MODULE 32 – DAG based Code Generation and Dynamic Programming

In this module, we would try to understand the code generation algorithm from DAG after the DAG has been reordered and labeled. As another approach to code generation, we will discuss the Dynamic programming approach to code generation.

32.1 Code generation from DAG

As discussed in the previous module, one of the ideas behind the DAG construction is code generation. The steps involved include reordering of the instructions, labeling the nodes with the number of registers required and use this information to generate target assembly language code.

The code generation algorithm uses a recursive procedure on a labeled DAG. Considers code generation based on the labels assigned to the nodes. It uses two stacks, one register stack "rstack" and another memory stack, "mstack". Stack "rstack" is used to allocate registers. Initially rstack contains all available registers. The algorithm retains the registers on rstack in the same order it has found them. The typical functions of the stack, like push(), pop() is used to rearrange the rstack and in addition, the algorithm uses a swap(rstack) function to interchange the top two registers on rstack.

The algorithm, considers five different cases to generate code. They are discussed as follows:

- **Case 0:** This is a simple and terminating case of the recursive procedure. If, 'n' is a leaf and the leftmost child of its parent, we generate just a load instruction.
- **Case 1:** This is the situation when the right node is a leaf and the left node could be a sub-tree. In this case, we generate code to evaluate n_1 into register R=top(rstack) followed by the instruction "op name R".
- **Case 2:** The right sub-tree requires more registers than the left sub-tree. A sub-tree of the form where n_1 can be evaluated without stores but n_2 is harder to evaluate than n_1 as it requires more registers. For this case, swap the top two registers on rsatck, then evaluate n_2 into R=top(rstack).We remove R from rstack and evaluate n_1 into S = top(rstack). Then we generate the instruction "op R, S", which produce the value of "n" in register S. Another call to swap leaves rstack as it was, upon this call code generation begins.
- **Case 3**: It is similar to case 2 except that here the left sub-tree is harder and is evaluated first. There is no need to swap registers here.
- **Case 4:** It occurs when both sub-trees require r or more registers to evaluate without stores. Since we must use a temporary memory location, we first evaluate the right sub-tree into the temporary T, then the left sub-tree, and finally the root.

All these cases are discussed in Algorithm 32.1 to generate code from the DAG and the algorithm is named gencode(n) where 'n' is the root of the DAG which is passed as argument

```
Procedure gencode(n);
Begin
/* case 0 */
if n is a left leaf representing operand name and n is the leftmost child of its parent then
        print 'MOV' || name || '.' || top(rstack)
else if n is an interior node with operator op, left child n1, and right child n2 then
/* case 1 */
        if label(n2) = 0 then begin
              let name be the operand represented by n2;
               gencode(n1);
              print op || name || '.' || top(rstack)
        end
/* case 2 */
       else if 1 ≤ label (n1) < label(n2) and label(n1) < r then begin
              swap(rstack);
              gencode(n2 );
              R := pop(rstack); /* n2  was evaluated into register R */
              gencode(n1);
              print op || R || '.' || top(rstack);
              push(rstack,R);
              swap(rstack)
        end
/* case 3 */
        else if 1 ≤ label (n2) < label(n1) and label(n2) < r then begin
              gencode(n1);
              R := pop(rstack); /* n1  was evaluated into register R */
              gencode(n2);
              print op || R || '.' || top(rstack);
              push(rstack,R);
         end
/* case 4, both labels ≥ r, the total number of registers */
       else begin
              gencode(n2 );
              T := pop(tstack);
              Print 'MOV' || top(rstack) || '.' || T;
              gencode(n1);
              push(rstack,R);
              print op || T || '.' || top(rstack)
      end
end
```

Case 1 checks if the right child of an interior node is a leaf node. If it is a leaf node, then we call the function recursively with the left child as the root node. To evaluate and finally conclude this case with generating an "op" instruction. Case 2 is evaluated if the right sub-tree is heavy. If it is heavy, we swap the register stack so that the right sub—tree is evaluated into the register which

is beneath the top register. We then recursively call gencode() function with the right sub-tree's node as root and we remove this register from rstack. We then call gencode() to evaluate the left sub-tree and use the top of the stack register. After that we swap the rstack contents to ensure the initial rstack content is retained. Case 3 is just the opposite of Case 2 and since in this context, we evaluate first the left sub-tree, the rstack contents are used as it is and not swapped. Case 4 is the situation when there are no registers in the rstack. We use a memory based operation where the operands would be memory to compute.

Consider the DAG given in figure 32.1 as an example for code generation.

