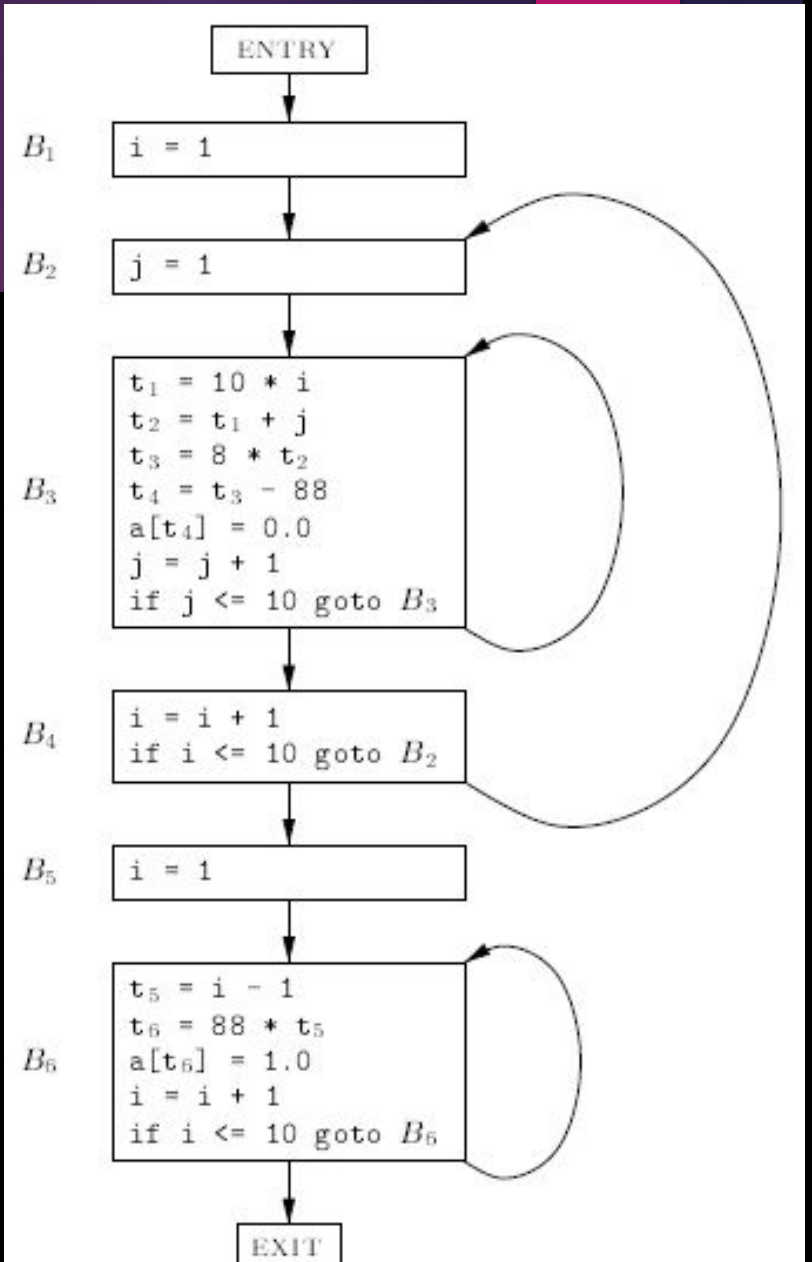
Code Generation

SINI ANNA ALEX

Loops

Loops of the Flow Graph

1. B_3 by itself.
2. B_6 by itself.
3. $\{B_2, B_3, B_4\}$.



Next Use Information

- Knowing when the value of a variable will be used next is essential for generating good code. If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.
- Wish to determine for each three-address statement x = y + z what the next uses of x, y, and z are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement.
- There is an algorithm to determine liveness and next-use information makes a backward pass over each basic block.
- We store the information in the symbol table.

Next Use Information

Algorithm : Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block B of three-address statements. We assume that the symbol table initially shows all nontemporary variables in B as being live on exit.

OUTPUT: At each statement i: $x = y + z$ in B, we attach to i the liveness and next-use information of x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: $x = y + z$ in B, we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
2. In the symbol table, set x to "not live" and "no next use."
3. In the symbol table, set y and z to "live" and the next uses of y and z to i.

Optimization of Basic Blocks

- We can often obtain a substantial improvement in the running time of code merely by performing local optimization within each basic block by itself.
- More thorough global optimization, which looks at how information flows among the basic blocks of a program.

The DAG Representation of Basic Blocks

- Construct a DAG for a basic block as follows:
  - 1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
  - 2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
  - 3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.
  - 4. Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block; that is, their values may be used later, in another block of the flow graph.

