Module 11: CS-1319-1: Programming Language Design and Implementation (PLDI)

Simple Code Generators

Partha Pratim Das

Department of Computer Science
Ashoka University

ppd@ashoka.edu.in, partha.das@ashoka.edu.in, 9830030880

Self-Study

Module Objectives

- Code Generation – Main Issues
- Samples of Generated Code
- Two Simple Code Generators
- Optimal Code Generation
  - Sethi-Ullman Algorithm
- Peephole Optimization

Module Outline

Code Generation – Main Issues (1)

- Transformation
  - Intermediate code $\rightarrow$ m/c code (binary or assembly)
  - We assume that quads, CFG and ST are available
- Which instructions to generate?
  - For the quadruple `A = A+1`, we may generate:

    ```
    Inc A
    ```

    or

    ```
    Load A, R1
    Add #1, R1
    Store R1, A
    ```

  - One sequence is faster than the other
    - Cost implication

Code Generation – Main Issues (2)

- In which order?
  - Some orders may use fewer registers and/or may be faster
- Which registers to use?
  - Optimal assignment of registers to variables is difficult to achieve
- Optimize for memory, time or power?
- Is the code generator easily re-target-able to other machines?
  - Can the code generator be produced automatically from specifications of the machine?

Samples of Generated Code

- B = A[i]
```
Load   i, R1 // R1 = i
Mult   R1, 4, R1 // R1 = R1 * 4
// each element of array
// A is 4 bytes long
Load   A(R1), R2 // R2 = (A + R1)
Store  R2, B // B = R2
```
- X[j] = Y
```
Load  Y, R1 // R1 = Y
Load  j, R2 // R2 = j
Mult  R2, 4, R2 // R2 = R2 * 4
Store  R1, X(R2)// (X + R2) = R1
```
- X = *p
```
Load   p, R1
Load   0(R1), R2 // R2 = (0 + R1)
Store  R2, X
```
- *q = Y
```
Load   Y, R1
Load   q, R2
Store  R1, 0(R2) // (0 + R2) = R1
```
- if X < Y goto L
```
Load  X, R1
Load  Y, R2
Cmp   R1, R2
Bltz  L // Branch on less than 0
```

A Simple Code Generator: Scheme A

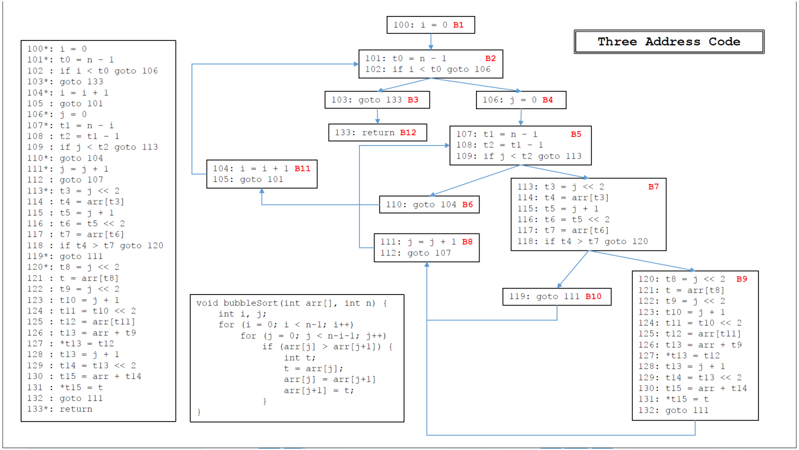
- Treat each quadruple as a *macro*
  - Example: The quad `A := B + C` will result in:

```
Load B, R1     OR    Load B, R1
Load C, R2
Add R2, R1           Add C, R1
Store R1, A          Store R1, A
```

  - Results in inefficient code
    - ▷ Repeated load/store of registers
  - Very simple to implement

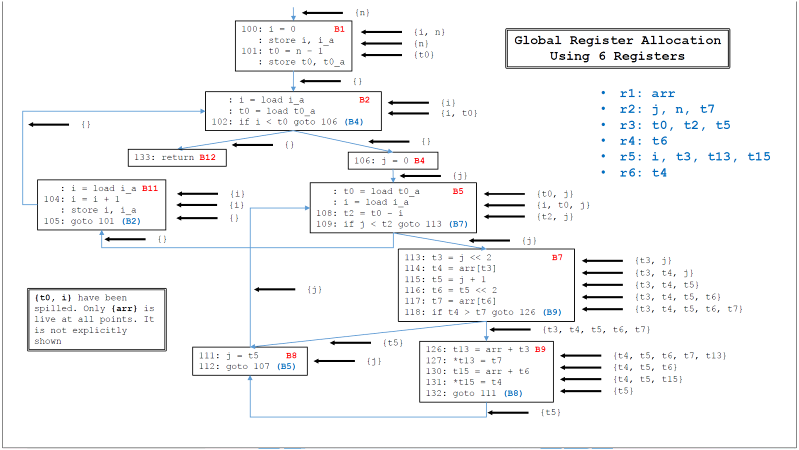
Sample Code Generation: Bubble Sort
Three Address Code

