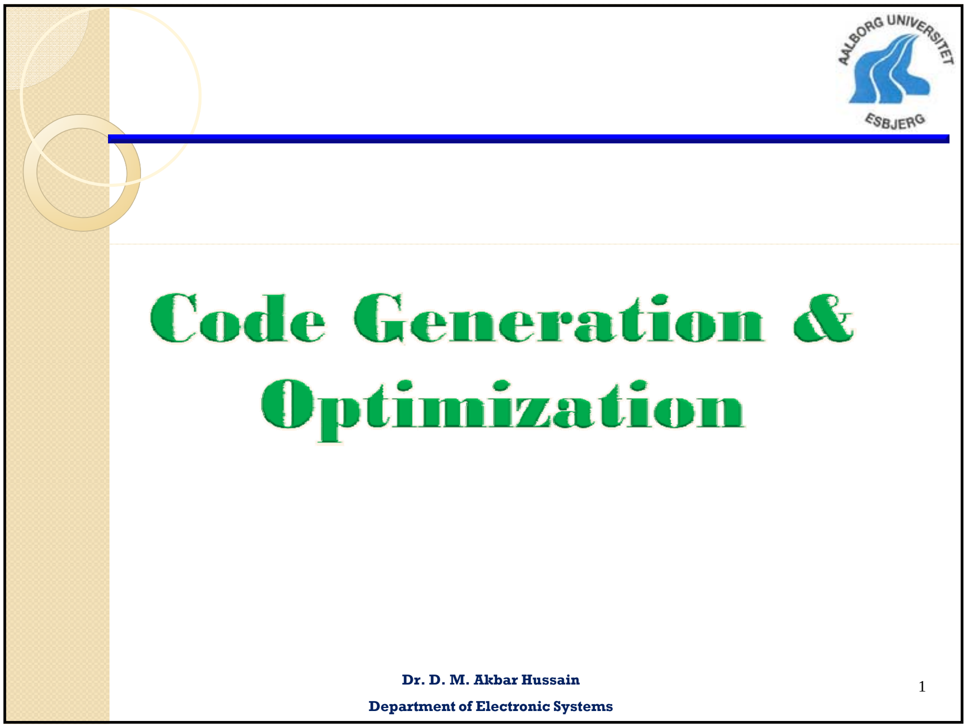
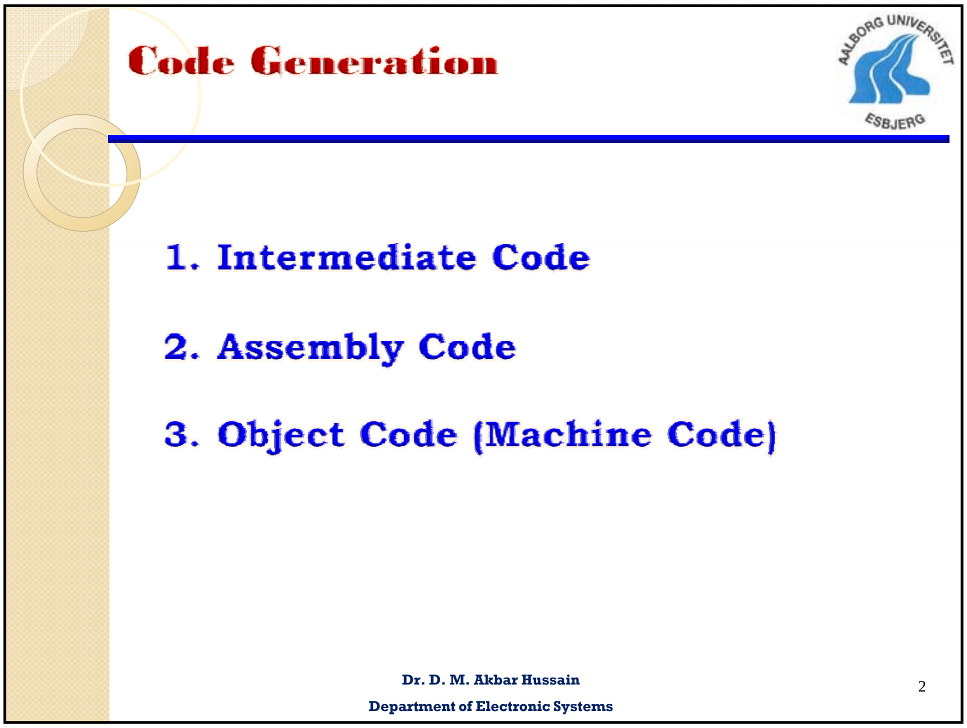


Code Generation & Optimization



Code Generation

1. Intermediate Code
2. Assembly Code
3. Object Code (Machine Code)

Intermediate Code

1. P-Code
2. Three Address Code

Intermediate Representation

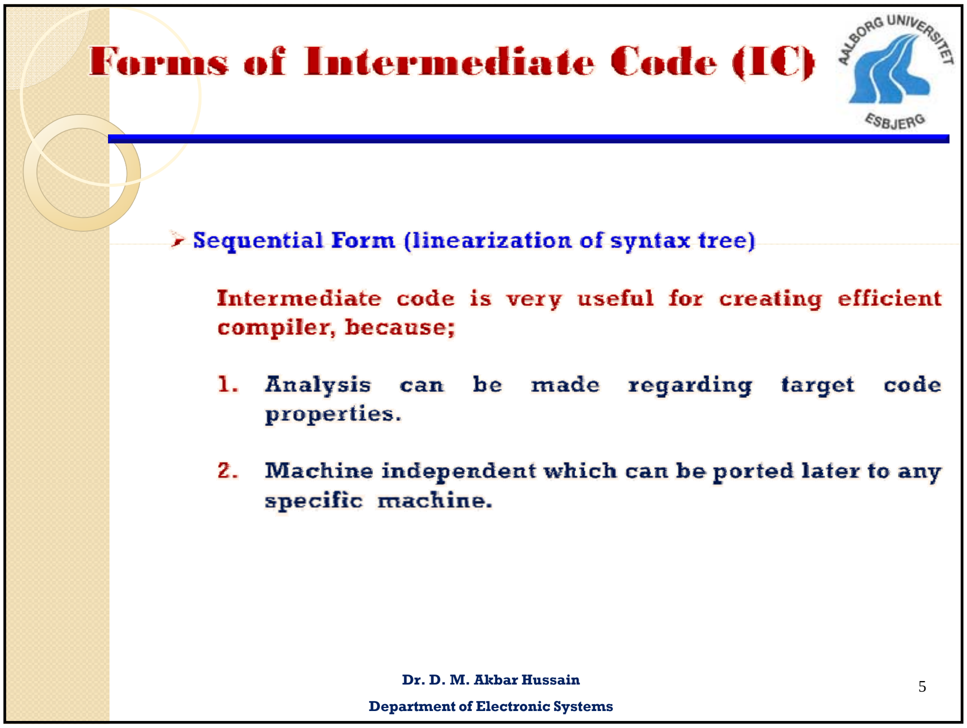
- Abstract Syntax Tree

Most Suitable data structure perhaps for the intermediate representation but miles away from the actual target code representation.

Especially, for

Control Flow Constructs
Jumps

Therefore, a transformation/Translation is required to make intermediate representation more look like a target code, so intermediate code is that translation construct.

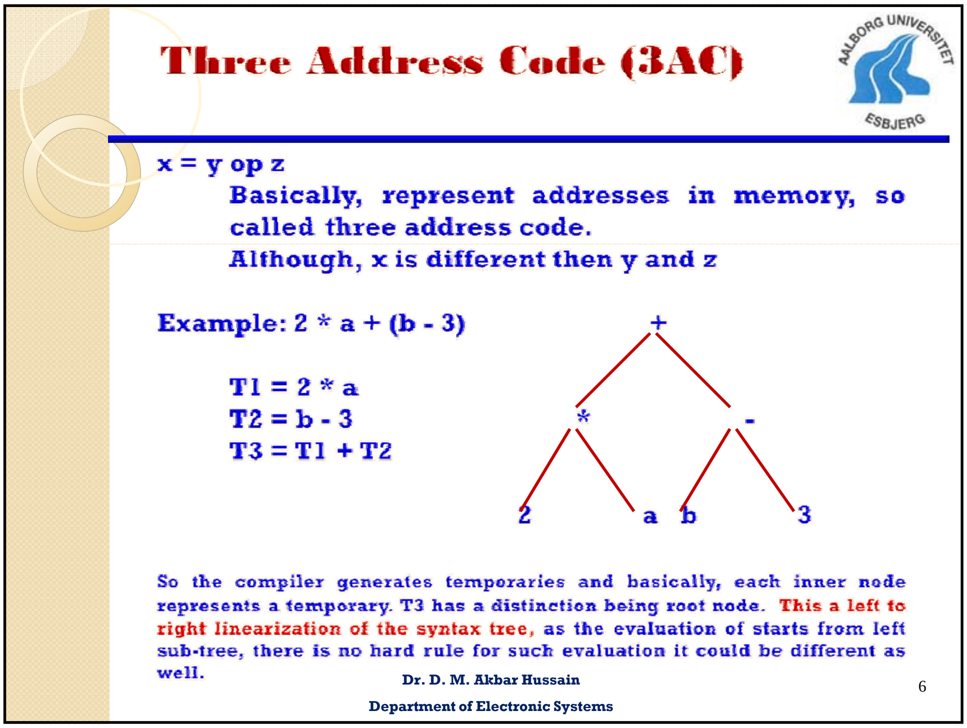


Forms of Intermediate Code (IC)

- Sequential Form (linearization of syntax tree)

Intermediate code is very useful for creating efficient compiler, because;

1. Analysis can be made regarding target code properties.
2. Machine independent which can be ported later to any specific machine.



Three Address Code (3AC)

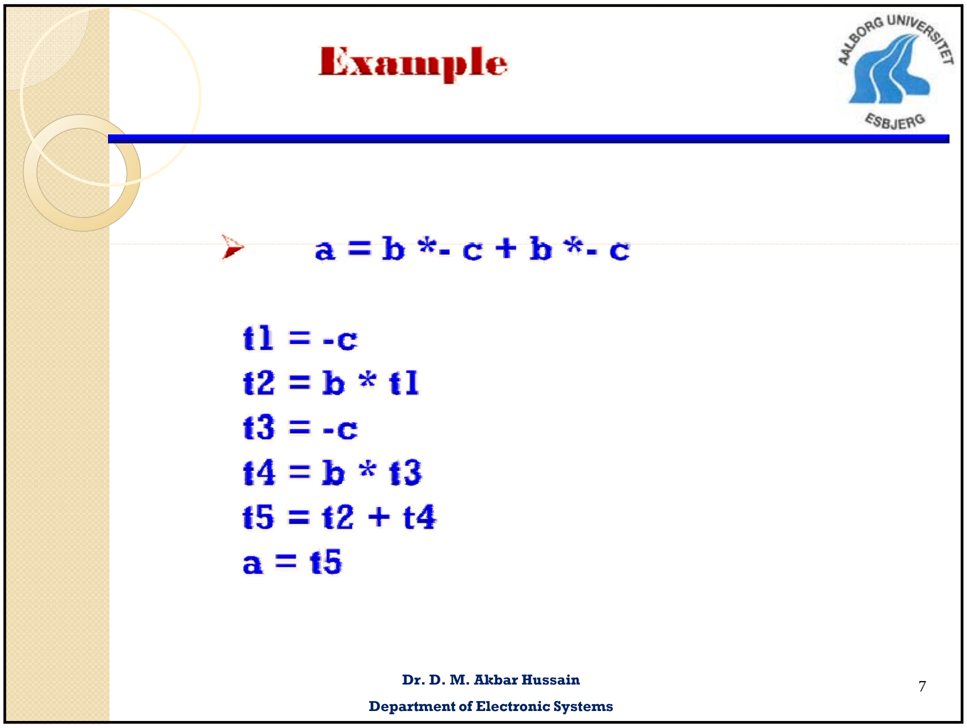
x = y op z

Basically, represent addresses in memory, so called three address code.
Although, x is different then y and z

Example: 2 * a + (b - 3)

T1 = 2 * a
T2 = b - 3
T3 = T1 + T2

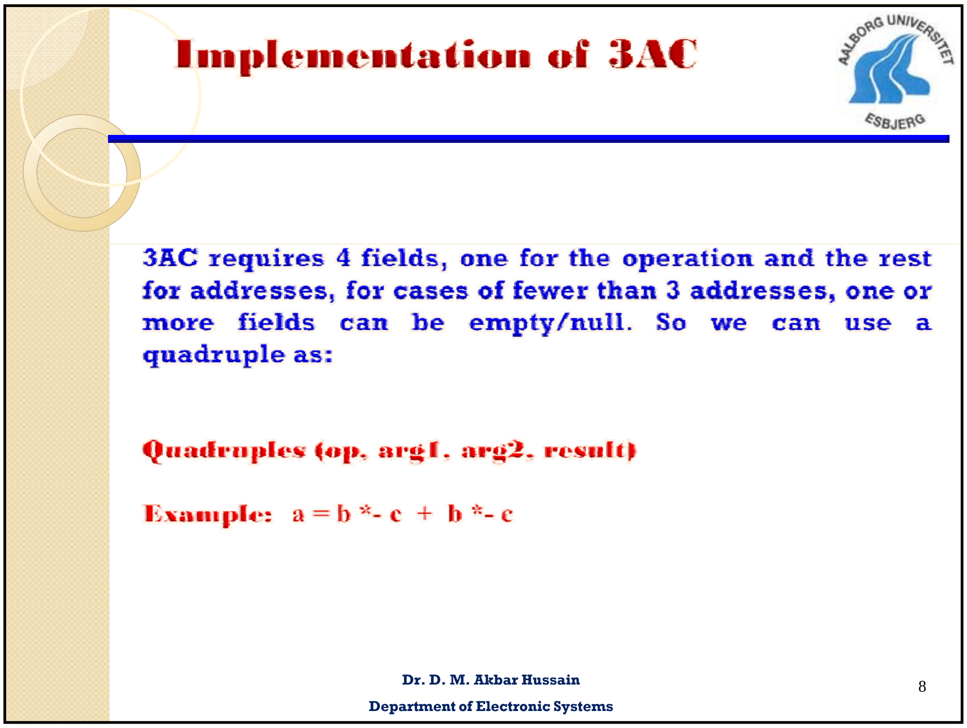
So the compiler generates temporaries and basically, each inner node represents a temporary. T3 has a distinction being root node. This a left to right linearization of the syntax tree, as the evaluation of starts from left sub-tree, there is no hard rule for such evaluation it could be different as well.



Example

- a = b *- c + b *- c

t1 = -c
t2 = b * t1
t3 = -c
t4 = b * t3
t5 = t2 + t4
a = t5

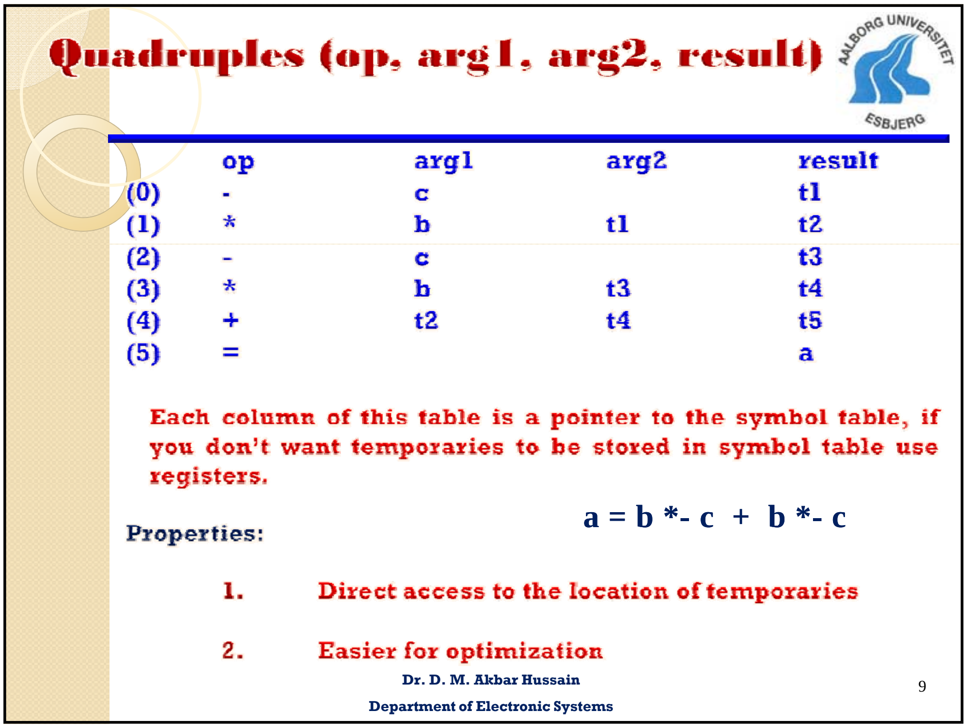


Implementation of 3AC

3AC requires 4 fields, one for the operation and the rest for addresses, for cases of fewer than 3 addresses, one or more fields can be empty/null. So we can use a quadruple as:

Quadruples (op, arg1, arg2, result)

Example: a = b *- c + b *- c



Quadruples (op, arg1, arg2, result)

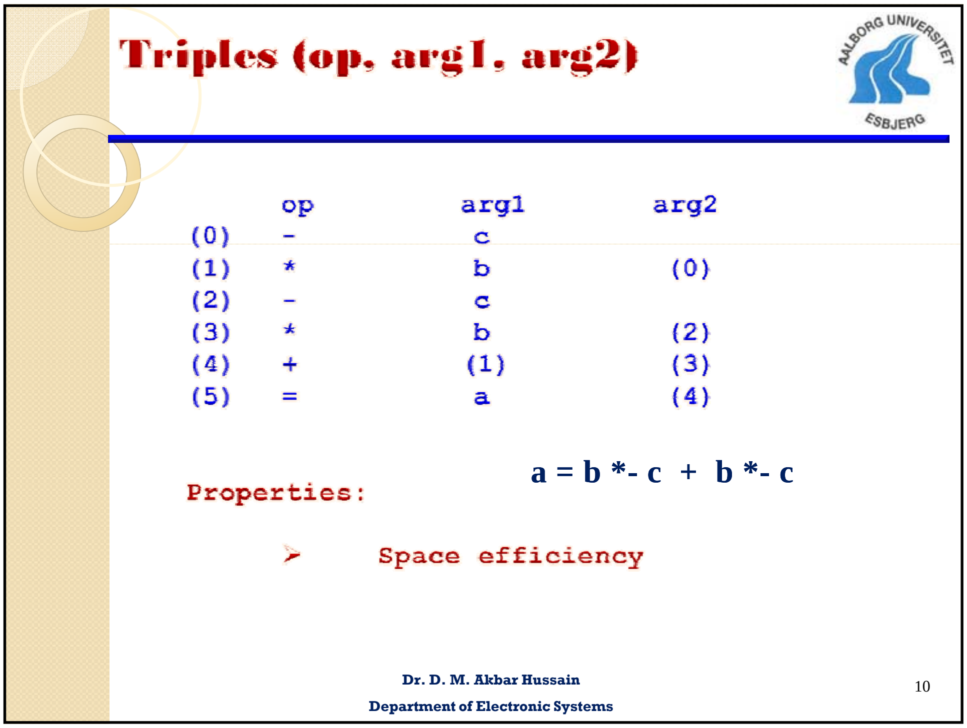
	op	arg1	arg2	result
(0)	-	c		t1
(1)	*	b	t1	t2
(2)	-	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=			a

Each column of this table is a pointer to the symbol table, if you don't want temporaries to be stored in symbol table use registers.

a = b *- c + b *- c

Properties:

