---|---|---|---|---|---|---|
| x := z + y | MOV r0, z | r0 ≡ z | z ≡ mem | T | T | T |
| | | | z ≡ r0 | | | |
| | | r1 ≡ y | y ≡ r1 | | | |
| | ADD r1, r0 | r0 ≡ x | x ≡ r0 | | | |
| | | r1 ≡ y | y ≡ r1 | | | |
| | | | z ≡ mem | | | |
| | MOV r1, y | | y ≡ mem | | | |
| | MOV r0, x | | x ≡ mem | | | |

# Summary

## Readings and References

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

## Summary I

- 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.