- Three Address Code for Bubble Sort as generated by syntax directed translation



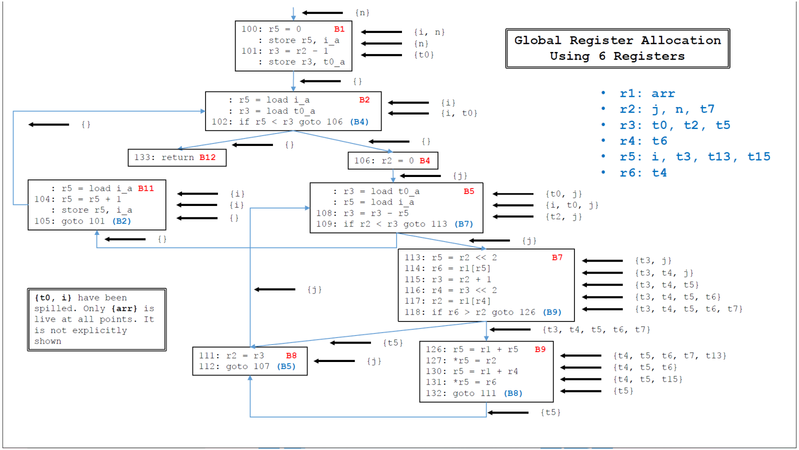
Sample Code Generation: Bubble Sort Liveness after LCSE, GCSE Optimization

- Three Address Code Optimized by peephole, LCSE by VN and GCSE by DFA. Finally, live variables are computed by DFA



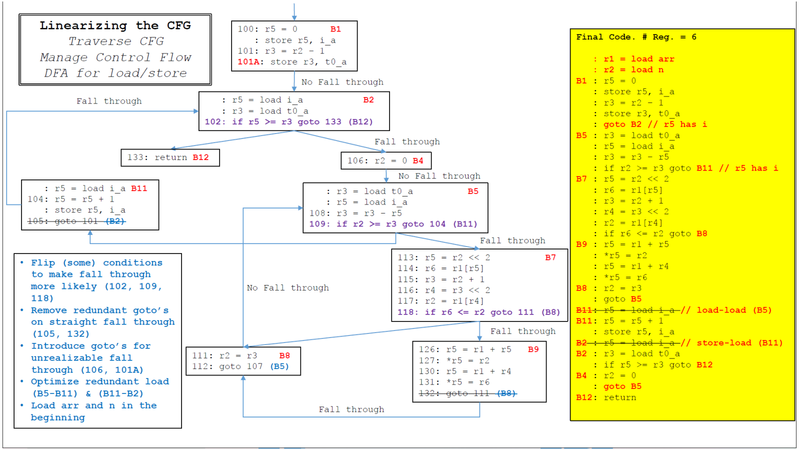
Sample Code Generation: Bubble Sort Global Register Allocation

- Registers are allocated globally using graph coloring based on the liveness information
- Variables are replaced by respective registers



Sample Code Generation: Bubble Sort Linearized and Optimized Target Code

- The CFG is linearized and further optimized to get the final target code



A Simple Code Generator: Scheme B

- Track values in registers and reuse them
  - If any operand is already in a register, take advantage of it
  - Register Descriptors
    - Tracks <register, variable name> pairs
    - A single register can contain values of multiple names, if they are all copies
  - Address Descriptors
    - Tracks <variable name, location> pairs
    - A single name may have its value in multiple locations, such as, memory, register, and stack

A Simple Code Generator: Scheme B

- Leave computed result in a register as long as possible
- Store only at the end of a basic block or when that register is needed for another computation
  - On exit from a basic block, store only live variables which are not in their memory locations already (use address descriptors to determine the latter)
  - If liveness information is not known, assume that all variables are live at all times