1. Direct access to the location of temporaries
2. Easier for optimization



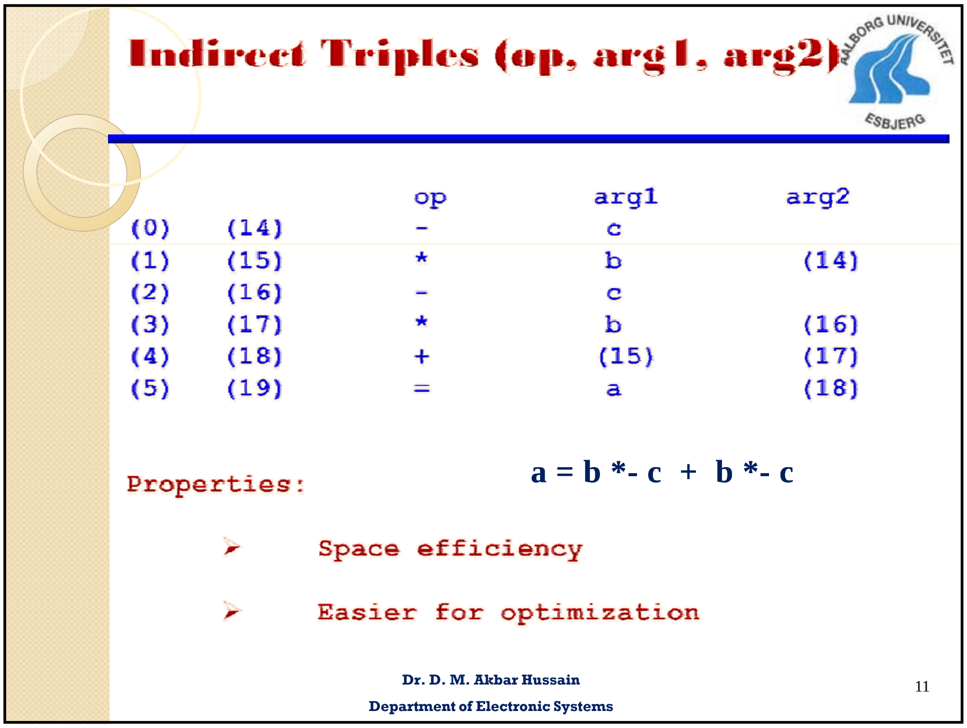
Triples (op, arg1, arg2)

	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

a = b *- c + b *- c

Properties:

- Space efficiency



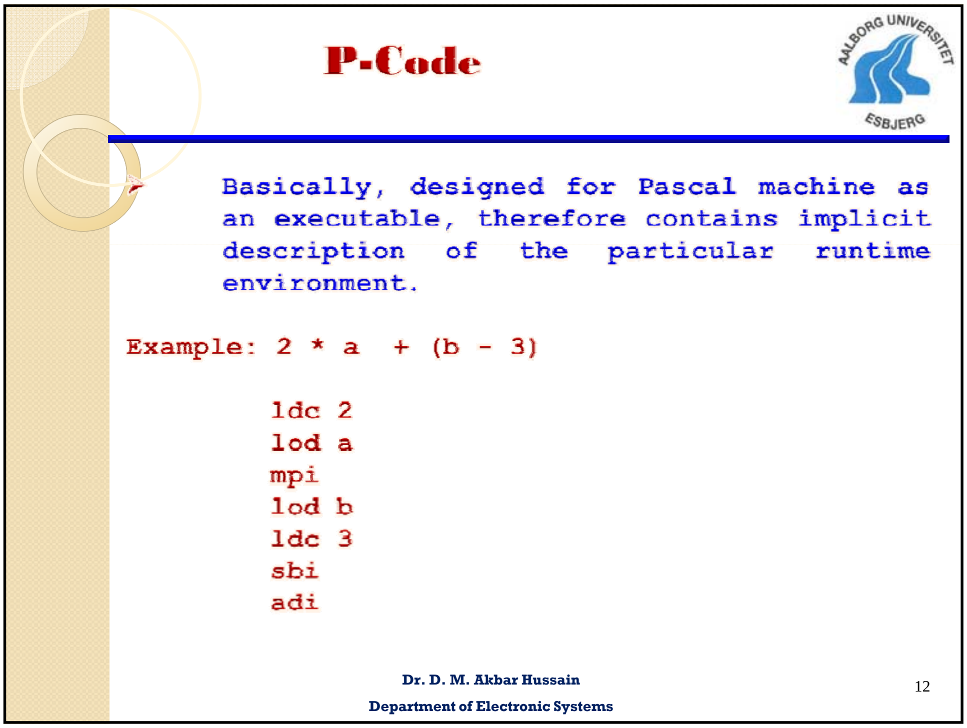
Indirect Triples (op, arg1, arg2)

		op	arg1	arg2
(0)	(14)	-	c	
(1)	(15)	*	b	(14)
(2)	(16)	-	c	
(3)	(17)	*	b	(16)
(4)	(18)	+	(15)	(17)
(5)	(19)	=	a	(18)

a = b *- c + b *- c

Properties:

- Space efficiency
- Easier for optimization

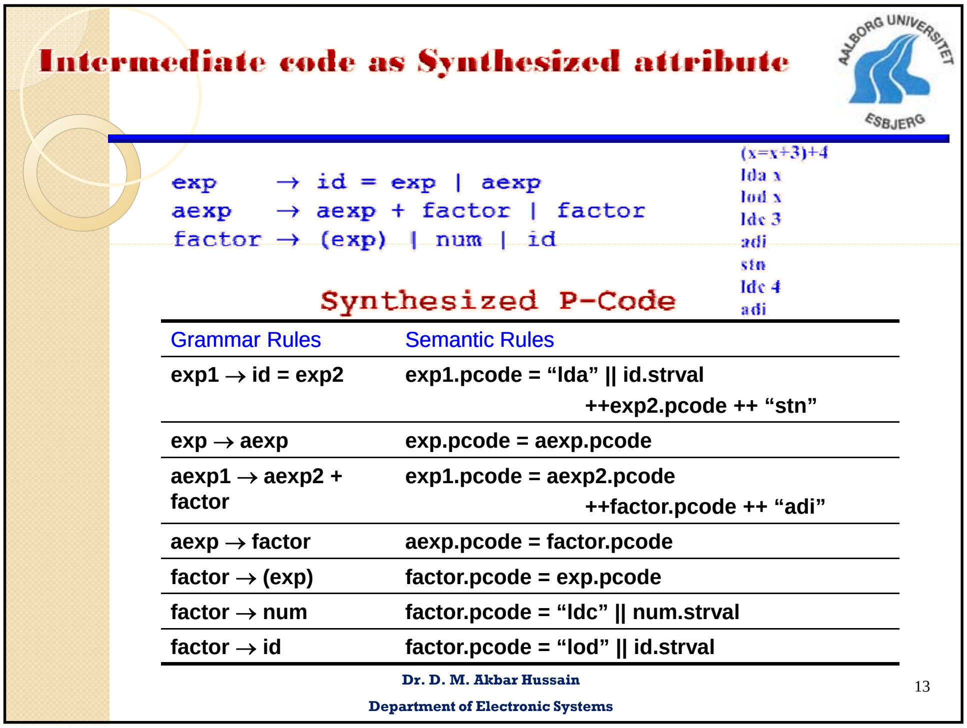


P-Code

- Basically, designed for Pascal machine as an executable, therefore contains implicit description of the particular runtime environment.

Example: 2 * a + (b - 3)

```
ldc 2
lod a
mpi
lod b
ldc 3
sbi
adi
```



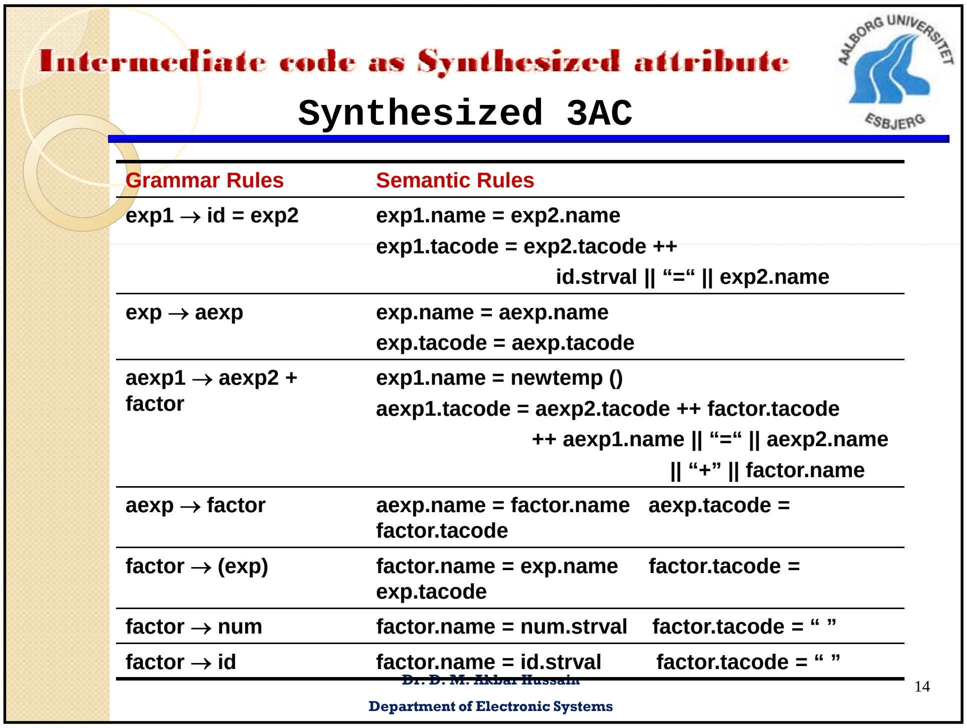
Intermediate code as Synthesized attribute

```
exp    → id = exp | aexp
aexp   → aexp + factor | factor
factor → (exp) | num | id
```

(x=x+3)+4
lda x
lod x
ldc 3
adi
stn
ldc 4
adi

Synthesized P-Code

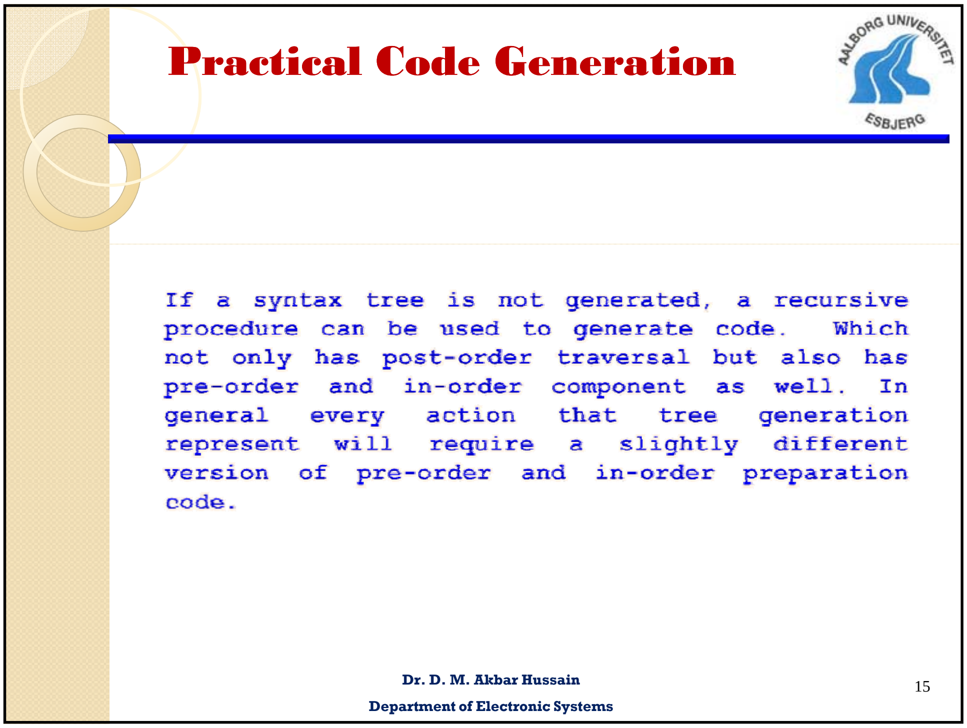
Grammar Rules	Semantic Rules
exp1 → id = exp2	**exp1.pcode = “lda” || id.strval ++exp2.pcode ++ “stn”**
exp → aexp	**exp.pcode = aexp.pcode**
aexp1 → aexp2 + factor	**exp1.pcode = aexp2.pcode ++factor.pcode ++ “adi”**
aexp → factor	**aexp.pcode = factor.pcode**
factor → (exp)	**factor.pcode = exp.pcode**
factor → num	**factor.pcode = “ldc” || num.strval**
factor → id	**factor.pcode = “lod” || id.strval**



Intermediate code as Synthesized attribute

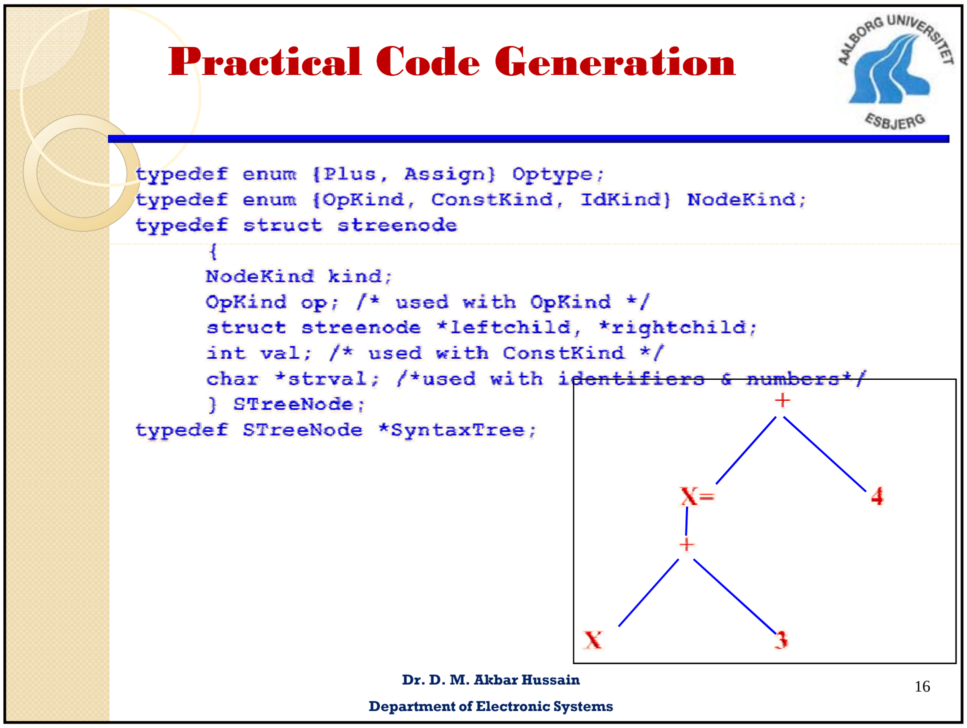
Synthesized 3AC

Grammar Rules	Semantic Rules
exp1 → id = exp2	**exp1.name = exp2.name exp1.tacode = exp2.tacode ++ id.strval || “=“ || exp2.name**
exp → aexp	**exp.name = aexp.name exp.tacode = aexp.tacode**
aexp1 → aexp2 + factor	**exp1.name = newtemp () aexp1.tacode = aexp2.tacode ++ factor.tacode ++ aexp1.name || “=“ || aexp2.name || “+” || factor.name**
aexp → factor	**aexp.name = factor.name aexp.tacode = factor.tacode**
factor → (exp)	**factor.name = exp.name factor.tacode = exp.tacode**
factor → num	**factor.name = num.strval factor.tacode = “ ”**
factor → id	**factor.name = id.strval factor.tacode = “ ”**



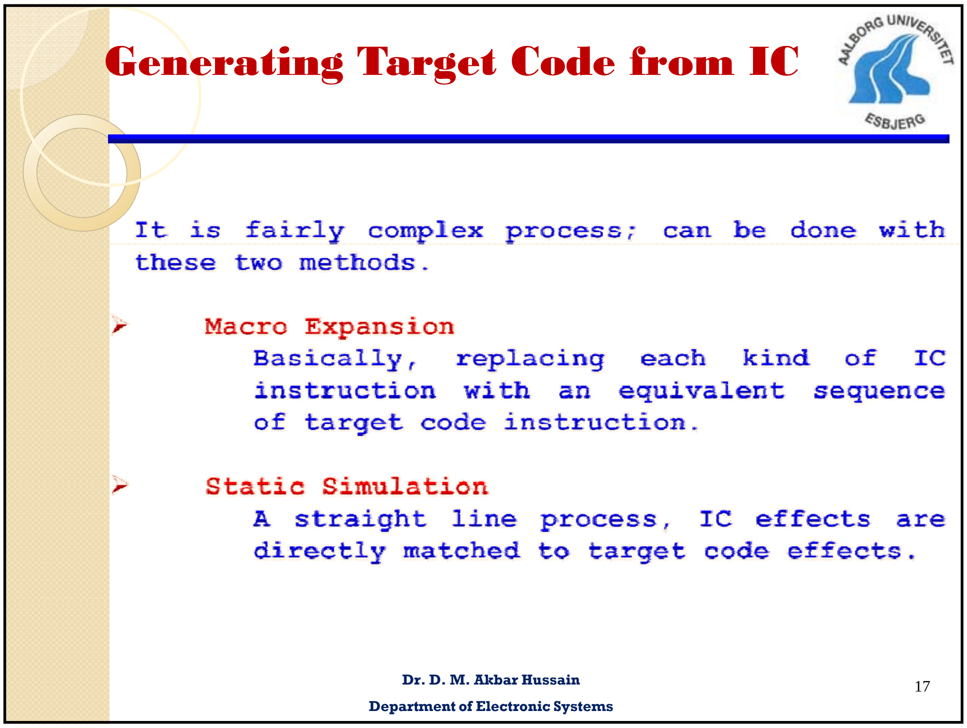
Practical Code Generation

If a syntax tree is not generated, a recursive procedure can be used to generate code. Which not only has post-order traversal but also has pre-order and in-order component as well. In general every action that tree generation represent will require a slightly different version of pre-order and in-order preparation code.



Practical Code Generation

```
typedef enum {Plus, Assign} Optype;
typedef enum {OpKind, ConstKind, IdKind} NodeKind;
typedef struct streenode
      {
      NodeKind kind;
      OpKind op; /* used with OpKind */
      struct streenode *leftchild, *rightchild;
      int val; /* used with ConstKind */
      char *strval; /*used with identifiers & numbers*/
      } STreeNode;
typedef STreeNode *SyntaxTree;
```

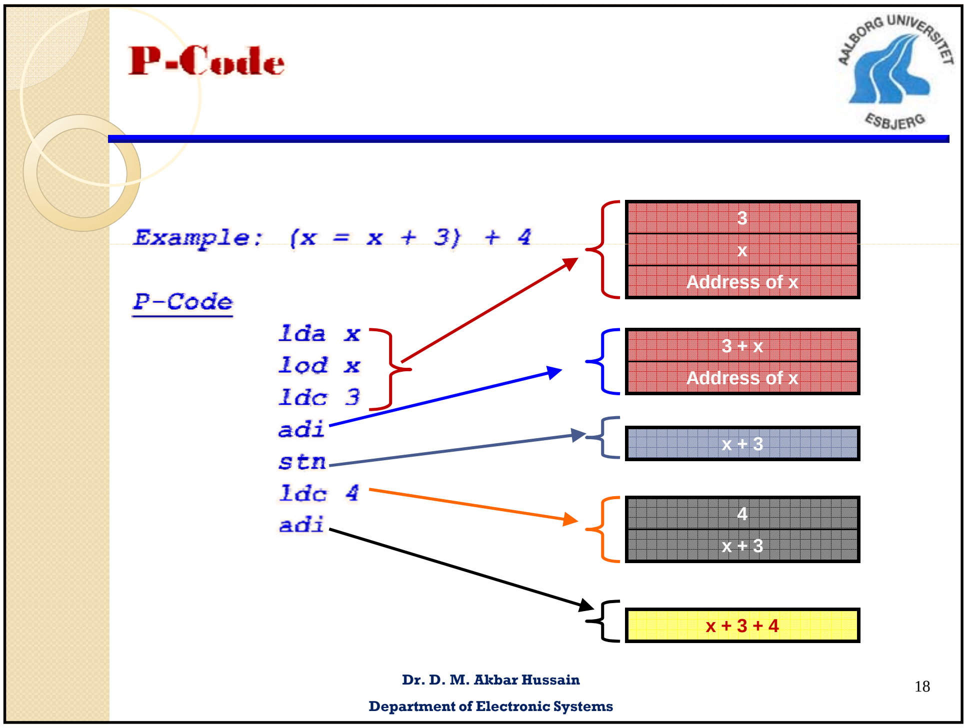


Generating Target Code from IC

It is fairly complex process; can be done with these two methods.

