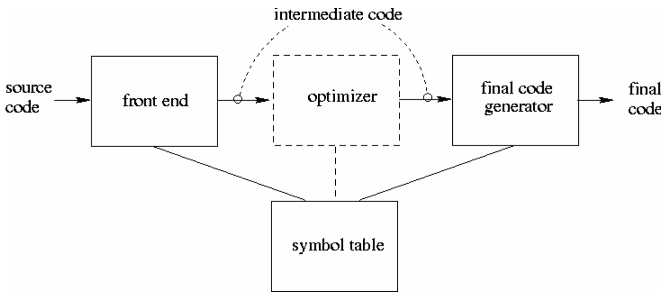
CSc 453
Final Code Generation

Saumya Debray

The University of Arizona

Tucson

Overview



- Input:
  - intermediate code program, symbol table
- Output:
  - target program (asm or machine code).

Issues

- Memory management:
  - map symbol table entries to machine locations (registers, memory addresses);
  - map labels to instruction addresses.
- Instruction selection:

  Peculiarities of the target machine have to be taken into account, e.g.:
  - different kinds of registers, e.g., address, data registers on M68k;
  - implicit register operands in some instructions, e.g., MUL, DIV;
  - branches: addressing modes (PC-relative vs. absolute); span (short vs. long).
- Performance considerations:

  Machine-level decisions (register allocation, instruction scheduling) can affect performance.

Translating 3-address code to final code

Almost a macro expansion process. The resulting code can be improved via various code optimizations.

3-address code	*MIPS assembly code*
`x = A[ i ]`	load **`i`** into *reg1* **`la`** *reg2*, **`A`** **`add`** *reg2, reg2, reg1* **`lw`** *reg2*, (*reg2*) **`sw`** *reg2*, **`x`**
`x = y + z`	load **`y`** into *reg1* load **`z`** into *reg2* **`add`** *reg3, reg1, reg2* **`sw`** *reg3*, **`x`**
`if x ≥ y goto L`	load **`x`** into *reg1* load **`y`** into *reg2* **`bge`** *reg1, reg2*, **`L`**

Storage Allocation

Delay decisions about storage allocation until final code generation phase:

- initialize *location* field of each identifier/temp to UNDEF;
- during code generation, first check each operand of each instruction:
  - if *location* == UNDEF, allocate appropriate-sized space for it (width obtained from type info);
  - update *location* field in its symbol table entry;
  - update information about next unallocated location.

Advantages:

- machine dependencies (width for each type) pushed to back end;
- variables that are optimized away, or which live entirely in registers, don't take up space in memory.

Improving Code Quality 1

Peephole Optimization: traverse the code looking for sequences that can be improved. E.g.:

redundant instruction elimination:

```
    goto L  /* L is next instruction */     ⇒
L:  ...                                          L:  ...
```

control flow optimizations:

```
    goto L1                                 ⇒        goto L2
    ...                                              ...
L1: goto L2                                      L1: goto L2
```

algebraic simplifications, e.g.:

```
x = x+0                                     }
x = x∗1                                     }   eliminate
y = 2∗x                                     ⇒    y = x+x
```

Improving Code Quality 2

Register Allocation: place frequently accessed values in registers.

- *Local register allocation*: simple algorithms that consider only small segments of code ("basic blocks").
- *Global register allocation*: algorithms that consider the entire body of a function.

  These are more complex, but are able to keep variables in registers over larger code fragments, e.g., over an entire loop.

Good global register allocation can reduce runtime by ~20–40% (Chow & Hennessy 1990).

Improving Code Quality 3

Code Optimization:

- Examine the program to identify specific program properties ("*dataflow analysis*").
- Use this information to change the code so as to improve its performance. E.g.:
  - invariant code motion out of loops
  - common subexpression elimination
  - dead code elimination

Improving Code Quality 4

Instruction Scheduling: Some instructions take many cycles to execute.

On modern architectures, this can cause the instruction pipeline to be blocked ("stalled") for several cycles.

Instruction scheduling refers to choosing an execution order on the instructions that allows useful work to be done by the CPU while it is waiting for an expensive operation to complete.