The DAG Representation of Basic Blocks

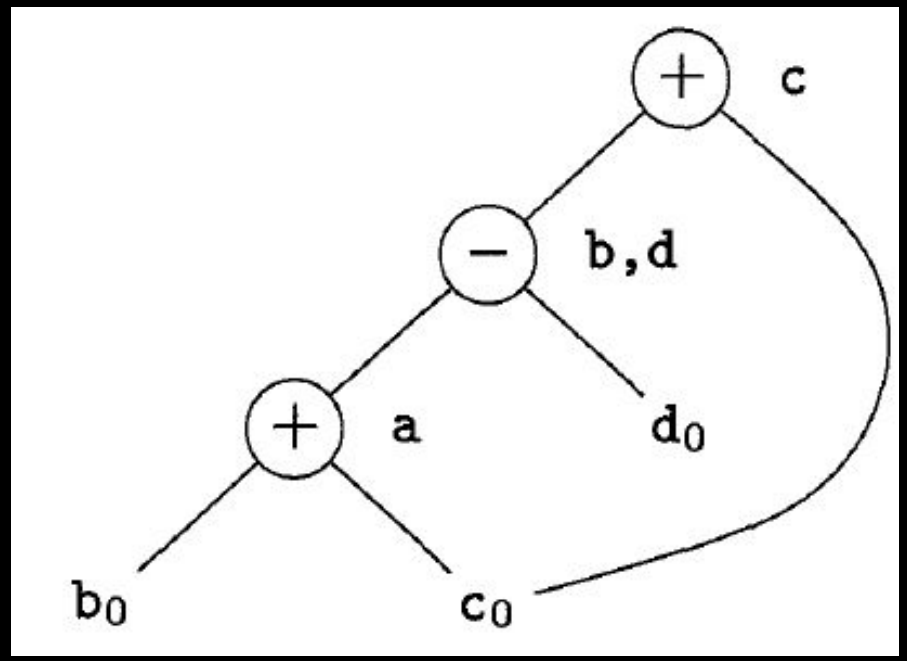
- The DAG representation of a basic block lets us perform several code improving transformations on the code represented by the block.
  - a) We can eliminate local common subexpressions, that is, instructions that compute a value that has already been computed.
  - b) We can eliminate dead code, that is, instructions that compute a value that is never used.
  - c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
  - d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

The DAG Representation of Basic Blocks

A DAG for the block

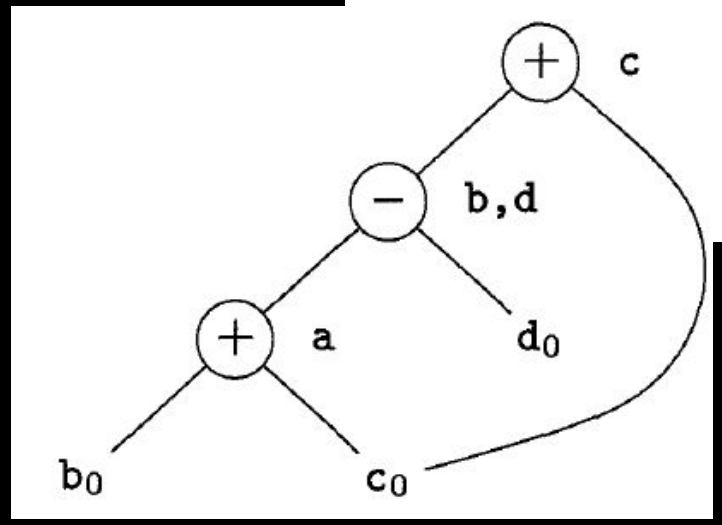
```
a = b + c
b = a - d
c = b + c
d = a - d
```

DAG for Basic Block



Finding Local Common Subexpressions

A DAG for the block




```
a = b + c
b = a - d
c = b + c
d = a - d
```

```
a = b + c
d = a - d
b = d
c = d + c
```

If b is not live on exit from the block

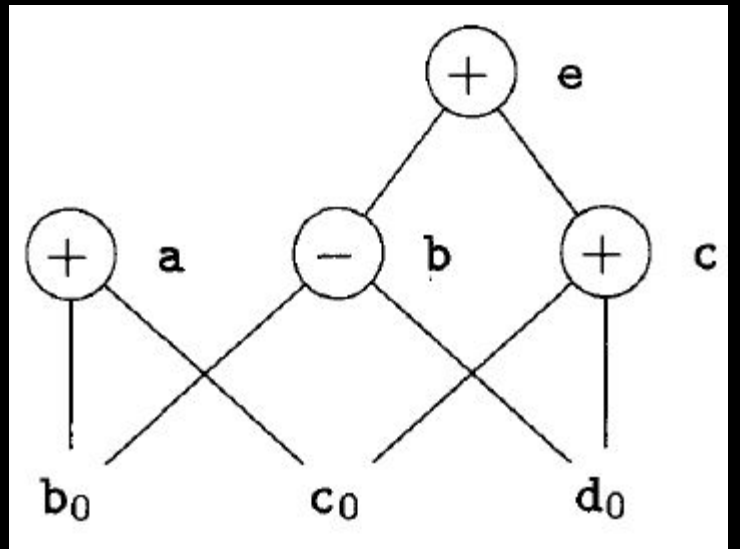
```
a = b + c
d = a - d
c = d + c
```

If b and d both is live on exit from the block

Dead Code Elimination

The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

If a and b are live but c and e are not, we can immediately remove the root labeled e. Then, the node labeled c becomes a root and can be removed. The roots labeled a and b remain, since they each have live variables attached.