- Macro Expansion
  Basically, replacing each kind of IC instruction with an equivalent sequence of target code instruction.
- Static Simulation
  A straight line process, IC effects are directly matched to target code effects.

Dr. D. M. Akbar Hussain
Department of Electronic Systems



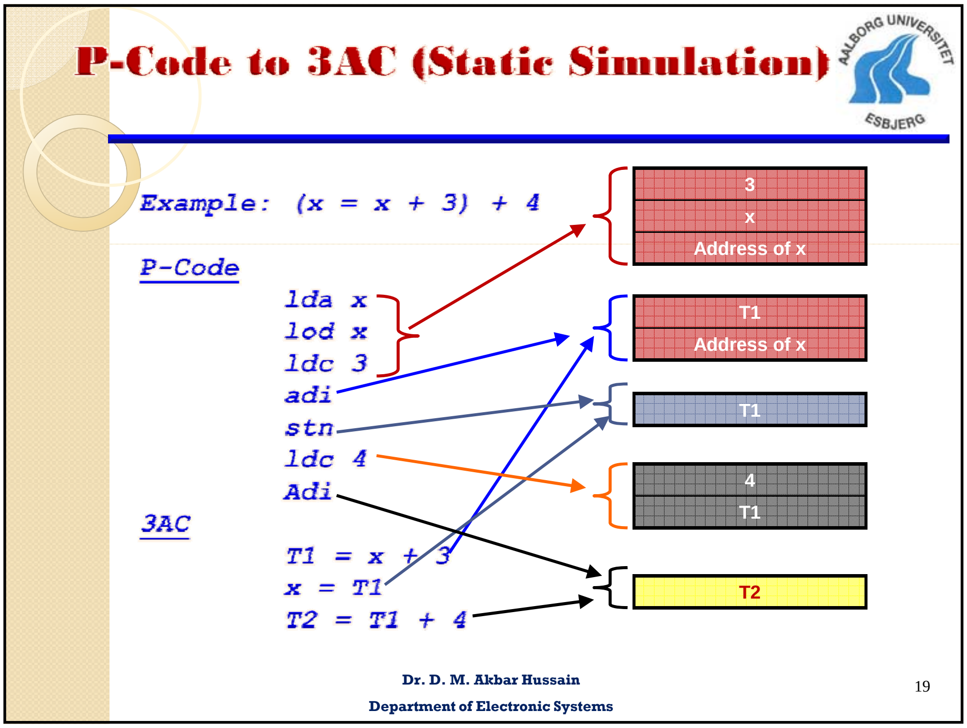
P-Code

Example: (x = x + 3) + 4

P-Code

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

Dr. D. M. Akbar Hussain
Department of Electronic Systems



P-Code to 3AC (Static Simulation)

Example: (x = x + 3) + 4

P-Code

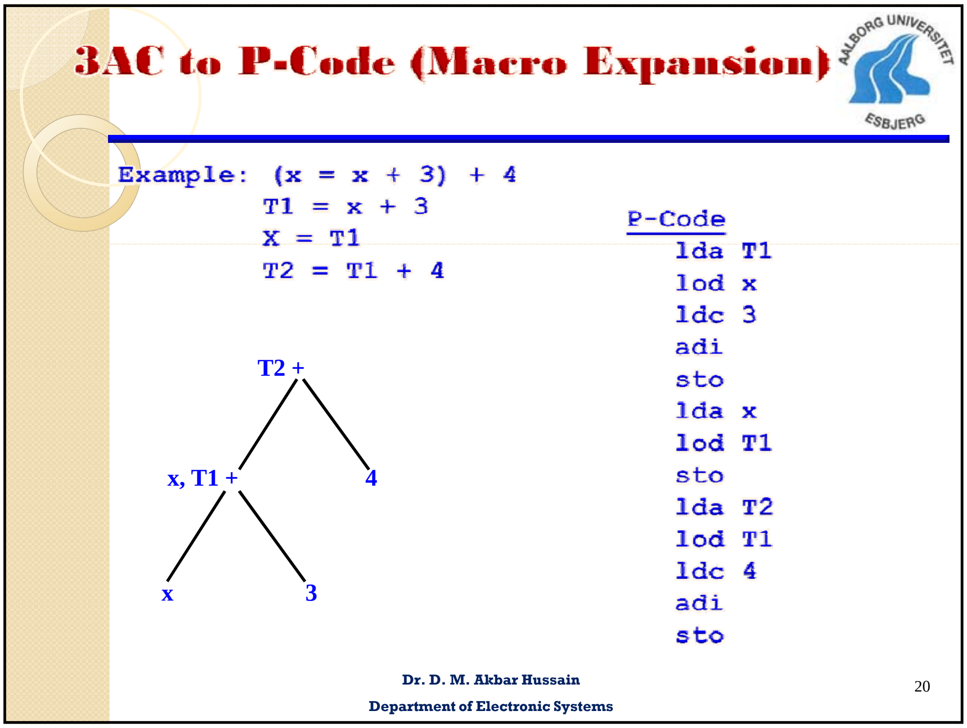
```
lda x
lod x
ldc 3
adi
stn
ldc 4
Adi
```

3AC

```
T1 = x + 3
x = T1
T2 = T1 + 4
```

Dr. D. M. Akbar Hussain

Department of Electronic Systems



3AC to P-Code (Macro Expansion)

Example: (x = x + 3) + 4

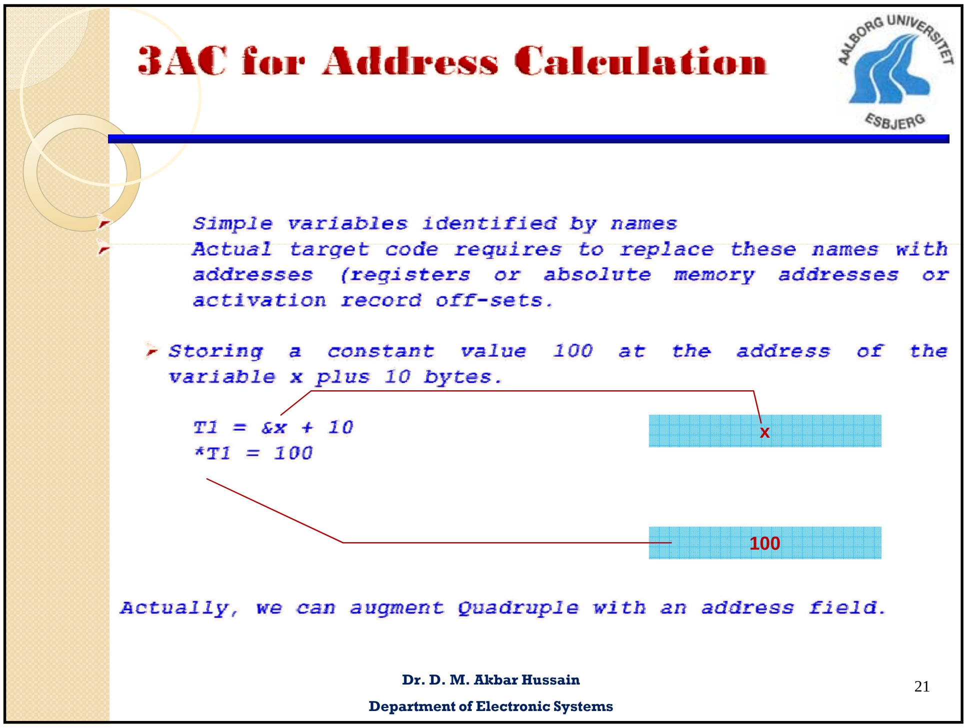
```
T1 = x + 3
X = T1
T2 = T1 + 4
```

P-Code

```
lda T1
lod x
ldc 3
adi
sto
lda x
lod T1
sto
lda T2
lod T1
ldc 4
adi
sto
```

Dr. D. M. Akbar Hussain

Department of Electronic Systems



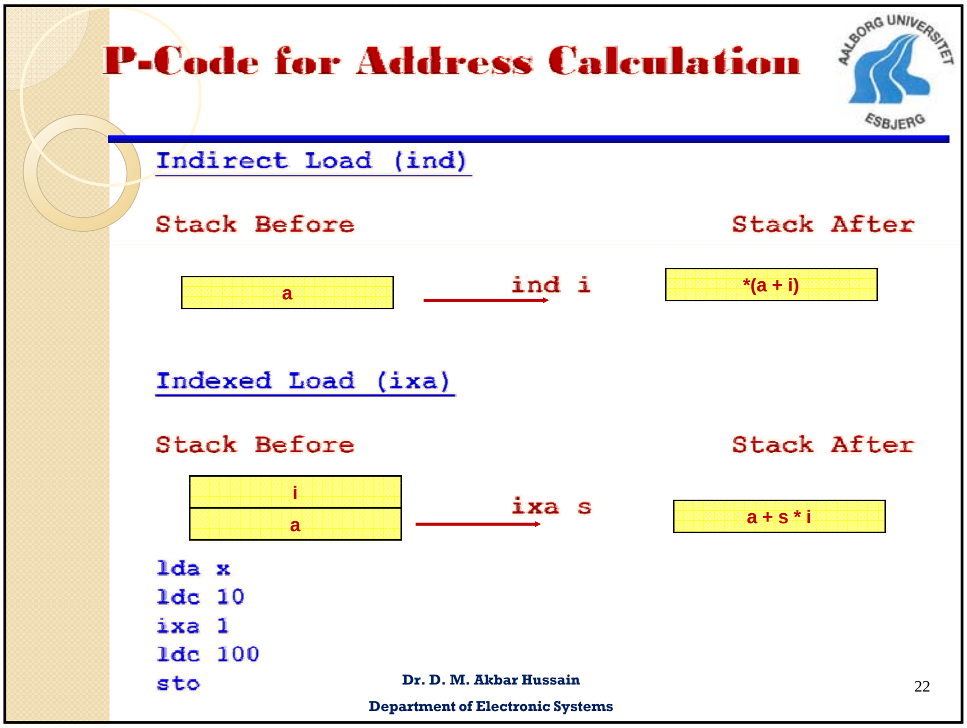
3AC for Address Calculation

- Simple variables identified by names
- Actual target code requires to replace these names with addresses (registers or absolute memory addresses or activation record off-sets.

- Storing a constant value 100 at the address of the variable x plus 10 bytes.

```
T1 = &x + 10
*T1 = 100
```

Actually, we can augment Quadruple with an address field.



P-Code for Address Calculation

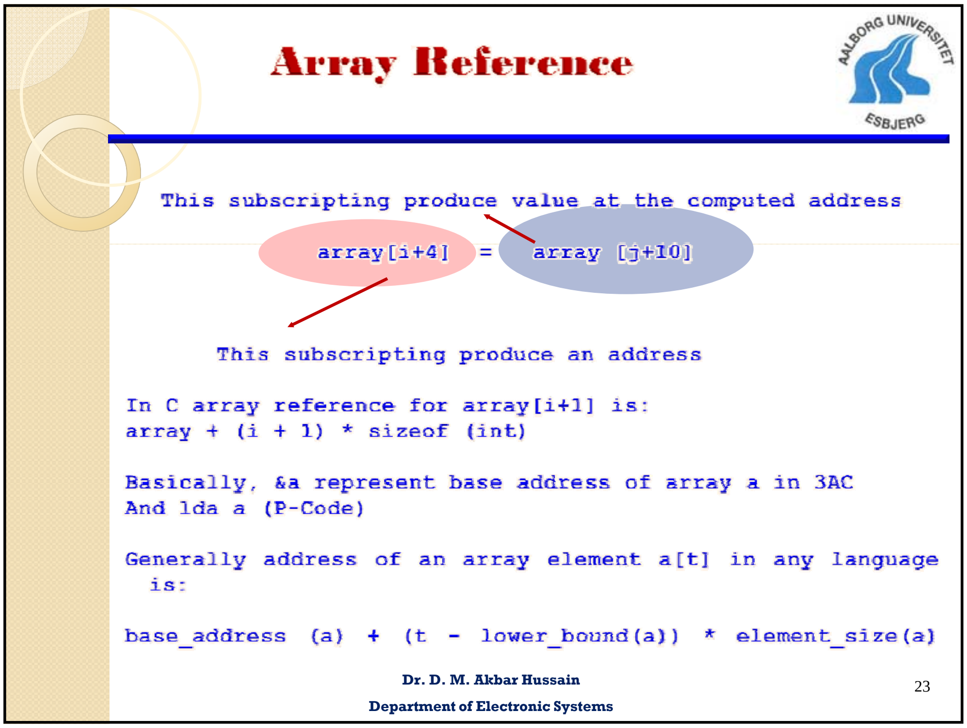
Indirect Load (ind)

Stack Before Stack After

Indexed Load (ixa)

Stack Before Stack After

```
lda x
ldc 10
ixa 1
ldc 100
sto
```



Array Reference

This subscripting produce value at the computed address

array[i+4] = array [j+10]

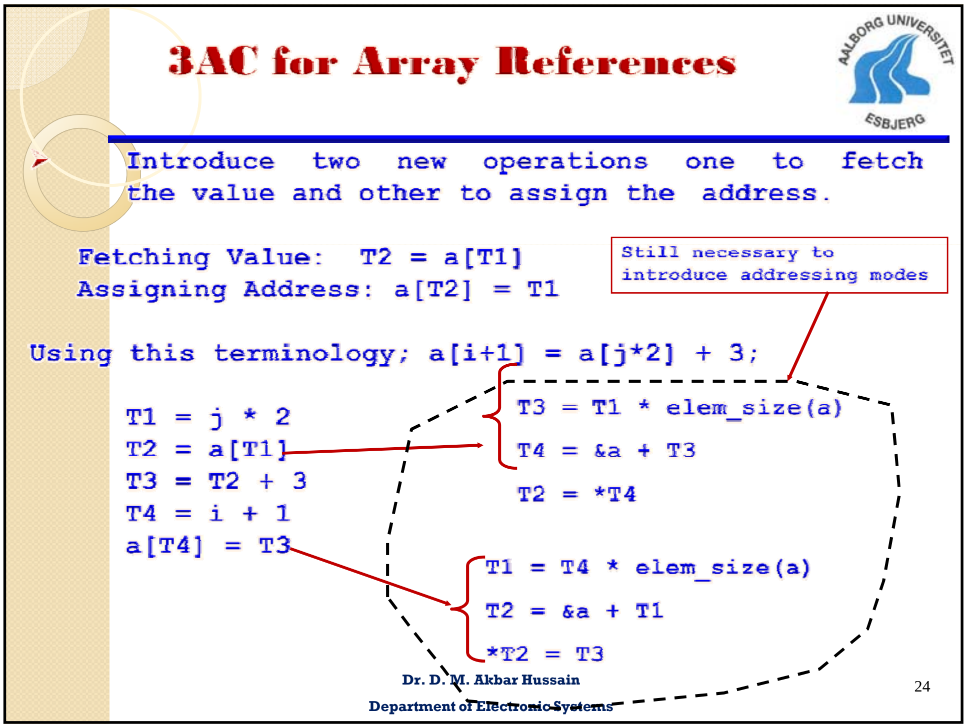
This subscripting produce an address

In C array reference for array[i+1] is:
array + (i + 1) * sizeof (int)

Basically, &a represent base address of array a in 3AC
And lda a (P-Code)

Generally address of an array element a[t] in any language is:

base_address (a) + (t - lower_bound(a)) * element_size(a)



3AC for Array References

Introduce two new operations one to fetch the value and other to assign the address.

Fetching Value: T2 = a[T1]
Assigning Address: a[T2] = T1

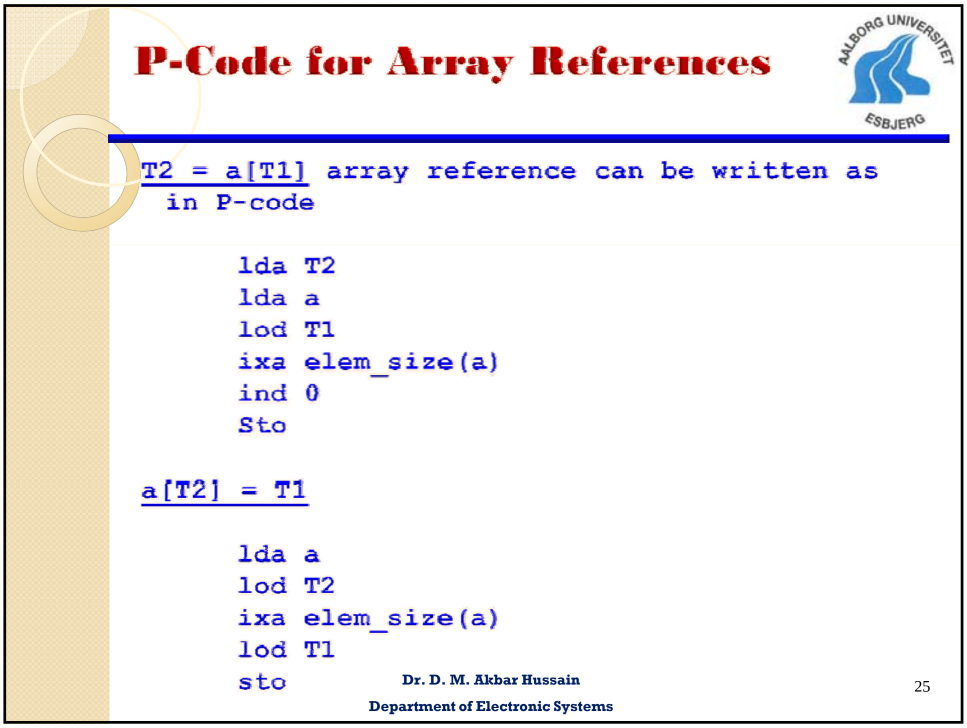
Still necessary to introduce addressing modes

Using this terminology; a[i+1] = a[j*2] + 3;

```
T1 = j * 2
T2 = a[T1]
T3 = T2 + 3
T4 = i + 1
a[T4] = T3
```

```
T3 = T1 * elem_size(a)
T4 = &a + T3
T2 = *T4
```

```
T1 = T4 * elem_size(a)
T2 = &a + T1
*T2 = T3
```