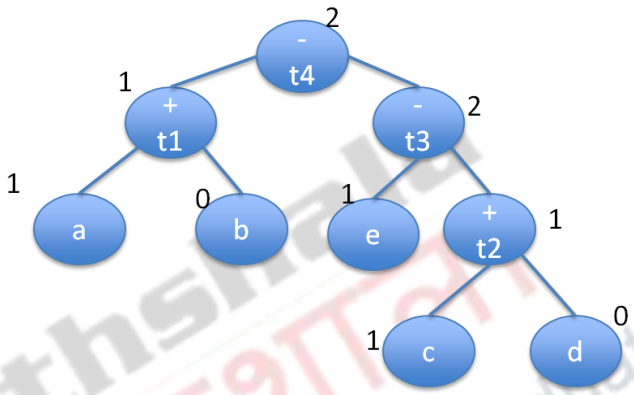

Figure 32.1 Example DAG for code generation

The nodes of the DAG are labeled with the number of registers that it requires for computation. Assume that there are two register R0 and R1 with R0 on the top of the stack. The algorithm gencode() is called with the root node t4. The children of this node are t1 and t3. Since the label of t3 is greater, case 2 is initiated. This calls recursively gencode() with t3 after swapping the register stack. This results in its left child being a leaf node and hence falls under case 0. Case 0 is a load instruction and the value 'e' is loaded into R1. After this call returns it goes to the next step of the previous call, which removes the register R1 from rstack using the pop() command and the gencode function is called with the right sub-tree node, which is t2. This falls under case 1 as the label of the right leaf node is 0 and thus gencode is again called with node 'c'. This again falls under case 0 and initiates a load instruction into R0 and returns. The next instruction is the next step of Case1 where an operator instruction is issued followed by the next instruction of case 2 where a SUB instruction is issued and the register is pushed and then swapped. Proceeding in a similar fashion we get the following code.

Code

```
gencode(t4)                          [R1 R0]      // case 2
   gencode(t3)                       [R0 R1]      // case 3
      gencode(e)                     [R0R1]       // case 0
         print MOV e, R1
      gencode(t2)                    [R0]         // case 1
```

```
        gencode(c)                          [R0]          // case 0
           print MOV c, R0
                              print ADD d, R0
        print SUB R0, R1
gencode(t1)                                      [R0]          // case 1
    gencode(a)                                   [R0]          // case 0
        print MOV a, R0
    print ADD b, R0
print SUB R1, R0
```

32.2 Multi-register operation

The code generation algorithm gencode() discussed uses only one register by default. The algorithm could be made to use two registers for regular operations by changing the labeling algorithm as discussed in the previous module as follows:

$$label\ (n) = \begin{cases} \max(2, l1, l2) if\ l1 != l2 \\ l1 + 1\ if\ l1 = l2 \end{cases}$$

The algorithm could be used to exploit the algebraic properties where we could swap left and right nodes effectively to use the code generation algorithm. This will also avoid recomputation of common sub-expression.

32.3 Dynamic programming

Constructing a DAG for code generation creates one more approach to code generation. We can adopt a bottom-up approach to compute the cost of evaluating each node using a dynamic programming approach. Dynamic programming is an algorithmic design strategy where all possible directions are explored and the least cost is chosen for computing at every point of time. For code generation, in order to compute the code for each node, all possibilities in terms of instruction cost is evaluated and the least cost to compute each node is used. For each node 'n' of the expression tree T an array *C* of costs, in which the ith component C[i] is the optimal cost of computing the sub-tree S rooted at 'n' into a register, assuming i registers are available for the computation

After computing the cost vector, we traverse the tree T, and use the cost vectors to determine which sub-trees of T must be computed into memory or register. Traverse each tree using the cost vectors and associated instructions to generate the final target code. The code for the sub-trees computed into memory locations is generated first.

To consider the dynamic programming approach to code generation the following assumptions are made:

- There are only two registers available for computation
- Cost of computing a node which is in memory involves 0
- The following instructions are alone permitted where the LHS corresponds to target and the RHS corresponds to the source.
  - Ri := Mj - Loads the memory content Mj into register Ri
  - Ri := Ri op Rj – operates Ri and Rj and the result is available in Ri

- Ri := Ri op Mj - operates Ri and Mj and the result is available in Ri
- Ri := Rj – Loads the register content Rj into register Ri
- Mi := Ri – Loads the register content Ri into memory Mi

Consider an example expression (a-b)+c*(d/e). The syntax tree for (a-b)+c*(d/e) is shown in figure 32.2 with cost vector at each node

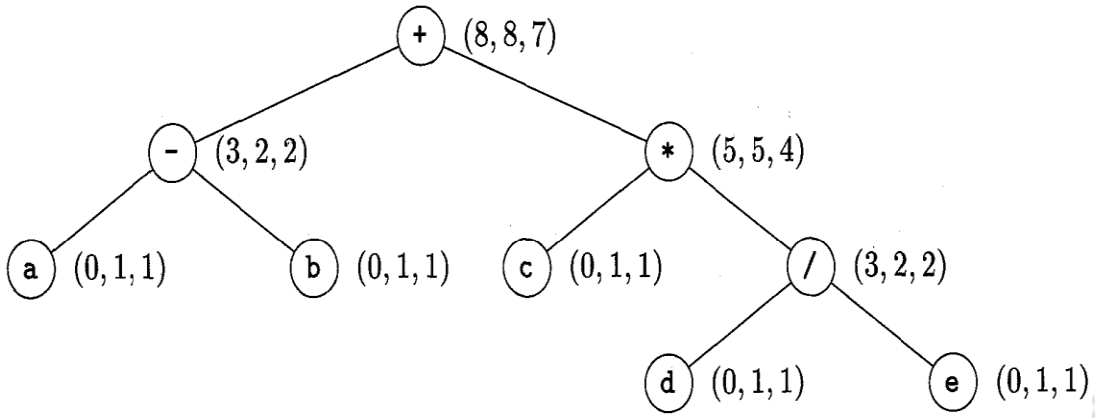

Figure 32.2 Syntax tree for the example expression.