Example

A := B+C

- If B and C are in registers R1 and R2, then generate
  - `ADD  R2, R1 (cost = 1, result in R1)`
    - legal only if B is not live after the statement
- If R1 contains B, but C is in memory
  - `ADD  C, R1 (cost = 2, result in R1)`

    or
  - `LOAD C, R2`

    `ADD  R2, R1 (cost = 3, result in R1)`
    - legal only if B is not live after the statement
    - attractive if the value of C is subsequently used (it can be taken from R2)

Next Use Information

- Next use info is used in code generation and register allocation
- Next use of A in quad i is j if

  Quad i : A = ... (assignment to A)

  $\Downarrow$ (control flows from i to j with no assignments to A)

  Quad j : = A op B (usage of A)
- In computing next use, we assume that on exit from the basic block
  - All temporaries are considered non-live
  - All programmer defined variables (and non-temps) are live
- Each procedure/function call is assumed to start a basic block
- Next use is computed on a backward scan on the quads in a basic block, starting from the end
- Next use information is stored in the symbol table

Example of computing Next Use

3	T1 := 4 * I	T1: (nlv, lu 0, nu 5), I: (lv, lu 3, nu 10)
4	T2 := addr(A) – 4	T2: (nlv, lu 0, nu 5), A: (lv, lu 4, nnu)
5	T3 := T2[T1]	T3: (nlv, lu 0, nu 8), T2: (nlv, lu 5, nnu),
		T1: (nlv, lu 5, nu 7)
6	T4 := addr(B) – 4	T4: (nlv, lu 0, nu 7), B: (lv, lu 6, nnu)
7	T5 := T4[T1]	T5: (nlv, lu 0, nu 8), T4: (nlv, lu 7, nnu),
		T1: (nlv, lu 7, nnu)
8	T6 := T3 * T5	T6: (nlv, lu 0, nu 9),T3: (nlv, lu 8, nnu),
		T5: (nlv, lu 8, nnu)
9	PROD := PROD + T6	PROD: (lv, lu 9, nnu), T6: (nlv, lu 9, nnu)
10	I := I + 1	I: (lv, lu 10, nu 11)
11	if I = 20 goto 3	I: (lv, lu 11, nnu)

nlv: not live lv: live lu: last use nu: next use nnu: no next use lu 0: no last use

Scheme B – Algorithm

- We deal with one basic block at a time
- We assume that there is no global register allocation
- For each quad A := B op C do the following
  - Find a location L to perform B op C
    - Usually a register returned by GETREG() (could be a mem loc)
  - Where is B?
    - B', found using address descriptor for B
    - Prefer register for B', if it is available in memory and register
    - Generate Load B' , L (if B' is not in L)
  - Where is C?
    - C', found using address descriptor for C
    - Generate op C' , L
  - Update descriptors for L and A
  - If B/C have no next uses, update descriptors to reflect this information

Function *GETREG*()

Finds L for computing A := B op C

[1] If B is in a register (say R), R holds no other names, and

- B has no next use, and B is not live after the block, then return R

[2] Failing (1), return an empty register, if available

[3] Failing (2)

- If A has a next use in the block, OR
  if B op C needs a register (e.g., op is an indexing operator)
  - Use a heuristic to find an occupied register R
    - **a register whose contents are referenced farthest in future, or**
    - **the number of next uses is smallest etc.**
  - Spill it by generating an instruction, *MOV R, mem*
    - *mem* **is the memory location for the variable in R**
    - **That variable is not already in** *mem*
  - Update Register and Address descriptors

[4] If A is not used in the block, or no suitable register can be found

- Return a memory location for L

Example

- T,U, and V are temporaries – not live at the end of the block
- W is a non-temporary – live at the end of the block, 2 registers
- Using two registers R0 and R1

Statements	**Code Generated**	**Register Descriptor**	**Address Descriptor**
T := A * B	Load A,R0 Mult B, R0	R0 contains T	T in R0
U := A + C	Load A, R1 Add C, R1	R0 contains T R1 contains U	T in R0 U in R1
V := T - U	Sub R1, R0	R0 contains V R1 contains U	U in R1 V in R0
W := V * U	Mult R1, R0	R0 contains W	W in R0
	Store R0, W		W in memory (restored)

Optimal Code Generation: The Sethi-Ullman Algorithm

- Generates the shortest sequence of instructions
  - Provably optimal algorithm (w.r.t. length of the sequence)
- Suitable for expression trees (basic block level)
- Machine model
  - All computations are carried out in registers
  - Instructions are of the form $op\ R_s,\ R_t$ or $op\ M_s,\ R_t$
- *Always computes the left subtree into a register and reuses it immediately*
- Two phases
  - Labelling phase
  - Code generation phase