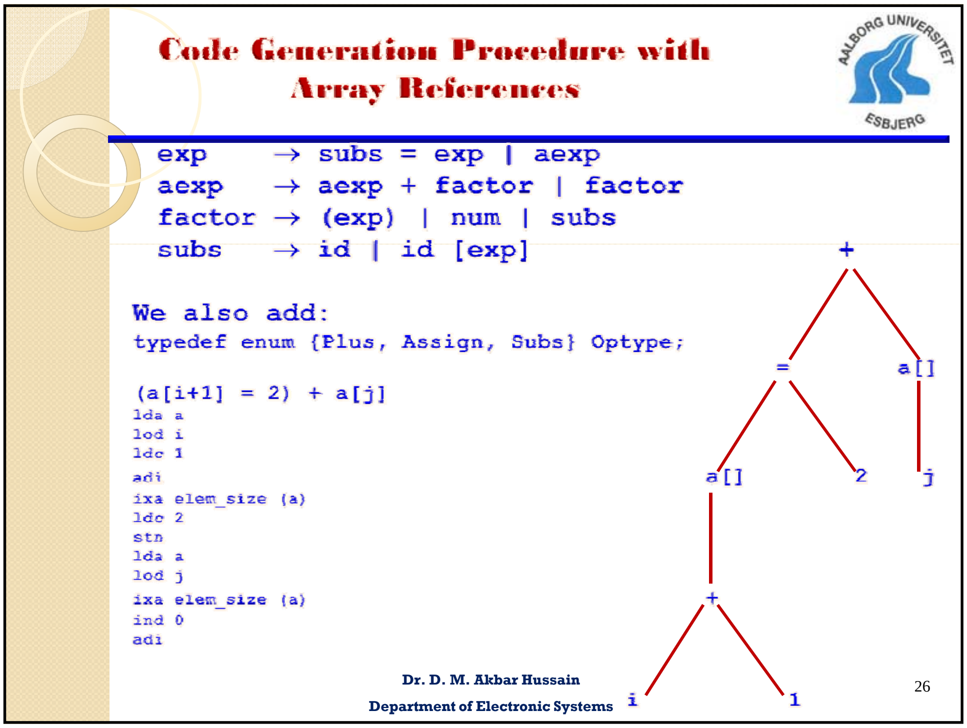
P-Code for Array References

T2 = a[T1] array reference can be written as in P-code

```
lda T2
lda a
lod T1
ixa elem_size(a)
ind 0
Sto
```

a[T2] = T1

```
lda a
lod T2
ixa elem_size(a)
lod T1
sto
```



Code Generation Procedure with Array References

```
exp     → subs = exp | aexp
aexp    → aexp + factor | factor
factor  → (exp) | num | subs
subs    → id | id [exp]
```

We also add:

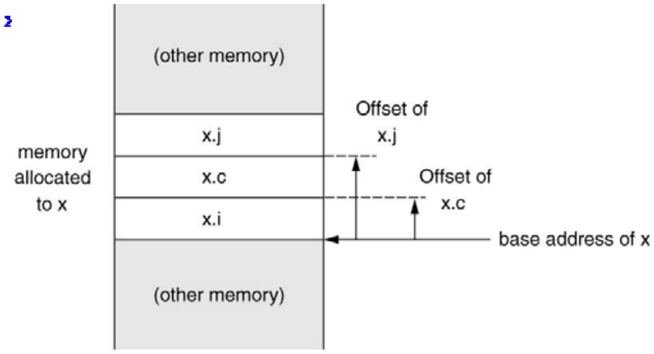
```
typedef enum {Plus, Assign, Subs} Optype;
```

(a[i+1] = 2) + a[j]

```
lda a
lod i
ldc 1
adi
ixa elem_size (a)
ldc 2
stn
lda a
lod j
ixa elem_size (a)
ind 0
adi
```

Record Structure and Pointer References

```
typedef struct rec
              {     int i;
                    char c;
                    int j;} Rec;
...
Rec x
```



Code Generation

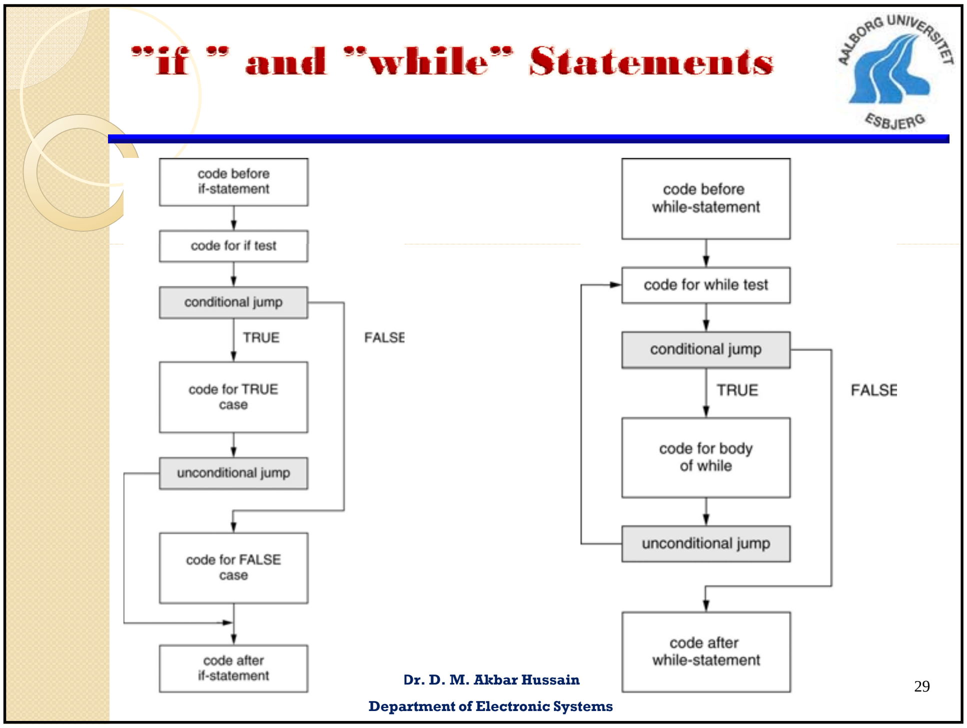
- Control Statements and logical expressions require labels for both 3AC and P-Code intermediate code generation. It is similar to have temporary name generation but stands for address to which jumps are made.

If and While

if-stmt → if (exp) stmt / if (exp) stmt else stmt
while stmt→ while (exp) stmt.

Job is to construct these structured control statements into unstructured statements coupled with jumps.

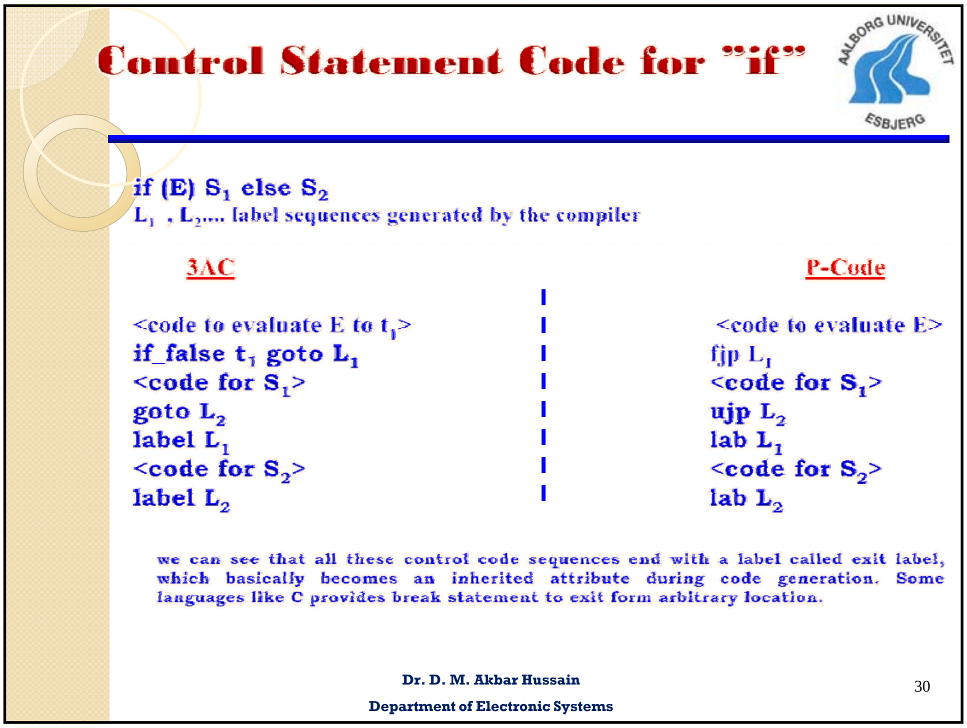
- Basically there are two kinds of jumps:
    - Conditional
    - Un-Conditional (e.g: goto)
- For true case, there is no jump (Fall through) reducing the number of jumps to two.



"if " and "while" Statements

code before if-statement
code for if test
conditional jump
TRUE
FALSE
code for TRUE case
unconditional jump
code for FALSE case
code after if-statement

code before while-statement
code for while test
conditional jump
TRUE
FALSE
code for body of while
unconditional jump
code after while-statement

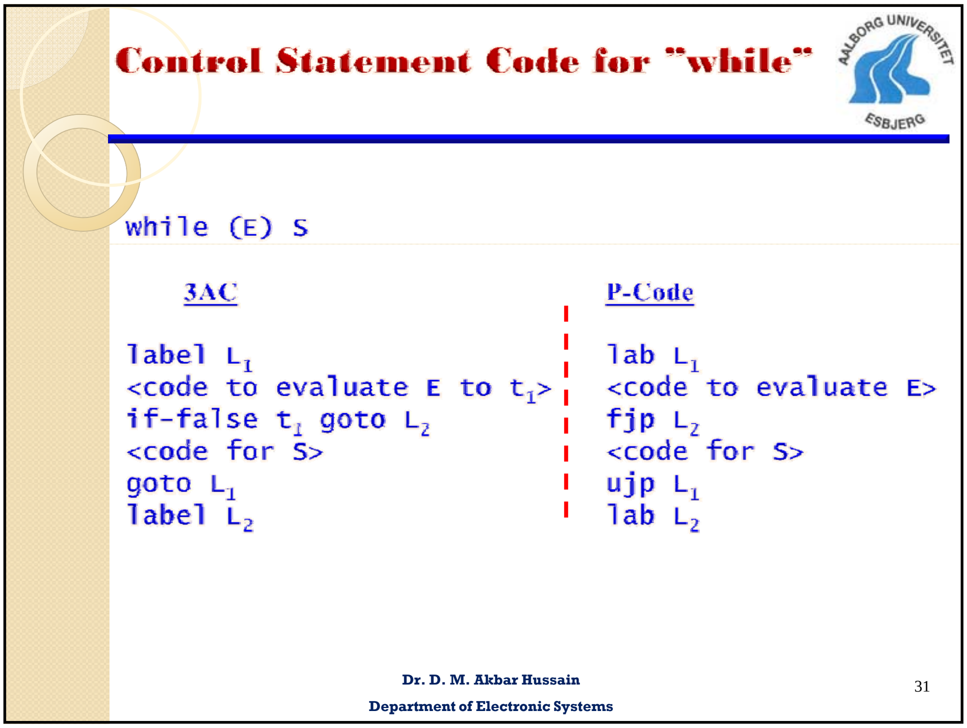


Control Statement Code for "if"

if (E) S_1 else S_2
L_1 , L_2.... label sequences generated by the compiler

3AC	P-Code
<code to evaluate E to t_1>	<code to evaluate E>
if_false t_1 goto L_1	fjp L_1
<code for S_1>	<code for S_1>
goto L_2	ujp L_2
label L_1	lab L_1
<code for S_2>	<code for S_2>
label L_2	lab L_2

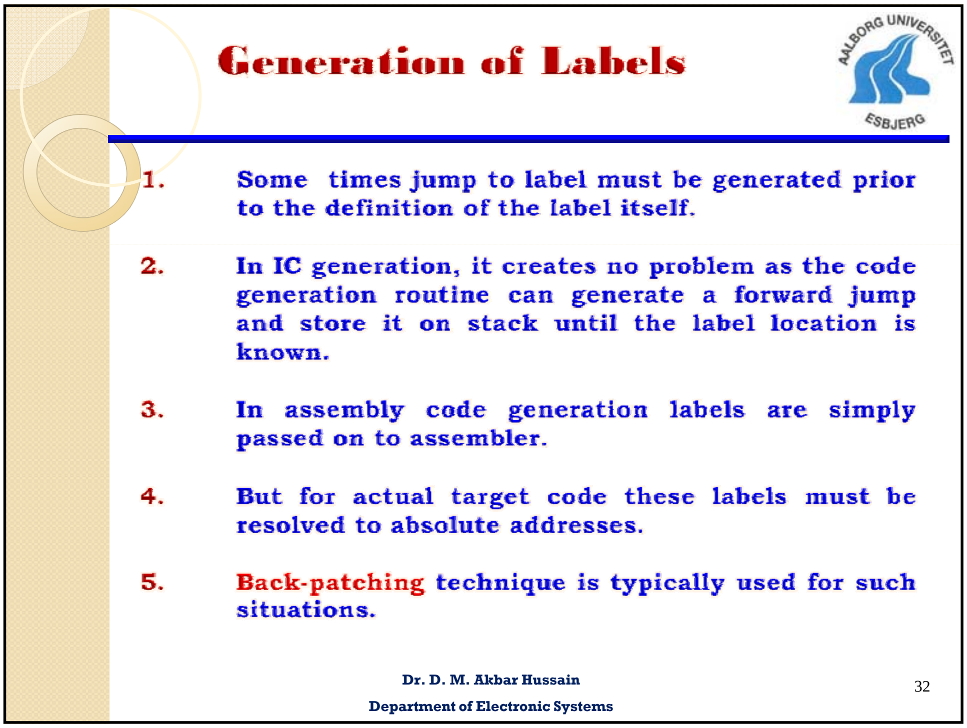
we can see that all these control code sequences end with a label called exit label, which basically becomes an inherited attribute during code generation. Some languages like C provides break statement to exit form arbitrary location.



Control Statement Code for "while"

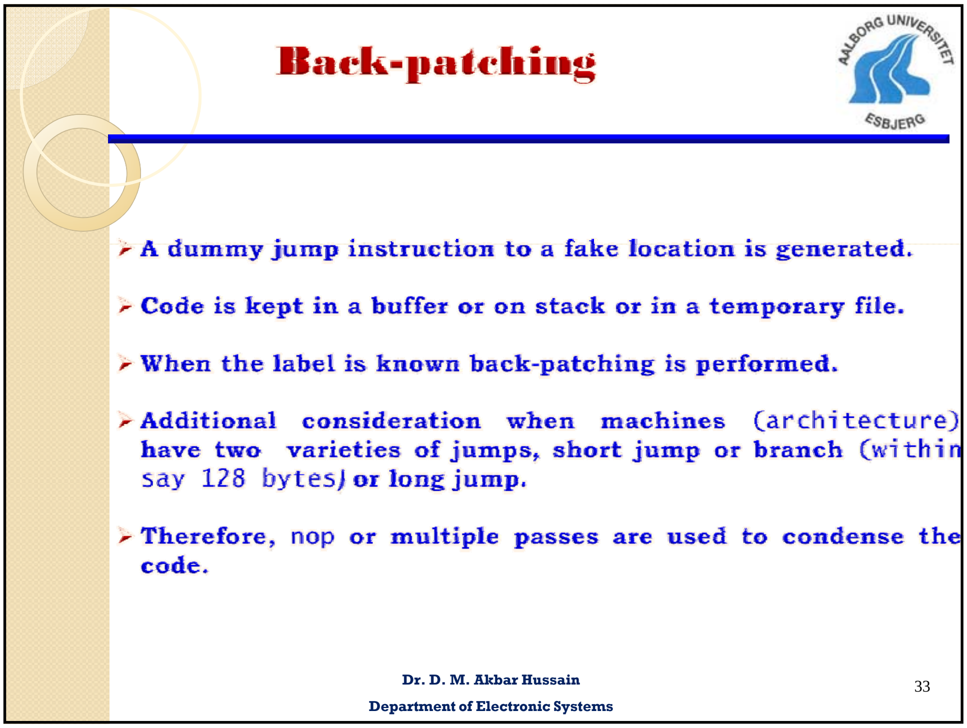
`while (E) S`

3AC	P-Code
label L_1	lab L_1
<code to evaluate E to t_1>	<code to evaluate E>
if-false t_1 goto L_2	fjp L_2
<code for S>	<code for S>
goto L_1	ujp L_1
label L_2	lab L_2



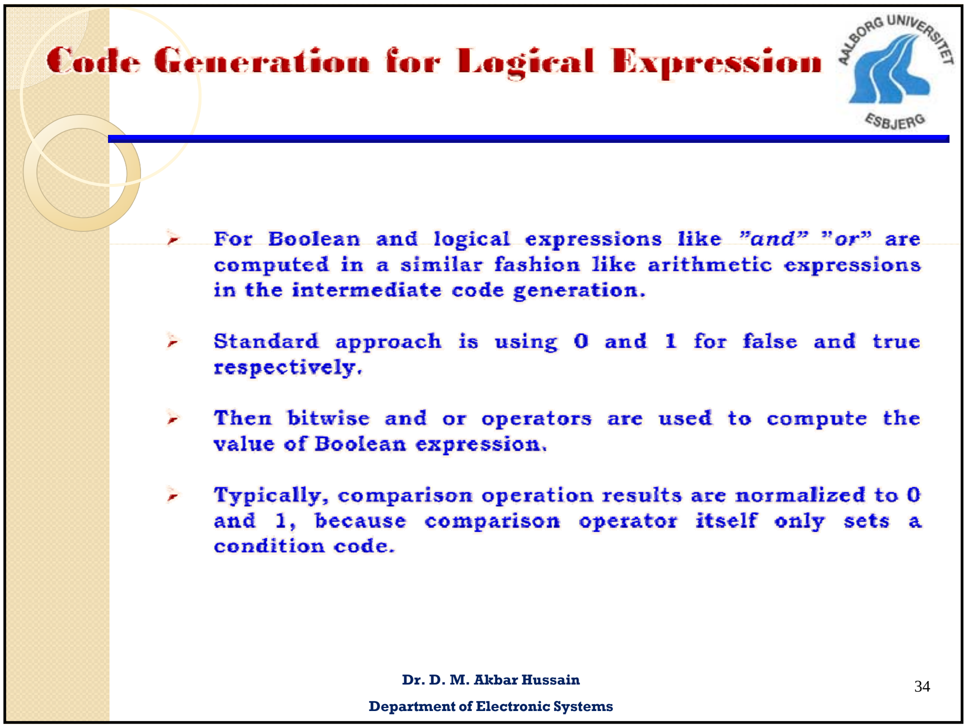
Generation of Labels

1. Some times jump to label must be generated prior to the definition of the label itself.
2. In IC generation, it creates no problem as the code generation routine can generate a forward jump and store it on stack until the label location is known.
3. In assembly code generation labels are simply passed on to assembler.
4. But for actual target code these labels must be resolved to absolute addresses.
5. Back-patching technique is typically used for such situations.