Improving Code Quality 5

Memory Hierarchy Optimizations:

- Modern processors typically use a multi-level memory hierarchy (cache, main memory).

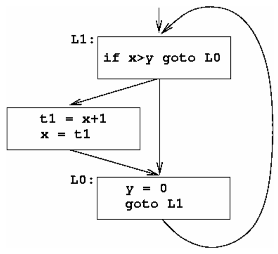
  Accessing main memory can be very expensive.
- Careful code layout can improve instruction cache utilization:
  - uses execution frequency information;
  - reduces cache conflicts between frequently executed code.

Basic Blocks and Flow Graphs

- For program analysis and optimization, we need to know the program's control flow behavior.
- For this, we:
  - group three-address instructions into *basic blocks*;
  - represent control flow behavior using *control flow graphs*.

Example:

```
L1: if x > y goto L0
    t1 = x+1
    x = t1
L0: y = 0
    goto L1
```



Basic Blocks

- *Definition*: A basic block *B* is a sequence of consecutive instructions such that:
  1. control enters *B* only at its beginning;
  2. control leaves *B* at its end (under normal execution); and
  3. control cannot halt or branch out of *B* except at its end.
- This implies that if any instruction in a basic block B is executed, then *all* instructions in B are executed.
  - ⇒ for program analysis purposes, we can treat a basic block as a single entity.

Identifying Basic Blocks

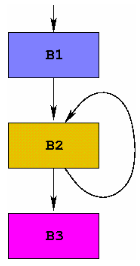
1. Determine the set of *leaders*, i.e., the first instruction of each basic block:
   - the entry point of the function is a leader;
   - any instruction that is the target of a branch is a leader;
   - any instruction following a (conditional or unconditional) branch is a leader.
2. For each leader, its basic block consists of:
   - the leader itself;
   - all subsequent instructions upto, but not including, the next leader.

Control Flow Graphs

- *Definition*: A control flow graph for a function is a directed graph $G = (V, E)$ such that:
  - each $v \in V$ is a basic block; and
  - there is an edge $a \rightarrow b \in E$ iff control can go directly from a to b.
- *Construction*:
  1. identify the basic blocks of the function;
  2. there is an edge from block a to block b if:
     i. there is a (conditional or unconditional) branch from the last instruction of a to the first instruction of b; or
     ii. b immediately follows a in the textual order of the program, and a does not end in an unconditional branch.

Example

```
int dotprod(int a[], int b[], int N)
{
   int i, prod = 0;
   for (i = 1; i ≤ N; i++) {
      prod += a[i]∗b[i];
   }
   return prod;
}
```

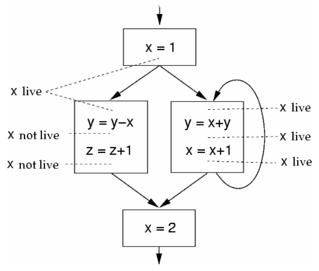


No.	Instruction	leader?	Block No.
1	`enter dotprod`	Y	1
2	`prod = 0`		1
3	`i = 1`		1
4	`t1 = 4*i`	Y	2
5	`t2 = a[t1]`		2
6	`t3 = 4*i`		2
7	`t4 = b[t3]`		2
8	`t5 = t2*t4`		2
9	`t6 = prod+t5`		2
10	`prod = t6`		2
11	`t7 = i+i`		2
12	`i = t7`		2
13	`if i ≤ N goto 4`		2
14	`retval prod`	Y	3
15	`leave dotprod`		3
16	`return`		3

Program Analysis Example: Liveness

Definition: A variable is live at a program point if it *may* be used at a later point before being redefined.

Example:



Example application of liveness information:

if **x** is in a register *r* at a program point, and **x** is not live, then *r* can be freed up for some other variable without storing **x** back into memory.

Liveness Analysis: Overview

- For each basic block, identify those variables that are:
  - (possibly) used before being redefined;
  - (definitely) redefined before being used.
- Propagate this information along control flow graph edges.
  - propagate iteratively until no change;
  - amounts to iterative solution of a set of equations;
  - solution gives, at each point, the set of variables that could be used before being redefined.