Optimization Techniques in C

Team Emertxe

Optimization Techniques

Basic Concepts

Programming Algorithm and Techniques

Optimization Techniques
Basic Concepts

- ✓ What is Optimization
- ✓ Methods
- ✓ Space and Time Optimization
- ✓ Evaluation
- ✓ When not to Optimize
- ✓ Identify Critical Area
- ✓ Confusion to Source Level Debugger

EMERTXE

Optimization Techniques Methods

- ✓ Choice of Complier
- ✓ Compiler setting
- ✓ Programming Algorithm and Techniques
- ✓ Rewrite Program in Assembly
- ✓ Code Profilers
- ✓ Disassemble Code Analysis

Optimization Techniques
Programming Algorithm & Techniques

- ✓ Data Handling
- ✓ Flow Control
- ✓ Other Handling

Optimization Techniques
Data Handling

- ✓ RAM Usage
- ✓ Data Type Usage
- ✓ Avoid Type Conversions
- ✓ Unsigned Advantage
- ✓ Float and Double
- ✓ Constant and Volatile
- ✓ Data Alignment – Arrangement and Packing
- ✓ Pass by Reference
- ✓ Register Store and Restore

Optimization Techniques
Flow Control

- ✓ Use of Switch Statement
- ✓ Inline Function
- ✓ Loop Unrolling
- ✓ Loop Hoisting
- ✓ Loop Overhead

Optimization Techniques
Other Handling

- ✓ Use of Operators
- ✓ Inline Assembly
- ✓ Fixed Point and Floating Point

Definition

Optimization is a process of improving efficiency of a program in time (speed) or space (size)

Basic illustration

Time and Space Optimization

```
//Faster Speed

main()
{
    ...
    XXXXX
    YYYYY
    ZZZZZ
    ...
    XXXXX
    YYYYY
    ZZZZZ
    ...
    XXXXX
    YYYYY
    ZZZZZ
    ...
}
```

```
//Smaller size
main()
{
    ...
    function();
    ...
    function();
    ...
    function();
    ...
}

void function()
{
    XXXXX
    YYYYY
    ZZZZZ
}
```

EMERTXE

Cont..

- ✓ It can be observed that optimization with one method may affect the other.
- ✓ A general phenomenon is faster operating code will have a bigger code size, whereas smaller code size will have a slower execution speed. However this may not be always true.

Evaluation

- ✓ A program's optimization level can be evaluated based on the measurement of:
- ✓ Total code size
- ✓ Total number of execution cycles (time taken).
- ✓ These are determined by the basic component of a program, which is the assembly code (Instruction set / Op-code / Mnemonic).

Cont..

- ✓ In the MCU manual, these assembly codes characteristic are detailed:
- ✓ Instruction length -> Determine Code Size
- ✓ Number of execution states -> Determine Execution Speed

Example:

Mnemonic	Instruction length (bytes)	Number of execution states	Remarks
MOV.B Rs, Rd	2	2	Register usage has faster execution
MOV.W Rs, @Rd	2	6	
JSR @aa:16	4	8	8-bit absolute address jump take up smaller space
JSR @@aa:8	2	8	

Do Not Optimize

It is suggested that unless necessary, optimization shall be omitted. This process should be planned for, and not done at the end of the development cycle, whereby most scenarios had been tested. This may cause changes to the initial design and introduce more wastage of time and resources in debugging.

Data Type Usage

The use of correct data type is important in a recursive calculation or large array processing. The extra size, which is not required, is taking up much space and processing time.

Example

- Speed concern:- Byte multiplication - MULXU .B R1L,R2L - take up 12 cycles
  - Word multiplication - MULXU.W R1,ER2 - take up 20 cycles
- Size concern: - char data_collect[100];
  - long data_collect[100]; -take up 4 times more spaces

RAM Usage

- ✓ Shortage of RAM space is a common concern. The nominal RAM size for most 8-bit MCU is a mere 1 to 4K bytes size.
- ✓ Three main components of RAM are:
  - Stack
  - Heap
  - Global data
- ✓ The reduction in one component will enable the increase in the others. Unlike stack and heap, which are dynamic in nature, global data is fixed in size.
- ✓ Programmers may like to consider the reduction of global data usage, and place more emphasis on local variables control in stack and registers.

I/O considerations

Instead of accessing external bus for two times, it may be better to read the data as a word (16-bit external data bus), and process it as a byte.

Use Variables of the Same Type for Processing

- ✓ Programmers should plan to use the same type of variables for processing. Type conversion must be avoided.
- ✓ Otherwise, precious cycles will be waste to convert one type to another (Unsigned and signed variables are considered as different types).