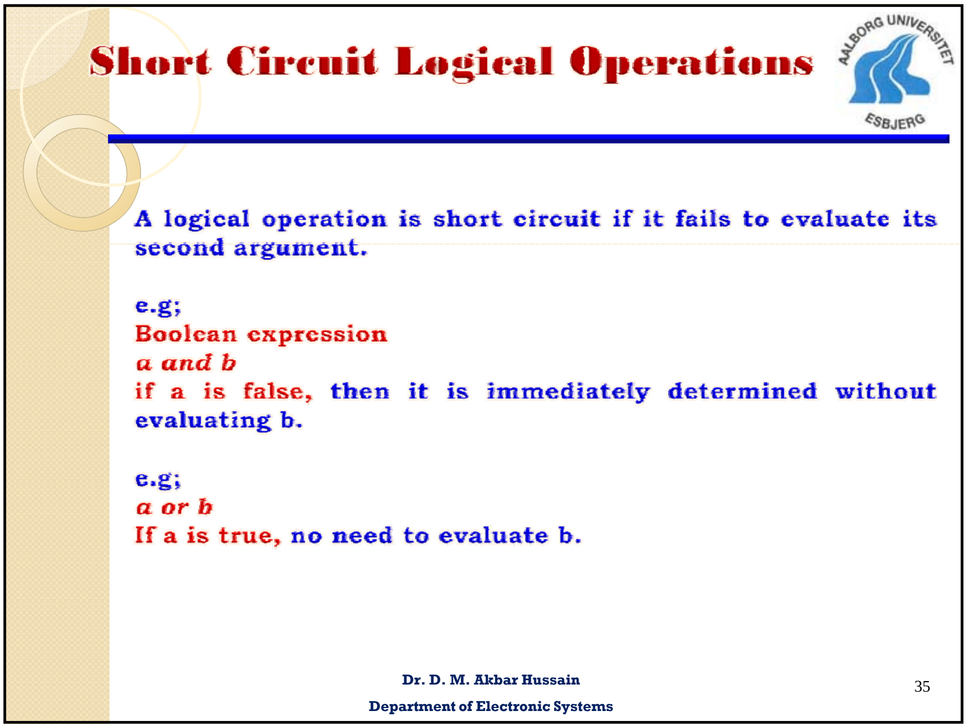
Back-patching

- A dummy jump instruction to a fake location is generated.
- Code is kept in a buffer or on stack or in a temporary file.
- When the label is known back-patching is performed.
- Additional consideration when machines (architecture) have two varieties of jumps, short jump or branch (within say 128 bytes) or long jump.
- Therefore, nop or multiple passes are used to condense the code.



Code Generation for Logical Expression

- For Boolean and logical expressions like *"and" "or"* are computed in a similar fashion like arithmetic expressions in the intermediate code generation.
- Standard approach is using 0 and 1 for false and true respectively.
- Then bitwise and or operators are used to compute the value of Boolean expression.
- Typically, comparison operation results are normalized to 0 and 1, because comparison operator itself only sets a condition code.

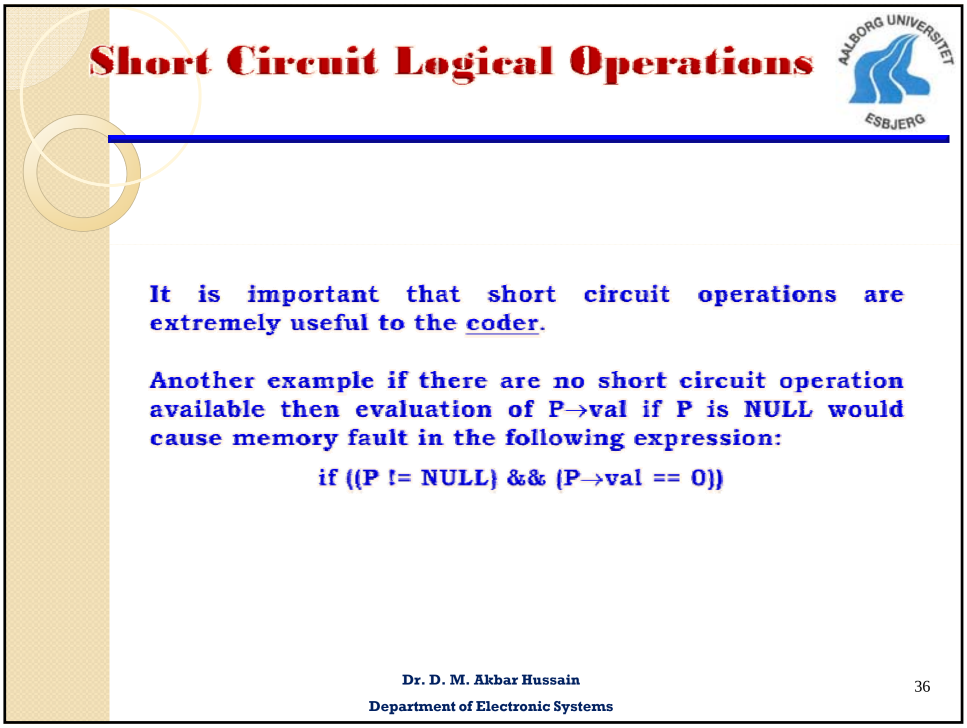


Short Circuit Logical Operations

A logical operation is short circuit if it fails to evaluate its second argument.

e.g;
Boolean expression
a and b
if a is false, then it is immediately determined without evaluating b.

e.g;
a or b
If a is true, no need to evaluate b.

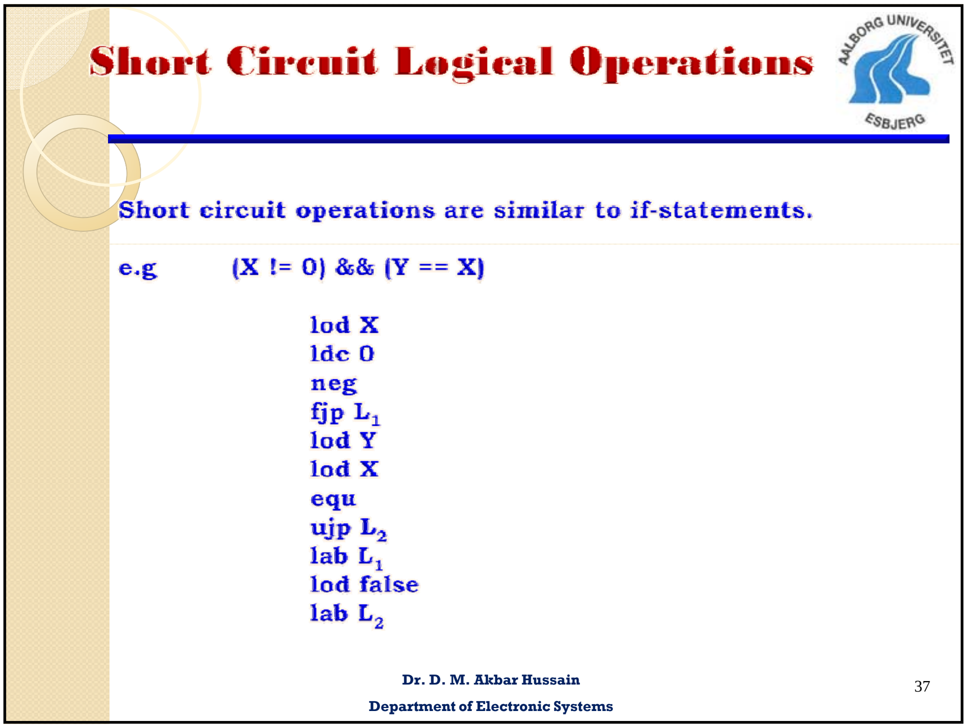


Short Circuit Logical Operations

It is important that short circuit operations are extremely useful to the coder.

Another example if there are no short circuit operation available then evaluation of P→val if P is NULL would cause memory fault in the following expression:

`if ((P != NULL) && (P→val == 0))`

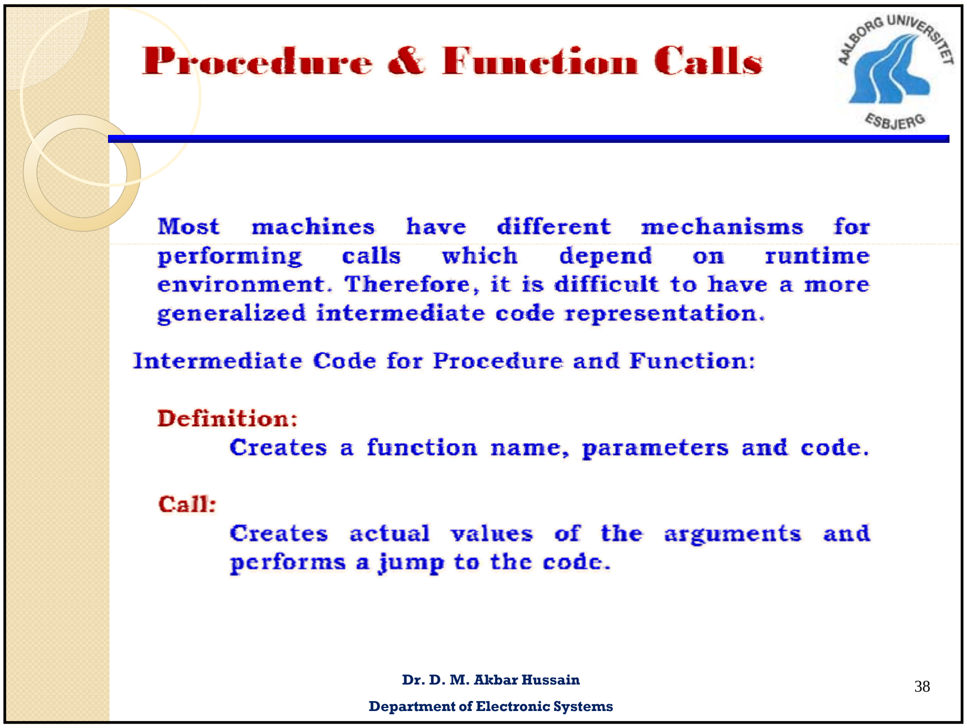


Short Circuit Logical Operations

Short circuit operations are similar to if-statements.

e.g (X != 0) && (Y == X)

lod X
ldc 0
neg
fjp L_1
lod Y
lod X
equ
ujp L_2
lab L_1
lod false
lab L_2



Procedure & Function Calls

Most machines have different mechanisms for performing calls which depend on runtime environment. Therefore, it is difficult to have a more generalized intermediate code representation.

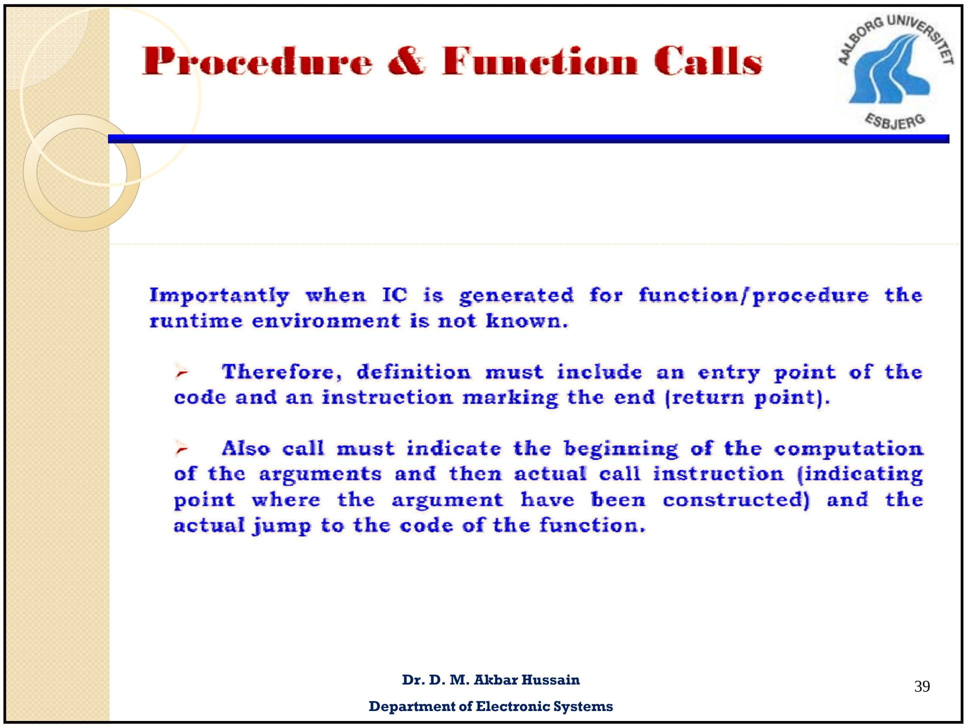
Intermediate Code for Procedure and Function:

Definition:

Creates a function name, parameters and code.

Call:

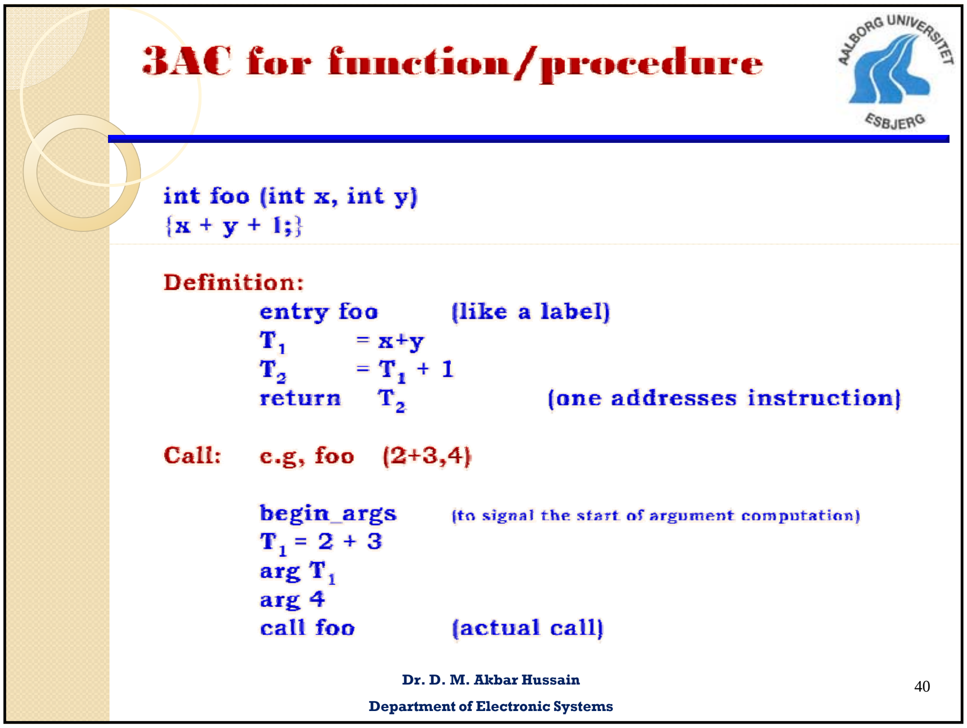
Creates actual values of the arguments and performs a jump to the code.



Procedure & Function Calls

Importantly when IC is generated for function/procedure the runtime environment is not known.

- **Therefore, definition must include an entry point of the code and an instruction marking the end (return point).**
- **Also call must indicate the beginning of the computation of the arguments and then actual call instruction (indicating point where the argument have been constructed) and the actual jump to the code of the function.**



3AC for function/procedure

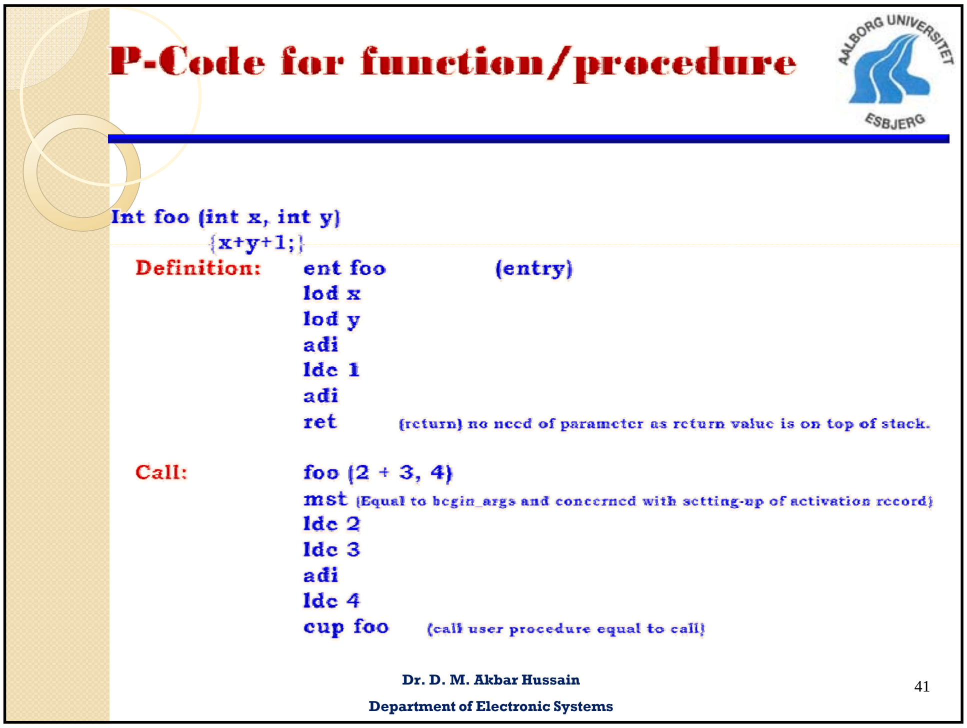
int foo (int x, int y)
{x + y + 1;}

Definition:

entry foo (like a label)
T_1 = x+y
T_2 = T_1 + 1
return T_2 (one addresses instruction)

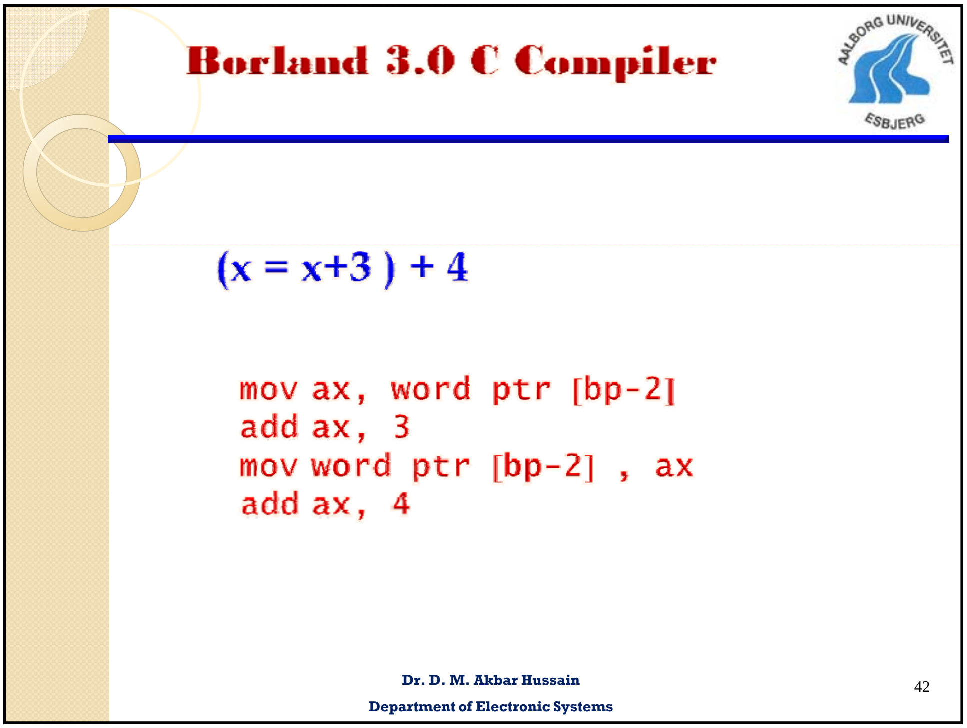
Call: c.g, foo (2+3,4)

begin_args (to signal the start of argument computation)
T_1 = 2 + 3
arg T_1
arg 4
call foo (actual call)



P-Code for function/procedure

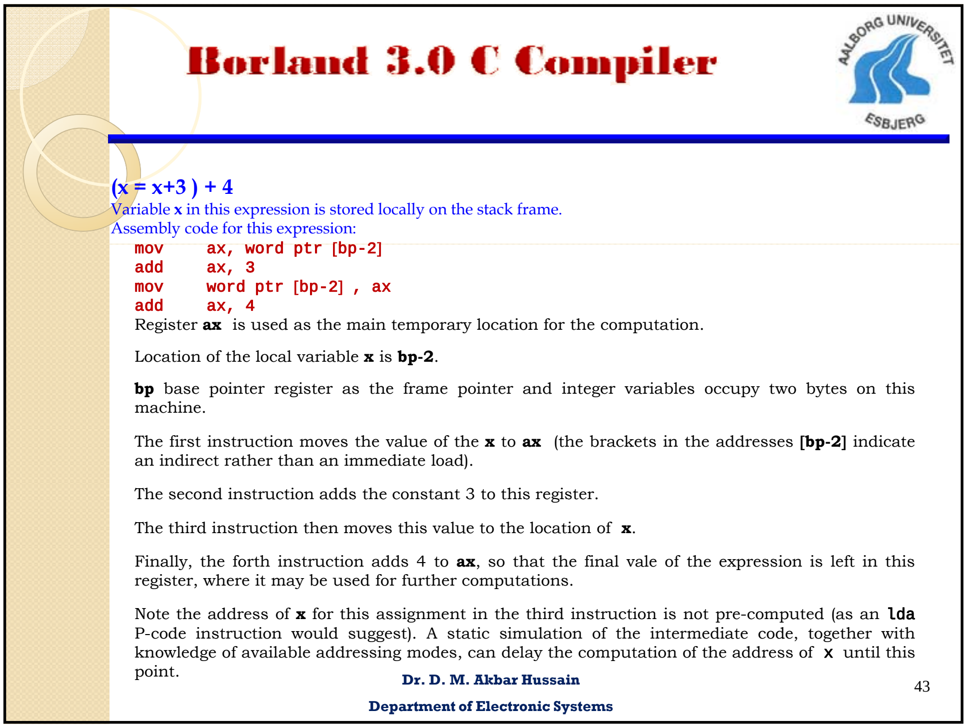
```
Int foo (int x, int y)
        {x+y+1;}
  Definition:     ent foo          (entry)
                  lod x
                  lod y
                  adi
                  ldc 1
                  adi
                  ret       (return) no need of parameter as return value is on top of stack.

  Call:           foo (2 + 3, 4)
                  mst (Equal to begin_args and concerned with setting-up of activation record)
                  ldc 2
                  ldc 3
                  adi
                  ldc 4
                  cup foo     (call user procedure equal to call)
```



Borland 3.0 C Compiler

(x = x+3) + 4

```
mov ax, word ptr [bp-2]
add ax, 3
mov word ptr [bp-2] , ax
add ax, 4
```



Borland 3.0 C Compiler

(x = x+3) + 4

Variable **x** in this expression is stored locally on the stack frame.
Assembly code for this expression:

```
mov     ax, word ptr [bp-2]
add     ax, 3
mov     word ptr [bp-2] , ax
add     ax, 4
```

Register **ax** is used as the main temporary location for the computation.