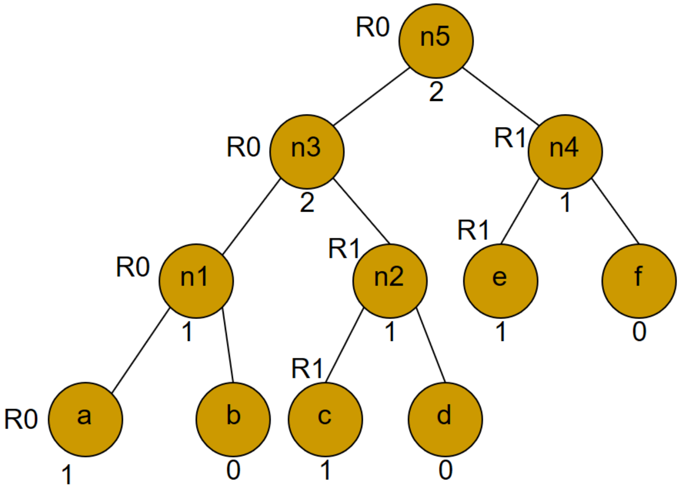
The Labelling Algorithm

- Label each node of the tree with an integer:
  - Consider binary trees
  - Fewest no. of registers required to evaluate the tree with no intermediate stores to memory
- For leaf nodes
  - if n is the leftmost child of its parent then

    $$label(n) := 1 \; else \; label(n) := 0$$
- For internal nodes

$$\begin{aligned} label(n) &= max(l_1, l_2), if \; l_1 \neq l_2 \\ &= l_1 + 1, if \; l_1 = l_2 \end{aligned}$$

Labelling – Example

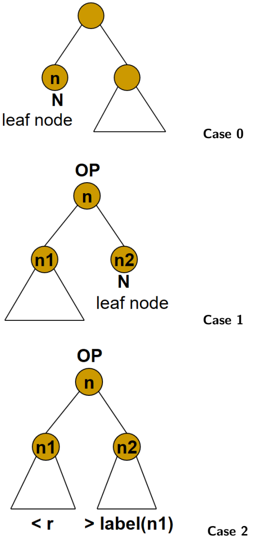


Code Generation Phase: Procedure GENCODE(n)

- RSTACK – stack of registers, R0, ..., R(r-1)
- TSTACK – stack of temporaries, T0, T1, ...
- A call to Gencode(n) generates code to evaluate a tree T, rooted at node n, into the register top(RSTACK), and
  - the rest of RSTACK remains in the same state as the one before the call
- A swap of the top two registers of RSTACK is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child

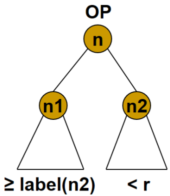
The Code Generation Algorithm (Cases 0-1-2)


```
Procedure gencode(n);
{ /* case 0 */
    if
        n is a leaf representing operand N
        and is the leftmost child of its parent
    then
        print(LOAD N, top(RSTACK))
    /* case 1 */
    else if
        n is an interior node with operator
        OP, left child n1, and right child n2
    then
    if  label(n2) == 0 then {
        let N be the operand for n2;
        gencode(n1);
        print(OP N, top(RSTACK));
    }
    /* case 2 */
    else if ((1 <= label(n1) < label(n2))
             and (label(n1) < r))
    then {
        swap(RSTACK); gencode(n2);
        R := pop(RSTACK); gencode(n1);
        /* R holds the result of n2 */
        print(OP R, top(RSTACK));
        push (RSTACK,R);
        swap(RSTACK);
        /* The swap() function ensures that a
        node is evaluated into the same
        register as its left child */
    }
```

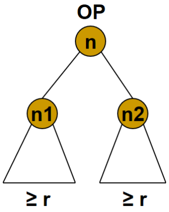


The Code Generation Algorithm (Cases 3-4)

```
        /* case 3 */
        else if ((1 <= label(n2) <= label(n1))
                and (label(n2) < r))
        then {
            gencode(n1);
            R := pop(RSTACK); gencode(n2);
            /* R holds the result of n1 */
            print(OP  top(RSTACK), R);
            push (RSTACK,R);
        }
        /* case 4, both labels are > r */
        else {
            gencode(n2); T:= pop(TSTACK);
            print(LOAD top(RSTACK), T);
            gencode(n1);
            print(OP T, top(RSTACK));
            push(TSTACK, T);
        }
    }
```