Use of Algebraic Identities

1) Algebraic identities represent important class of optimizations on basic blocks.

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

2) Another class of algebraic optimizations includes local reduction in strength, that is, replacing a more expensive operator by a cheaper one as in:

EXPENSIVE		CHEAPER
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

3) Constant folding - A third class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values.

Thus the expression 2 * 3.14 would be replaced by 6.28.

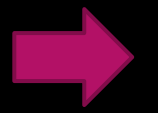
Reorder of statements

The following intermediate code might be generated:

```
a = b + c;
e = c + d + b;
```

```
a = b + c
t = c + d
e = t + b
```

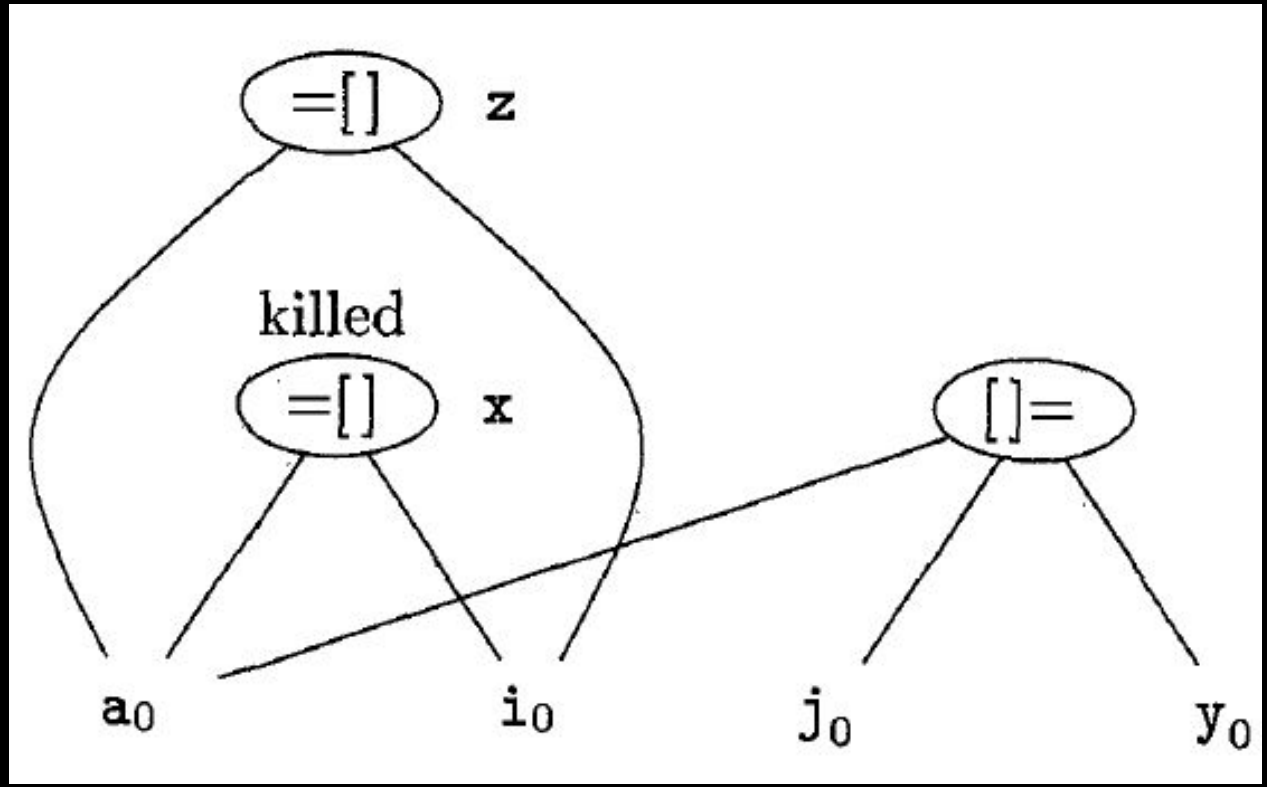
If t is not needed outside this block, we can change this sequence to



```
a = b + c
e = a + d
```

Representation of Array References

```
x = a[i]
a[j] = y
z = a[i]
```



Representation of Array References

```
b = 12 + a
x = b[i]
b[j] = y
```