Location of the local variable **x** is **bp-2**.

bp base pointer register as the frame pointer and integer variables occupy two bytes on this machine.

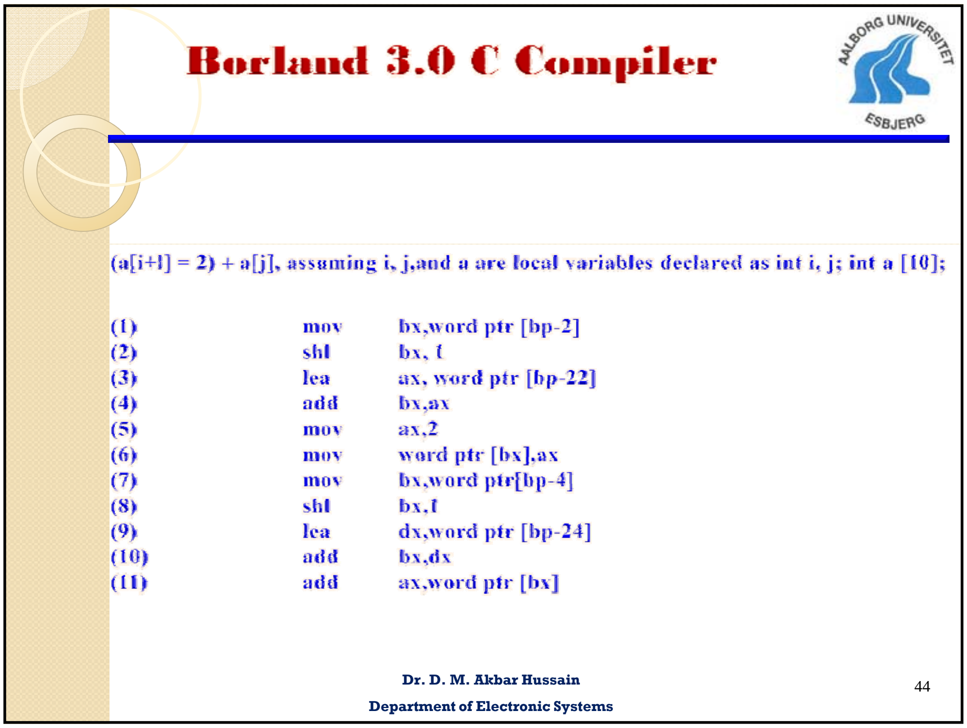
The first instruction moves the value of the **x** to **ax** (the brackets in the addresses **[bp-2]** indicate an indirect rather than an immediate load).

The second instruction adds the constant 3 to this register.

The third instruction then moves this value to the location of **x**.

Finally, the forth instruction adds 4 to **ax**, so that the final vale of the expression is left in this register, where it may be used for further computations.

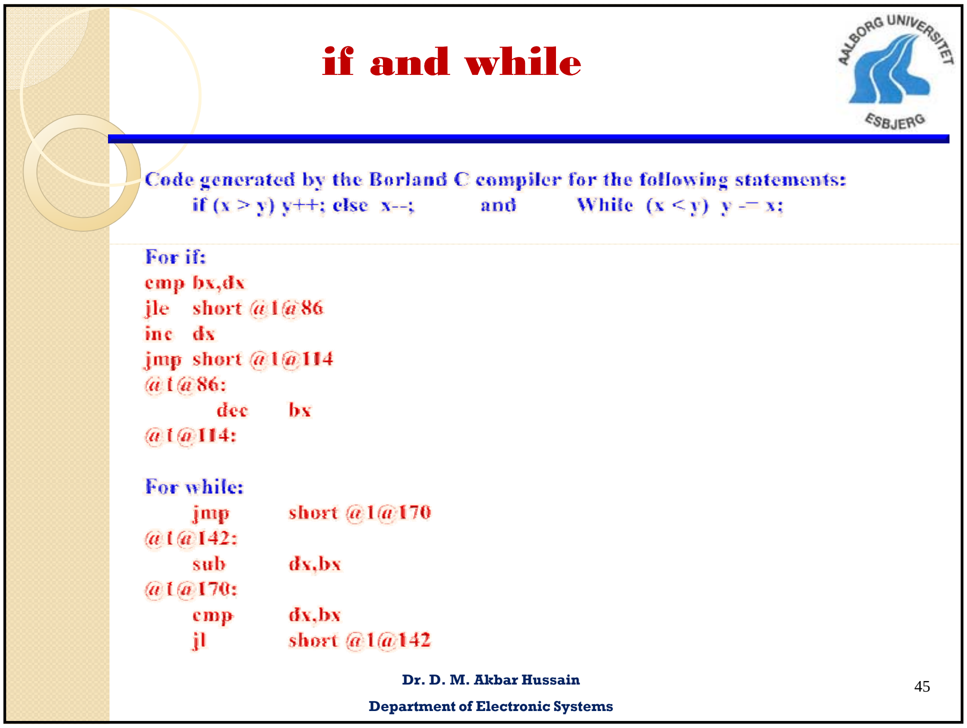
Note the address of **x** for this assignment in the third instruction is not pre-computed (as an `lda` P-code instruction would suggest). A static simulation of the intermediate code, together with knowledge of available addressing modes, can delay the computation of the address of `x` until this point.



Borland 3.0 C Compiler

(a[i+1] = 2) + a[j], assuming i, j,and a are local variables declared as int i, j; int a [10];

```
(1)          mov     bx,word ptr [bp-2]
(2)          shl     bx, 1
(3)          lea     ax, word ptr [bp-22]
(4)          add     bx,ax
(5)          mov     ax,2
(6)          mov     word ptr [bx],ax
(7)          mov     bx,word ptr[bp-4]
(8)          shl     bx,1
(9)          lea     dx,word ptr [bp-24]
(10)         add     bx,dx
(11)         add     ax,word ptr [bx]
```



if and while

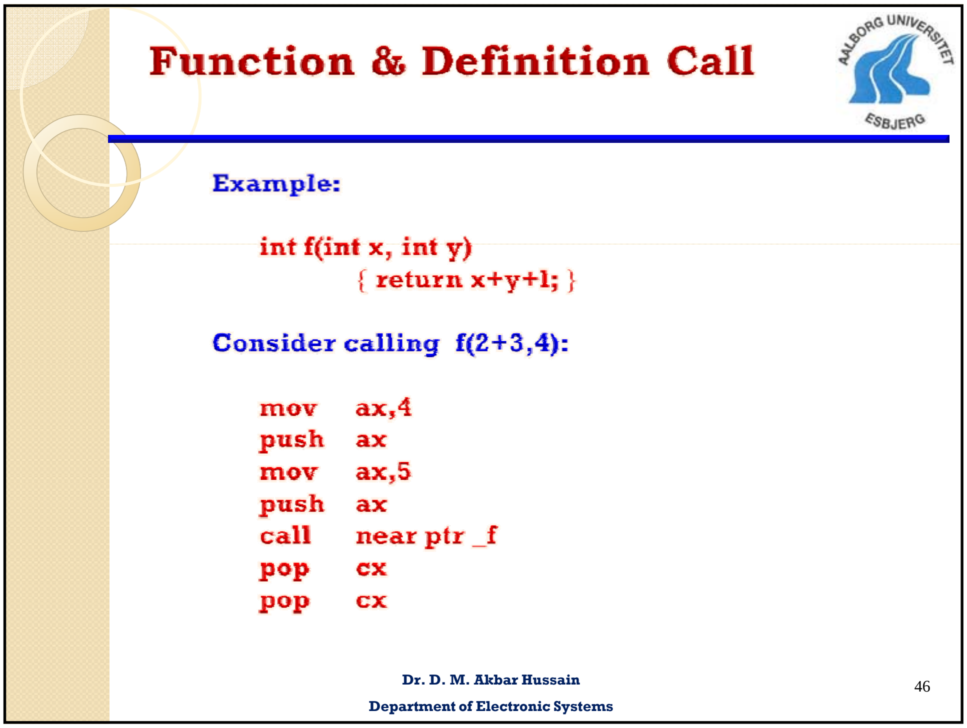
Code generated by the Borland C compiler for the following statements:
if (x > y) y++; else x--; and While (x < y) y -= x;

For if:

```
cmp bx,dx
jle   short @1@86
inc   dx
jmp  short @1@114
@1@86:
        dec      bx
@1@114:
```

For while:

```
      jmp          short @1@170
@1@142:
      sub          dx,bx
@1@170:
      cmp          dx,bx
      jl           short @1@142
```



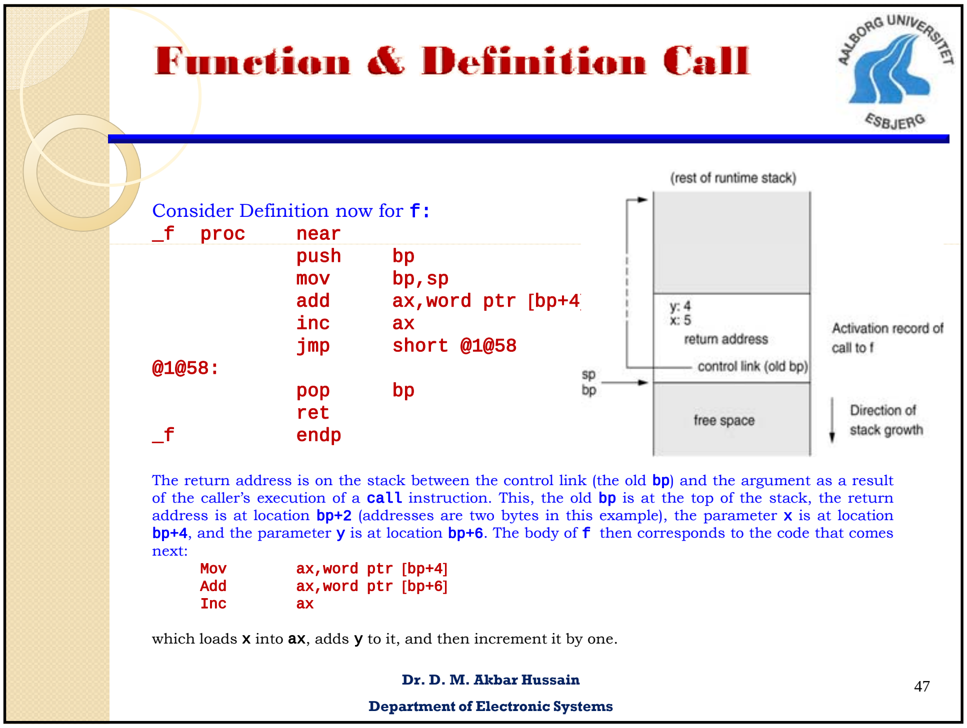
Function & Definition Call

Example:

```
int f(int x, int y)
        { return x+y+1; }
```

Consider calling f(2+3,4):

```
mov     ax,4
push    ax
mov     ax,5
push    ax
call    near ptr _f
pop     cx
pop     cx
```



Function & Definition Call

Consider Definition now for **f**:

```
_f   proc      near
               push     bp
               mov      bp,sp
               add      ax,word ptr [bp+4
               inc      ax
               jmp      short @1@58
@1@58:
               pop      bp
               ret
_f             endp
```

The return address is on the stack between the control link (the old **bp**) and the argument as a result of the caller's execution of a **call** instruction. This, the old **bp** is at the top of the stack, the return address is at location **bp+2** (addresses are two bytes in this example), the parameter **x** is at location **bp+4**, and the parameter **y** is at location **bp+6**. The body of **f** then corresponds to the code that comes next:

```
Mov       ax,word ptr [bp+4]
Add       ax,word ptr [bp+6]
Inc       ax
```

which loads **x** into **ax**, adds **y** to it, and then increment it by one.

Array References

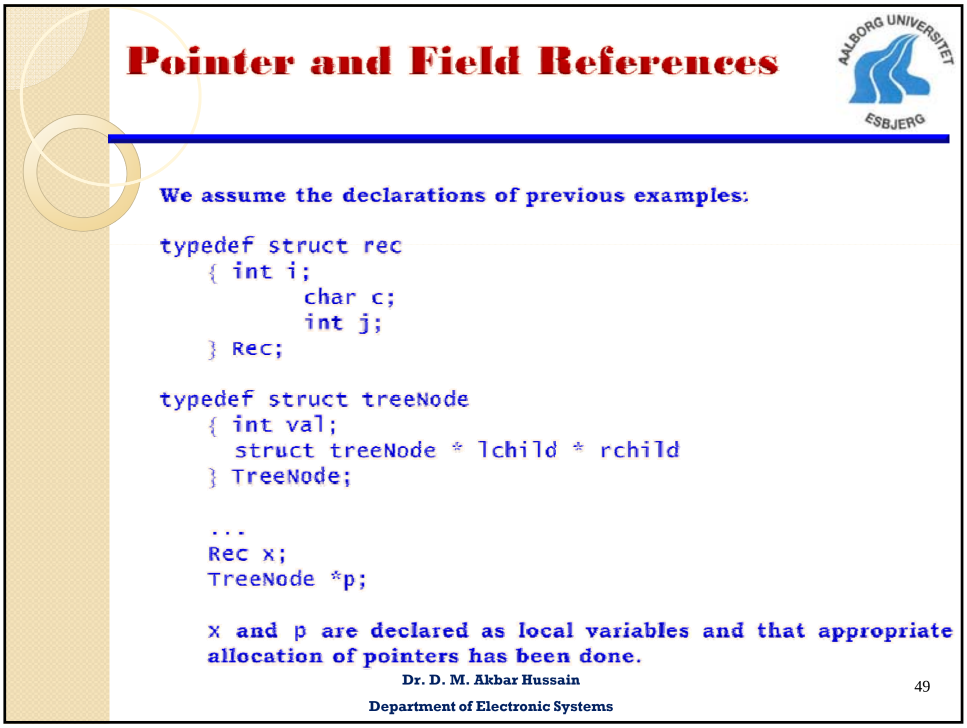
Example: (a[i+1] = 2) + a[j]

Assuming i, j,and a are local variables declared as

```
int i,j;
int a [10];
```

Borland C compiler generates the following assembly code:

```
(1)     mov     bx,word ptr [bp-2]
(2)     shl     bx, 1
(3)     lea     ax, word ptr [bp-22]
(4)     add     bx,ax
(5)     mov     ax,2
(6)     mov     word ptr [bx],ax
(7)     mov     bx,word ptr[bp-4]
(8)     shl     bx,1
(9)     lea     dx,word ptr [bp-24]
(10)    add     bx,dx
(11)    add     ax,word ptr [bx]
```

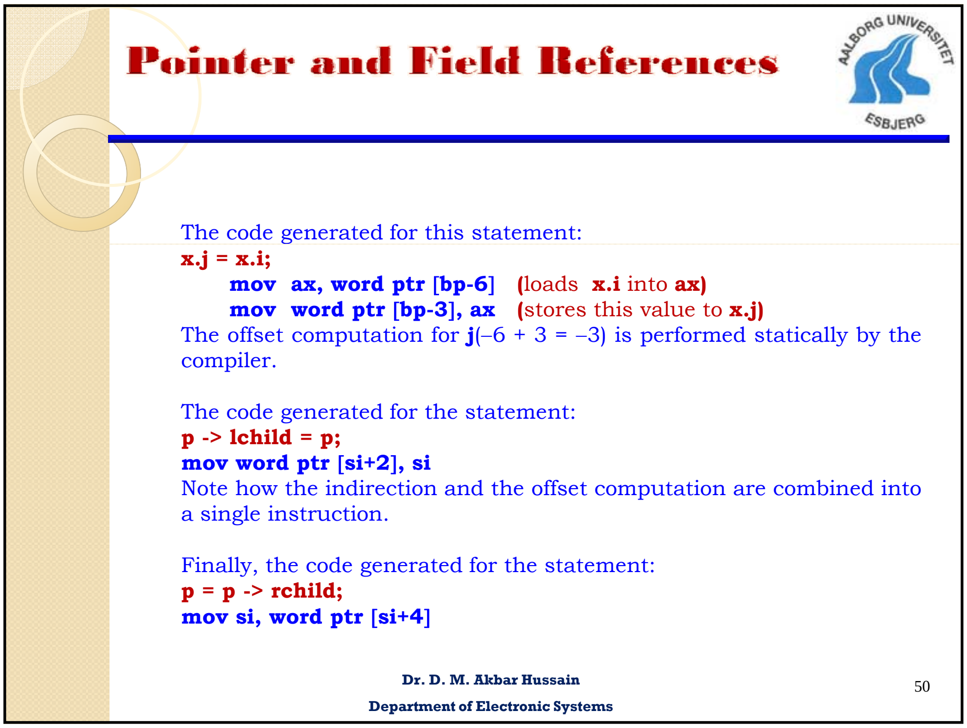


Pointer and Field References

We assume the declarations of previous examples:

```
typedef struct rec
    { int i;
            char c;
            int j;
    } Rec;

typedef struct treeNode
    { int val;
      struct treeNode * lchild * rchild
    } TreeNode;

    ...
    Rec x;
    TreeNode *p;
```

x and p are declared as local variables and that appropriate allocation of pointers has been done.