Case 3

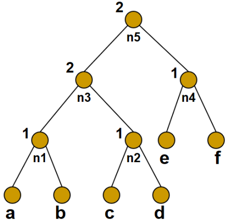


Case 4

Code Generation Phase – Examples

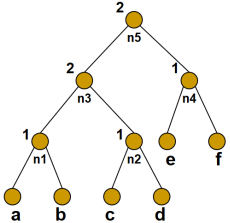

```
No. of registers = r = 2

n5 -> n3 (case 3)
->  n1  -> a (case 3)
->  Load a, R0, op_n1 b, R0
->  n2  -> c     -> Load c, R1
                 -> op_n2 d, R1
                 -> op_n3 R1, R0
->  n4   -> e     -> Load e, R1
                 -> op_n4 f, R1
->  op_n5 R1, R0
```




```
No. of registers = r = 1.
Here we choose rst first so that lst can be
computed into R0 later (case 4)

n5  -> n4 -> e -> Load e, R0
            -> op_n4  f, R0
    -> Load R0, T0 {release R0}
    -> n3   -> n2   -> c     -> Load c, R0
                             -> op_n2 d, R0
            -> Load R0, T1 {release R0}
            -> n1   -> a     -> Load a, R0
                             -> op_n1 b, R0
            -> op_n3 T1, R0 {release T1}
    -> op_n5 T0, R0 {release T0}
```



Peephole Optimizations

- Simple but effective local optimization
- Usually carried out on machine code, but intermediate code can also benefit from it
- Examines a sliding window of code (peephole), and replaces it by a shorter or faster sequence, if possible
- Each improvement provides opportunities for additional improvements
- Therefore, repeated passes over code are needed

Peephole Optimizations

- Some well known peephole optimizations
  - eliminating redundant instructions
  - eliminating unreachable code
  - eliminating jumps over jumps
  - algebraic simplifications
  - strength reduction
  - use of machine idioms

Elimination of Redundant Loads and Stores

Basic block B

```
Load X, R0
{no modifications
to X or R0 here}
Store R0, X
```

Store instruction can be deleted

Basic block B

```
Load X, R0
{no modifications
to X or R0 here}
Load X, R0
```

Second Load instr can be deleted

Basic block B

```
Store R0, X
{no modifications
to X or R0 here}
Load X, R0
```

Load instruction can be deleted

Basic block B

```
Store R0, X
{no modifications
to X or R0 here}
Store R0, X
```

Second Store instr can be deleted

Eliminating Unreachable Code

- An unlabeled instruction immediately following an unconditional jump may be removed
  - May be produced due to debugging code introduced during development
  - Or due to updates to programs (changes for fixing bugs) without considering the whole program segment

Eliminating Unreachable Code

```
    if print == 1 goto L1
    goto L2
L1: print instructions
L2:
```

```
    if print != 1 goto L2
    print instructions
L2:
```

print initialized to 0 at the beginning of the program

```
    if 0 != 1 goto L2
    print instructions
L2:
```

```
    goto L2
    print instructions
L2:
```

```
    goto L2
    ...
L2:
```

print instructions are now unreachable and hence can be eliminated

Flow-of-Control Optimizations

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

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

No jumps to L1

```
goto L2
...
...
```

Statement L1: ... can be removed only if it is preceded by an unconditional jump

```
if a<b goto L1
...
L1: goto L2
...
```

```
if a<b goto L2
...
L1: goto L2
...
```

always executes “goto L1”

```
goto L1
...
L1: if a<b goto L2
L3:
...
```

Only one jump to L1, L1 is preceded by an unconditional goto

sometimes skips “goto L3”

```
if a<b goto L2
goto L3
...
L3:
...
```

Reduction in Strength and Use of Machine Idioms

- x^2 is cheaper to implement as $x * x$, than as a call to an exponentiation routine
- For integers, $x * 2^3$ is cheaper to implement as $x << 3$ (x left-shifted by 3 bits)
- For integers, $x/2^2$ is cheaper to implement as $x >> 2$ (x right-shifted by 2 bits)

Reduction in Strength and Use of Machine Idioms

- Floating point division by a constant can be approximated as multiplication by a constant
- Auto-increment and auto-decrement addressing modes can be used wherever possible
  - Subsume INCREMENT and DECREMENT operations (respectively)
- Multiply and add is a more complicated pattern to detect