Use of Unsigned Type

- ✓ All variables must be defined as “unsigned” unless mathematical calculation for the signed bit is necessary.
- ✓ The “signed-bit” may create complication, unwanted failure, slower processing and extra ROM size.

Float and Double

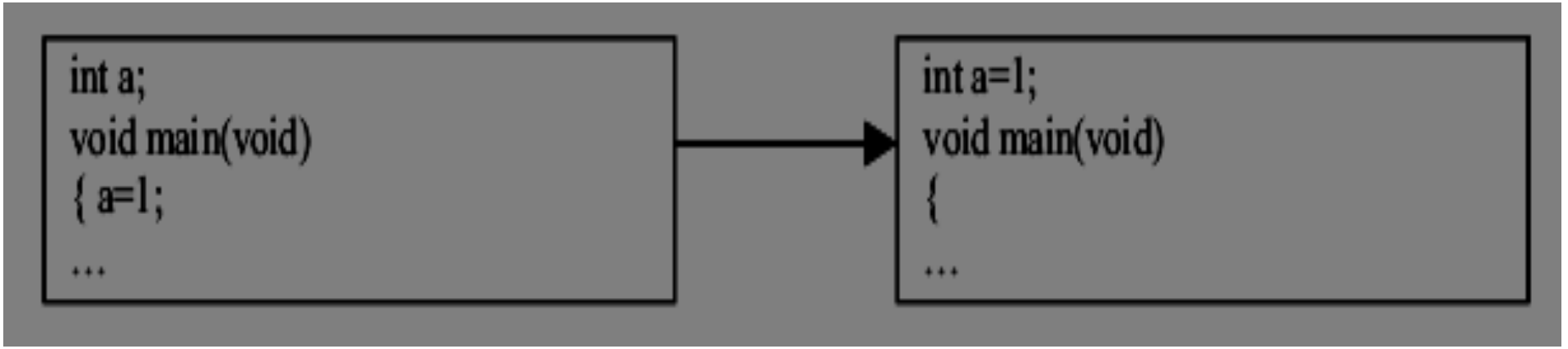
- ✓ Maximum value of Float = 0x7F7F FFFF
- ✓ Maximum value of Double = 0x7F7F FFFF FFFF FFFF
- ✓ To avoid the unnecessary type conversion or confusion, programmers can assign the letter “f” following the numeric value.
- ✓ x = y + 0.2f;

Data Declaration - Constant

- ✓ The “const” keyword is to define the data as a constant, which will allocate it in the ROM space.
- ✓ Otherwise a RAM space will also be reserved for this data. This is unnecessary as the constant data is supposed to be read-only.

Data Initialization at Declaration

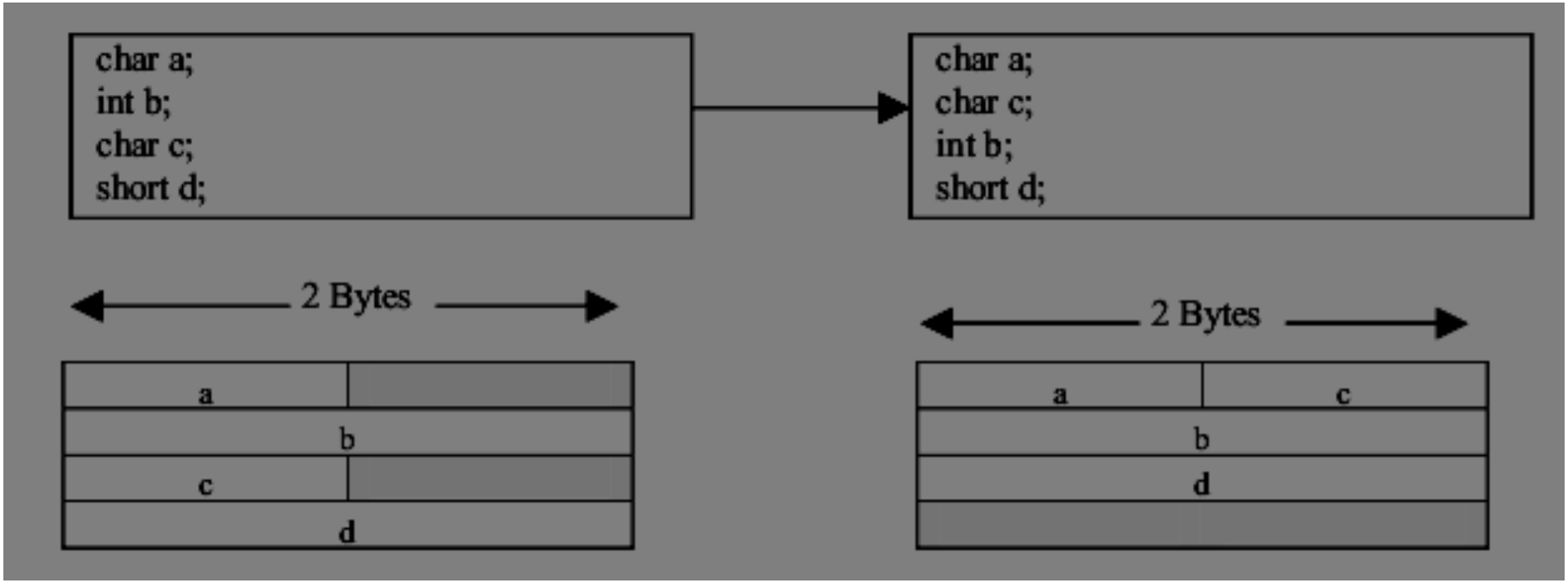
Data should be initialized at declaration.