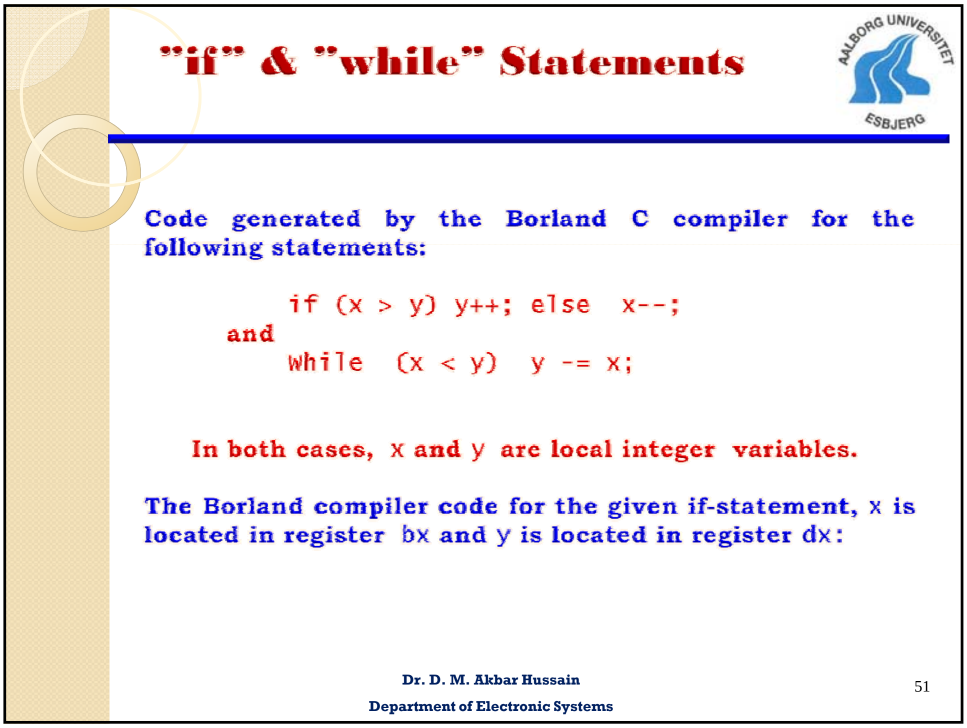


Pointer and Field References

The code generated for this statement:
x.j = x.i;
mov ax, word ptr [**bp-6**] (loads **x.i** into **ax)**
mov word ptr [**bp-3**]**, ax** (stores this value to **x.j)**
The offset computation for **j**(–6 + 3 = –3) is performed statically by the compiler.

The code generated for the statement:
p -> lchild = p;
mov word ptr [**si+2**]**, si**
Note how the indirection and the offset computation are combined into a single instruction.

Finally, the code generated for the statement:
p = p -> rchild;
mov si, word ptr [**si+4**]



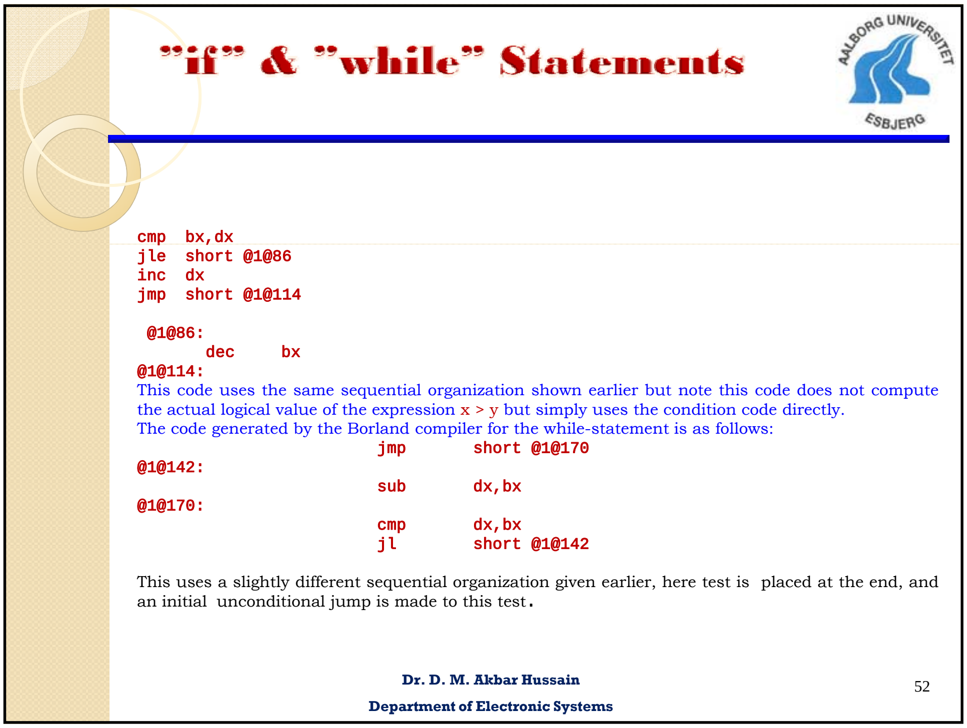
"if" & "while" Statements

Code generated by the Borland C compiler for the following statements:

```
if (x > y) y++; else  x--;
and
while  (x < y)  y -= x;
```

In both cases, x and y are local integer variables.

The Borland compiler code for the given if-statement, x is located in register bx and y is located in register dx:



"if" & "while" Statements

```
cmp  bx,dx
jle  short @1@86
inc  dx
jmp  short @1@114

 @1@86:
       dec     bx
@1@114:
```

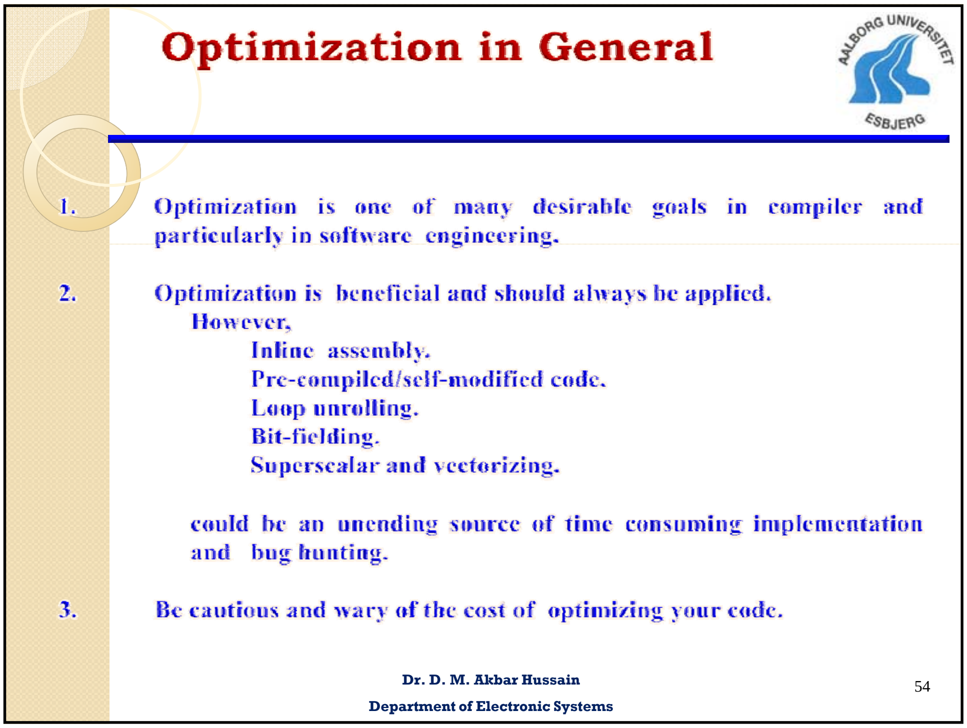
This code uses the same sequential organization shown earlier but note this code does not compute the actual logical value of the expression x > y but simply uses the condition code directly.
The code generated by the Borland compiler for the while-statement is as follows:

```
                         jmp       short @1@170
@1@142:
                         sub       dx,bx
@1@170:
                         cmp       dx,bx
                         jl        short @1@142
```

This uses a slightly different sequential organization given earlier, here test is placed at the end, and an initial unconditional jump is made to this test.

Code Optimization

1. Register Allocation
2. Unnecessary Operations
3. Costly Operations
4. Program Behavior Prediction (Profiling)



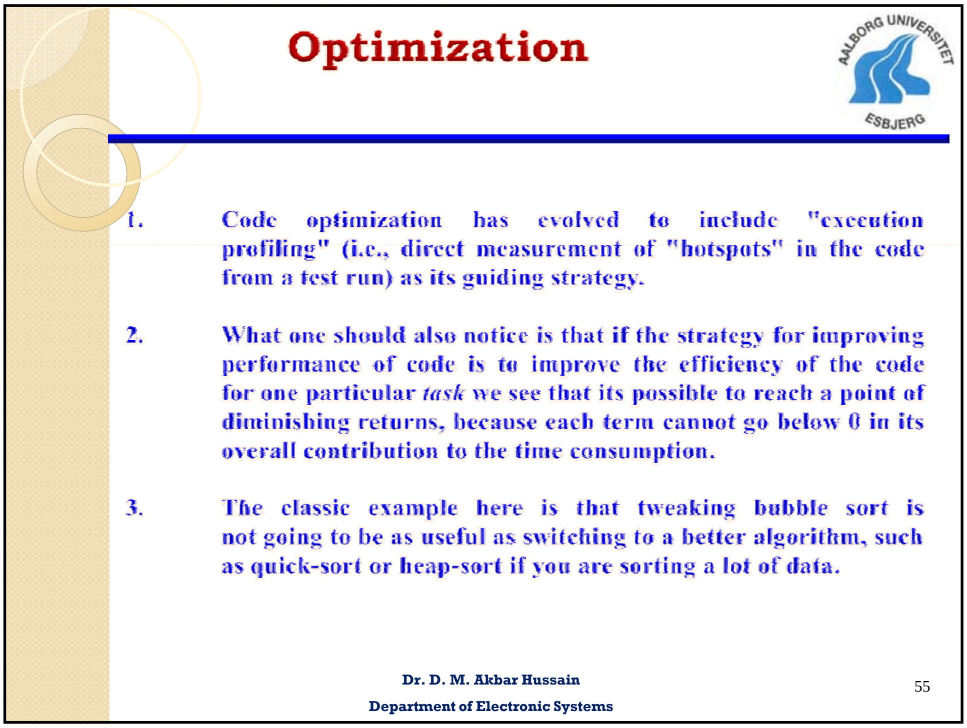
Optimization in General

1. Optimization is one of many desirable goals in compiler and particularly in software engineering.

2. Optimization is beneficial and should always be applied.
   However,
      Inline assembly.
      Pre-compiled/self-modified code.
      Loop unrolling.
      Bit-fielding.
      Superscalar and vectorizing.

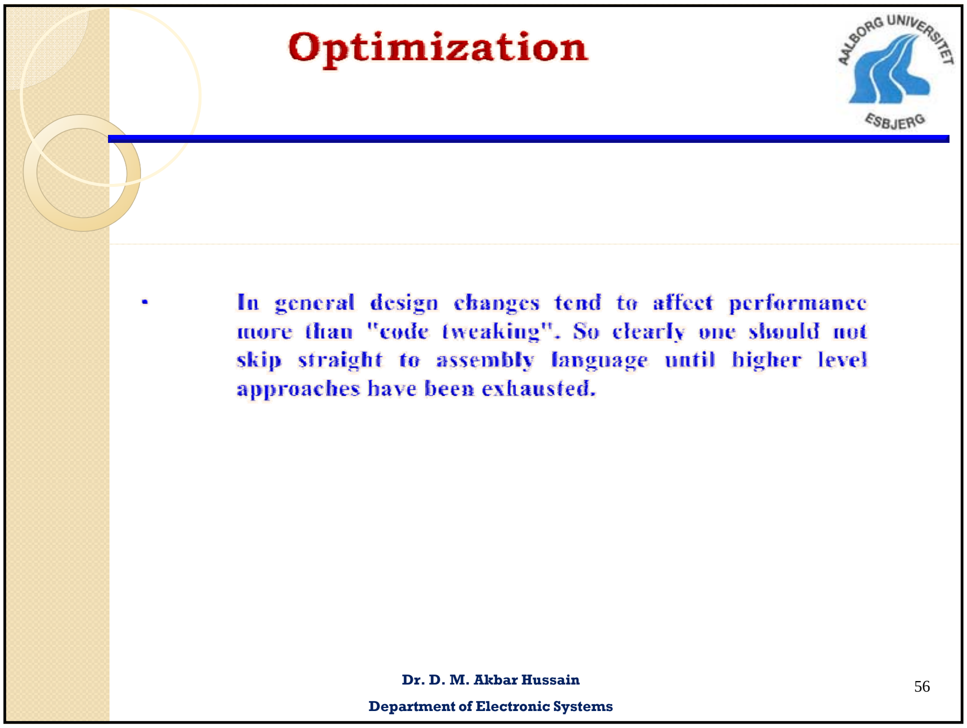
   could be an unending source of time consuming implementation and bug hunting.

3. Be cautious and wary of the cost of optimizing your code.



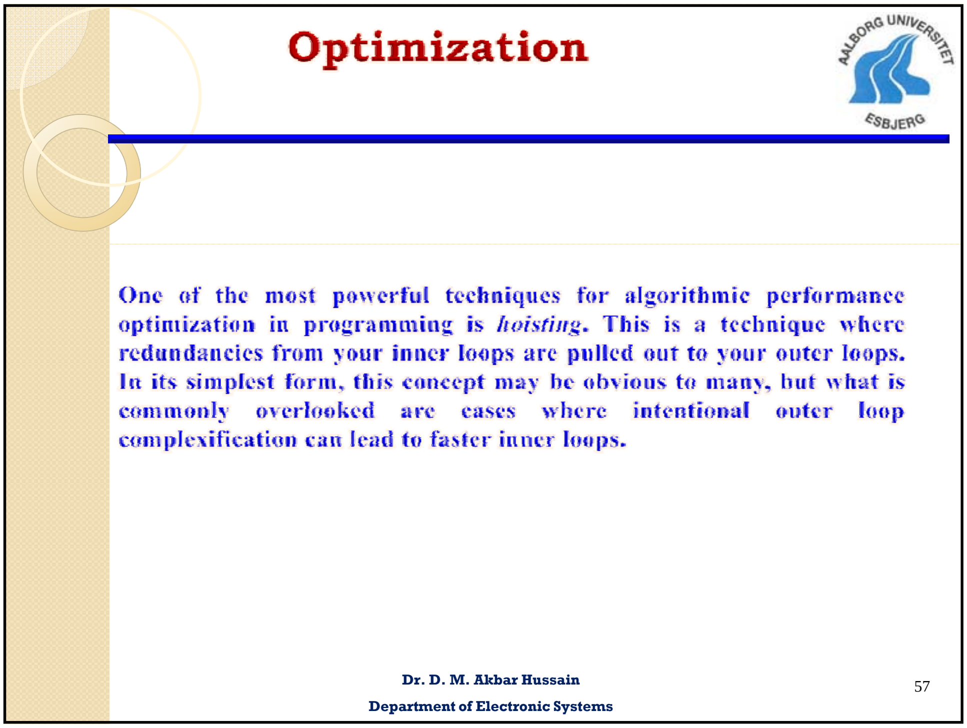
Optimization

1. Code optimization has evolved to include "execution profiling" (i.e., direct measurement of "hotspots" in the code from a test run) as its guiding strategy.
2. What one should also notice is that if the strategy for improving performance of code is to improve the efficiency of the code for one particular *task* we see that its possible to reach a point of diminishing returns, because each term cannot go below 0 in its overall contribution to the time consumption.
3. The classic example here is that tweaking bubble sort is not going to be as useful as switching to a better algorithm, such as quick-sort or heap-sort if you are sorting a lot of data.



Optimization

- In general design changes tend to affect performance more than "code tweaking". So clearly one should not skip straight to assembly language until higher level approaches have been exhausted.



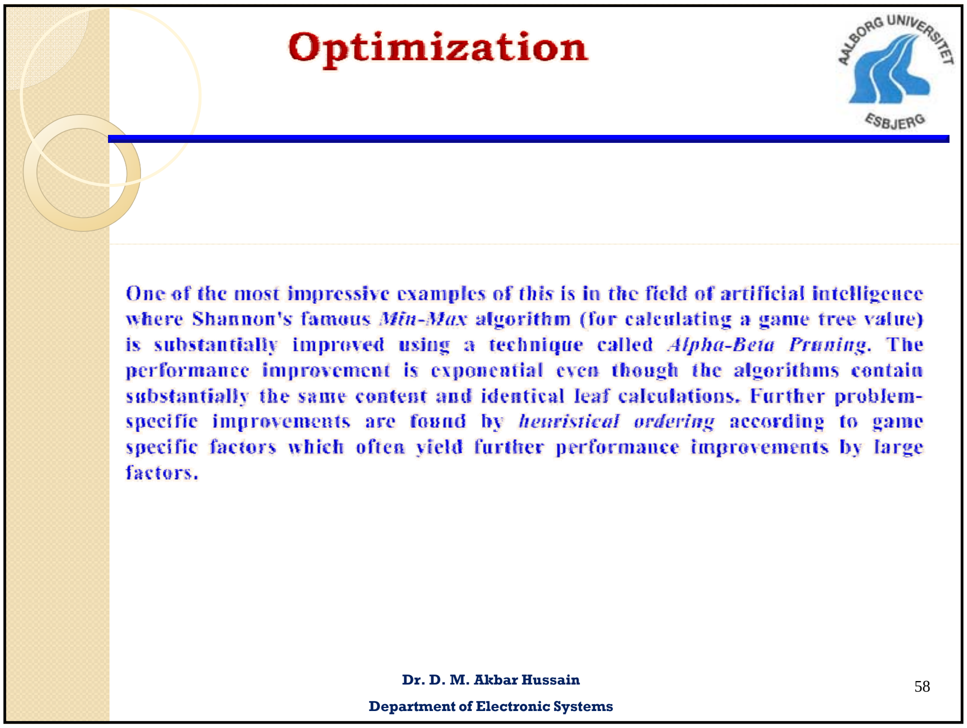
Optimization

One of the most powerful techniques for algorithmic performance optimization in programming is *hoisting*. This is a technique where redundancies from your inner loops are pulled out to your outer loops. In its simplest form, this concept may be obvious to many, but what is commonly overlooked are cases where intentional outer loop complexification can lead to faster inner loops.

Dr. D. M. Akbar Hussain

Department of Electronic Systems

57



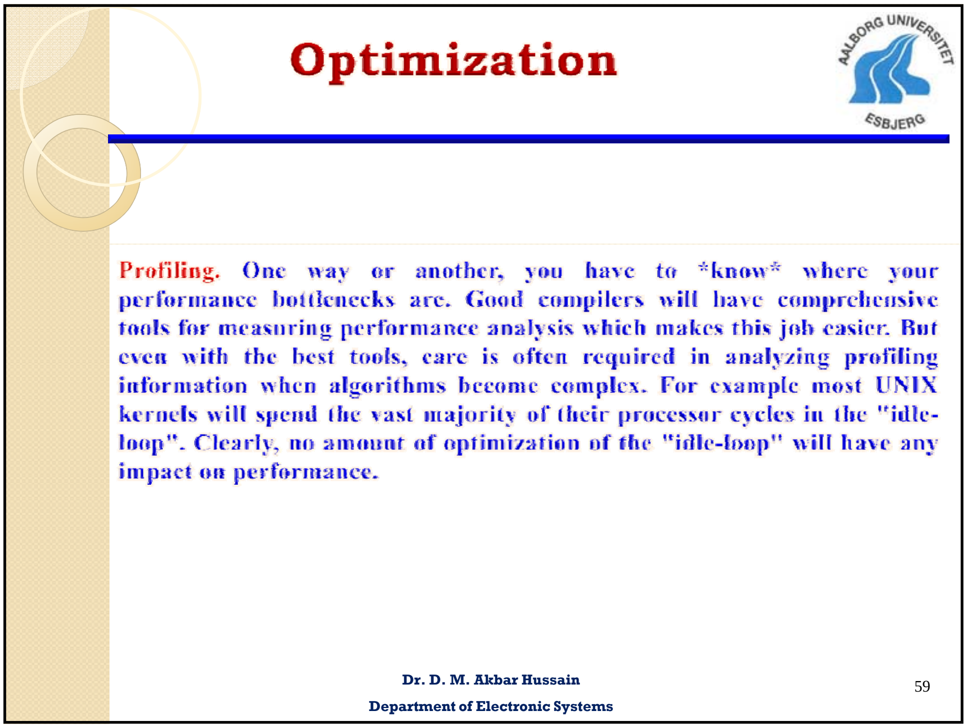
Optimization

One of the most impressive examples of this is in the field of artificial intelligence where Shannon's famous *Min-Max* algorithm (for calculating a game tree value) is substantially improved using a technique called *Alpha-Beta Pruning*. The performance improvement is exponential even though the algorithms contain substantially the same content and identical leaf calculations. Further problem-specific improvements are found by *heuristical ordering* according to game specific factors which often yield further performance improvements by large factors.