Each node has a cost vector that has 3 values which indicate the cost of computing that node with 0, 1 and 2 registers. The computation for various cases is detailed below:

- Leaf nodes – All leaf nodes have the same cost vector (0, 1, 1)
  - Cost of moving a variable with no registers is 0 – variable in register itself
  - Cost of moving a variable with 1/ 2 registers is 1 using the instruction Ri : = Mj
  - Cost of leaf nodes are just (0, 1, 1) using no register, 1 register, 2 registers

- Last but one node – find all possible and choose the least cost
  - Computing this using 0 registers,
    - Ri := Mj (costs 1)
    - Ri := Ri op Mj (costs 1)
    - Mj : = Ri (costs 1)

    } Total cost = 3

    (or)

  - Ri := Mj (costs 1)
  - Rj := Mi (costs 1)}
  - Ri := Ri op Rj (costs 1)
  - Mj : = Ri (costs 1)

  } Total cost = 4

- Computing this using one register
  - Ri := Mj (costs 1)
  - Ri := Ri op Mj (costs 1)

  Total cost = 2

  (or)

  - Ri := Mj (costs 1)
  - Rj := Mi (costs 1)}
  - Ri := Ri op Rj (costs 1)
  - Mj : = Ri (costs 1)

  Total cost = 4

- Computing this using two registers
  - Ri := Mj (costs 1)
  - Ri := Ri op Mj (costs 1)

  Total cost = 2

  (or)

  - Ri := Mj (costs 1)
  - Rj := Mi (costs 1)}
  - Ri := Ri op Rj (costs 1)
  - Mj : = Ri (costs 1)

  Total cost = 4

- Cost vector of the last but one level node is (3, 2, 2) as this value is the minimum based on individual node computation

- Computing the * node.
  - Using 0 registers

    Ri = Ri op Mj (costs 1)
    Mj = Ri (costs 1)

    The cost to compute the children nodes of * is 1 + 3. Thus resulting in 6.

    (or)

    Ri = Ri op Rj (costs 1)
    Mj = Ri (costs 1)

    The cost to compute the children nodes of * is 1+2. Thus resulting in a total of 5 and between 5 and 6 we choose 5 as the cost of the first quantity.

  - Using 1 registers

    Ri = Ri op Mj (costs 1)

    The cost to compute the children nodes of * is 1 + 3. Thus resulting in 5 as the instruction Ri op Mj is supported which expects the RHS node to be in memory.

- Using 2 registers

  Ri = Ri op Rj (costs 1)

  The cost to compute the children nodes of * is 1+2. If the computation involves two registers it could be computed using one register also. Thus resulting in a total of 4 for two registers and between 5 and 4 we choose 4 as the cost of the third cost vector C.

The following is the logic for the computation of the cost vector for the root.

- Compute the left subtree with two registers available into register R0, compute the right subtree with one register available into register R1, and use the instruction ADD R0, R0, R1 to compute the root. This sequence has cost 2+5+1=8.
- Compute the right subtree with two registers available into R l , compute the left subtree with one register available into R0, and use the instruction ADD R0, R0, R1. This sequence has cost 4+2+1=7.
- Compute the right subtree into memory location M, compute the left subtree with two registers available into register RO, and use the instruction ADD R0, R0, M. This sequence has cost 5+2+1=8

This tree is traversed from bottom to top to generate code as follows:

- R0 := c – the value is a memory based operation to move ‘c’ into R0
- R1 := d – same as ‘c’ and we use another register to move ‘d’ into R1
- R1 := R1/e – Only one register is used to compute this node and the result is in R1
- R0 := R0 * R1 – Two registers resulted in a lower cost and we use that
- R1 := a – R1 is free and hence ‘a’ is loaded to R1
- R1 := R1 – b –The last but one node computation involving only one register
- R1:= R1 + R0 - The root involving two registers.

With a wide set of instructions we could use all combinations and arrive at the optimum cost of computing every node in a very effective manner.

Summary: In this module we discussed the DAG based approach to code generation which involves calling recursively the gencode() algorithm using 5 cases. We also looked at the dynamic programming approach to code generation involving cost vector computation at every node and using a bottom-up strategy to accumulate the cost to compute the root node. In the next module, we would look at a template based approach to code generation.