Dr. D. M. Akbar Hussain

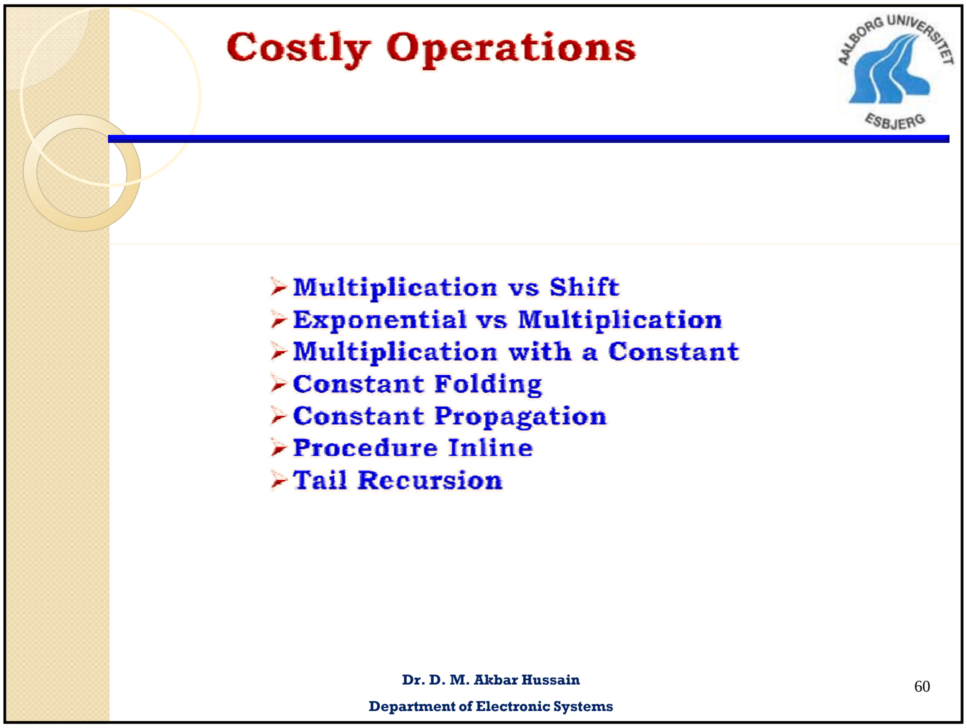
Department of Electronic Systems

58



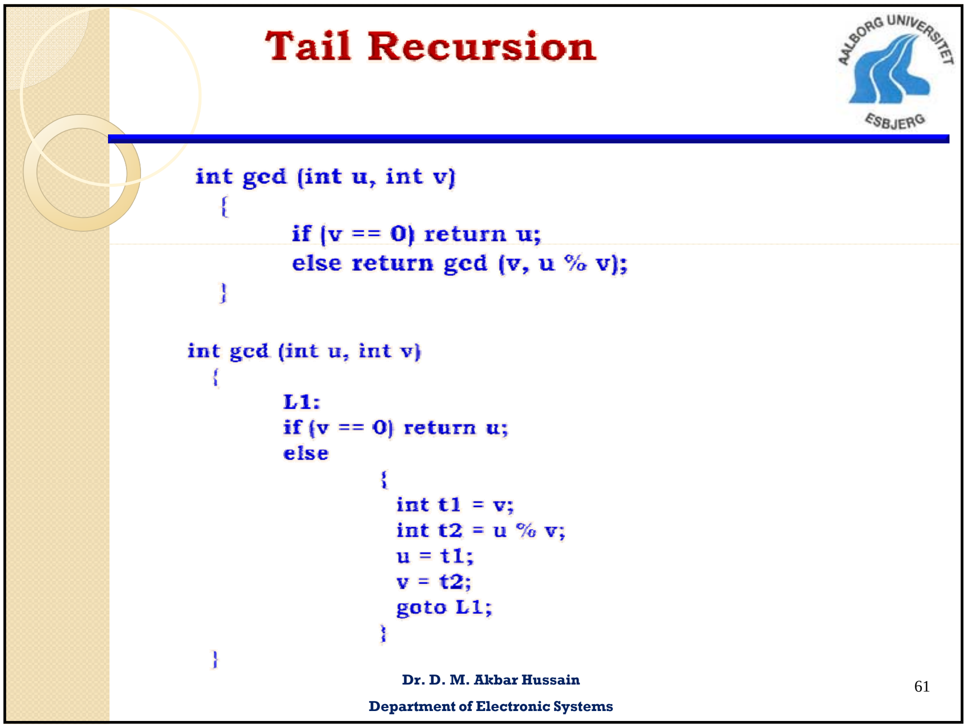
Optimization

Profiling. One way or another, you have to *know* where your performance bottlenecks are. Good compilers will have comprehensive tools for measuring performance analysis which makes this job easier. But even with the best tools, care is often required in analyzing profiling information when algorithms become complex. For example most UNIX kernels will spend the vast majority of their processor cycles in the "idle-loop". Clearly, no amount of optimization of the "idle-loop" will have any impact on performance.



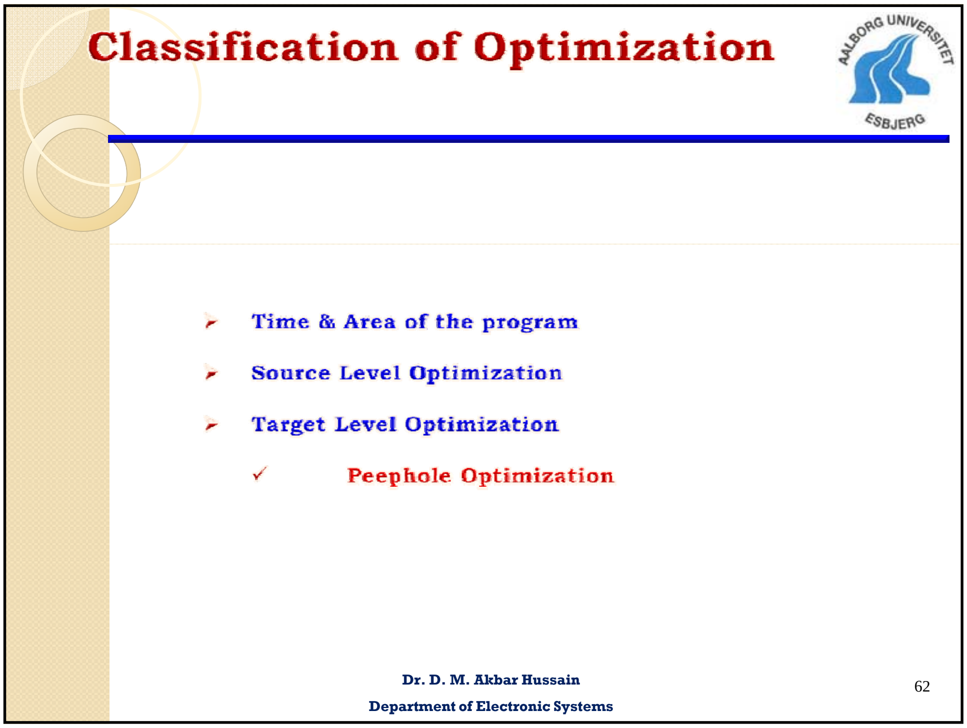
Costly Operations

- Multiplication vs Shift
- Exponential vs Multiplication
- Multiplication with a Constant
- Constant Folding
- Constant Propagation
- Procedure Inline
- Tail Recursion



Tail Recursion

```
int gcd (int u, int v)
   {
          if (v == 0) return u;
          else return gcd (v, u % v);
   }

int gcd (int u, int v)
   {
          L1:
          if (v == 0) return u;
          else
                 {
                   int t1 = v;
                   int t2 = u % v;
                   u = t1;
                   v = t2;
                   goto L1;
                 }
   }
```



Classification of Optimization

- Time & Area of the program
- Source Level Optimization
- Target Level Optimization
  - ✓ Peephole Optimization

Optimization Spillage

```
x = 1;
.....
y = 0;
......
if (y) x = 10;
.......
if (x) y = 100;
```

Constant Propagation

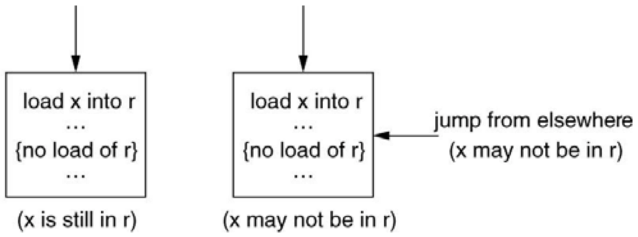
```
x = 1;
...
y = 0;
....
if (0) x = 10;
....
if (x) y = 100;
```

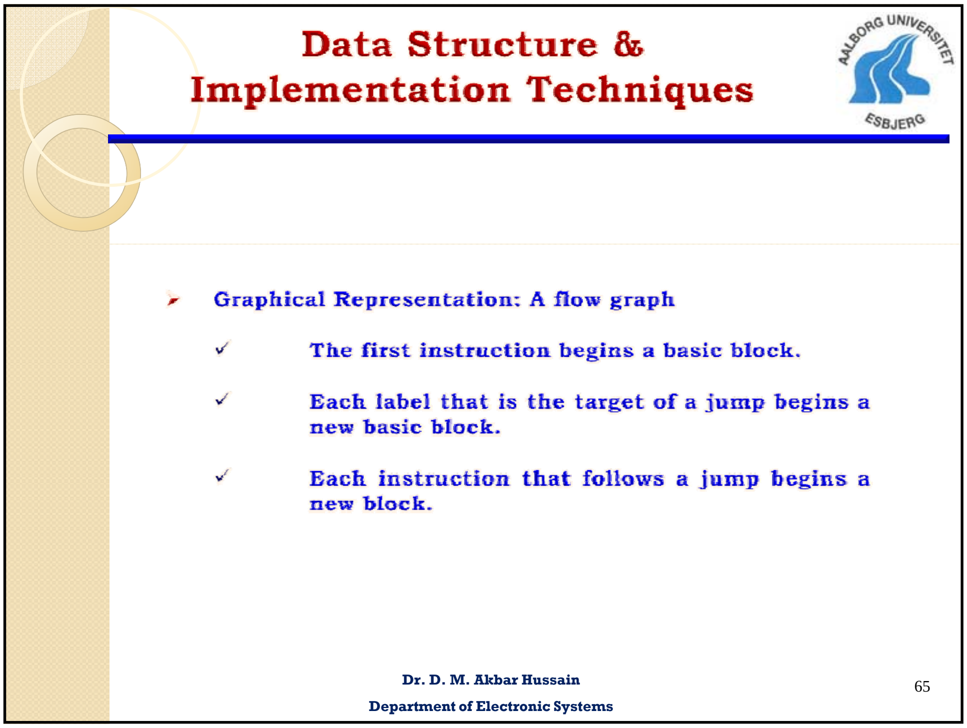
Un-reachable

```
x = 1;
...
y = 0;
....
if (x) y = 100;
```

Local, Global & Inter-processor Optimization

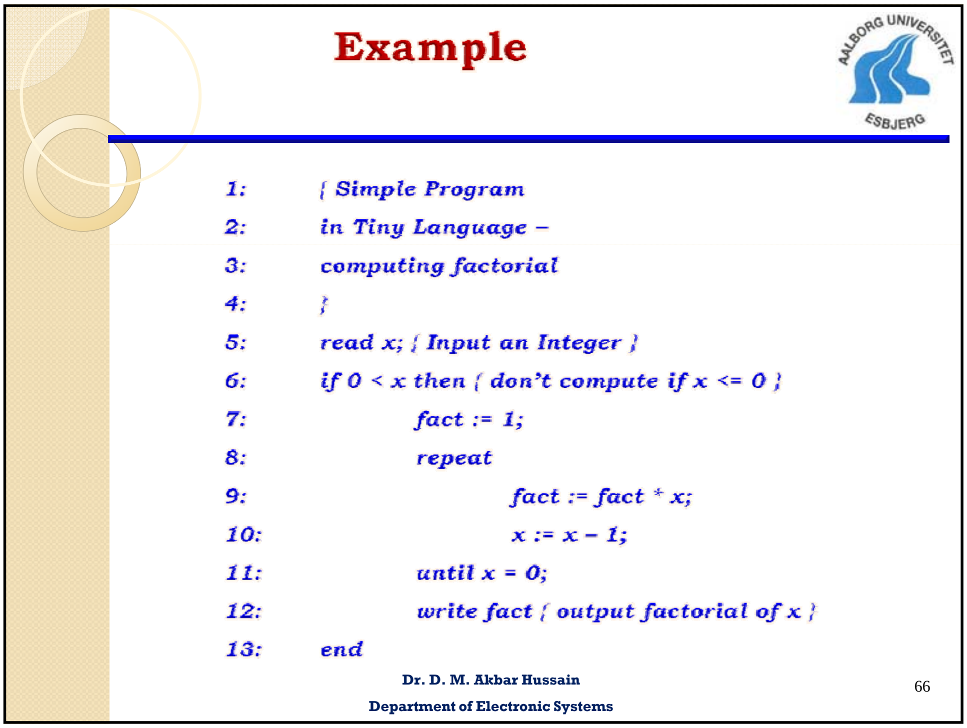
- Basic Block
- Global (Limited to procedure & functions)
- Beyond boundaries of procedures/functions





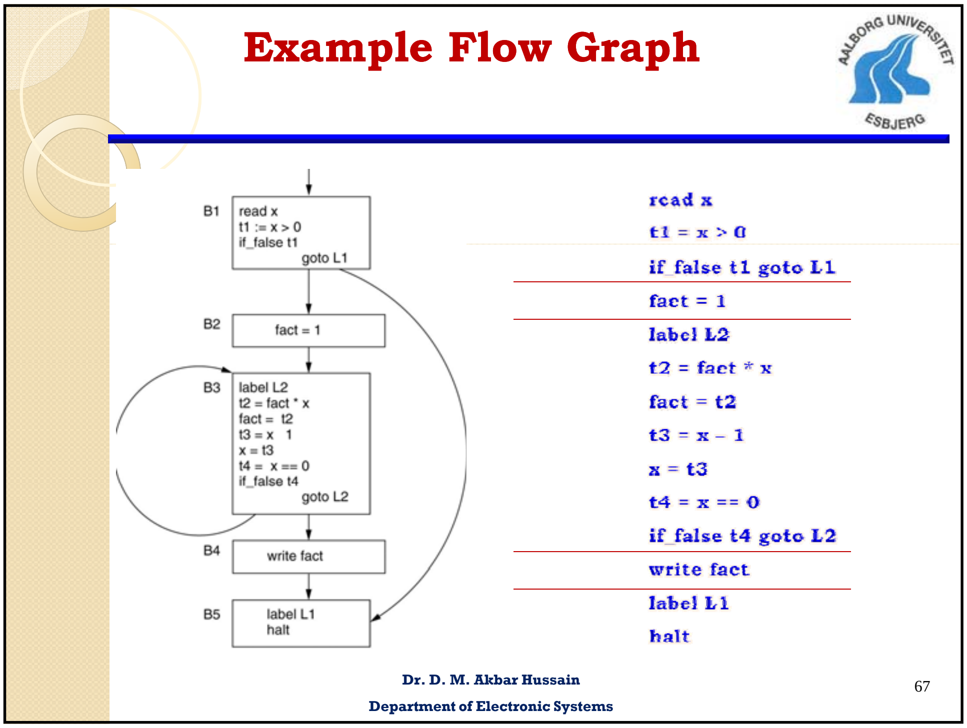
Data Structure & Implementation Techniques

- Graphical Representation: A flow graph
  - ✓ The first instruction begins a basic block.
  - ✓ Each label that is the target of a jump begins a new basic block.
  - ✓ Each instruction that follows a jump begins a new block.



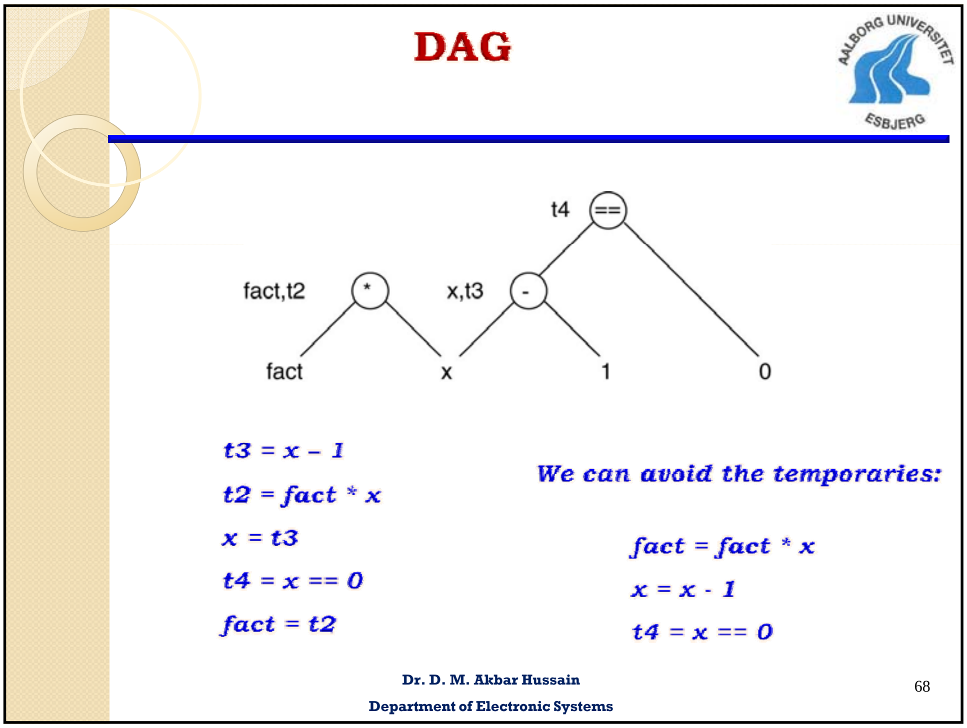
Example

```
{ Simple Program
in Tiny Language –
computing factorial
}
read x; { Input an Integer }
if 0 < x then { don't compute if x <= 0 }
          fact := 1;
          repeat
                    fact := fact * x;
                    x := x – 1;
          until x = 0;
          write fact { output factorial of x }
end
```



Example Flow Graph

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x – 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```



DAG

t3 = x – 1

*t2 = fact * x*

x = t3

t4 = x == 0

fact = t2

We can avoid the temporaries:

*fact = fact * x*

x = x - 1

t4 = x == 0