Data Definition
Arrangement and Packing

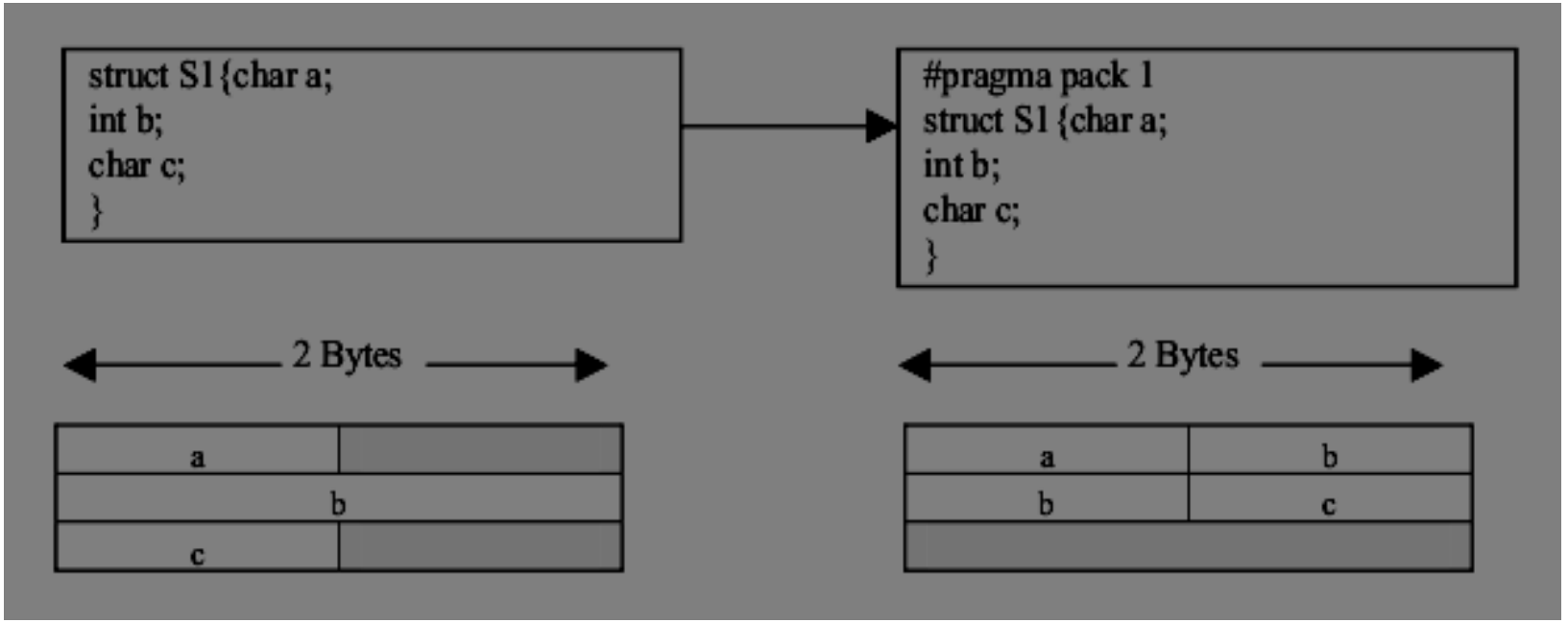
- ✓ The declaration of the components in a structure will determine how the components are being stored.
- ✓ Due to the memory alignment, it is possible to have dummy area within the structure. It is advised to place all similar size variables in the same group.

Data Arrangement



Structure Padding

- ✓ As the structure is packed, integer b will not be aligned.
- ✓ This will improve the RAM size but operational speed will be degraded, as the access of ‘b’ will take up two cycles.



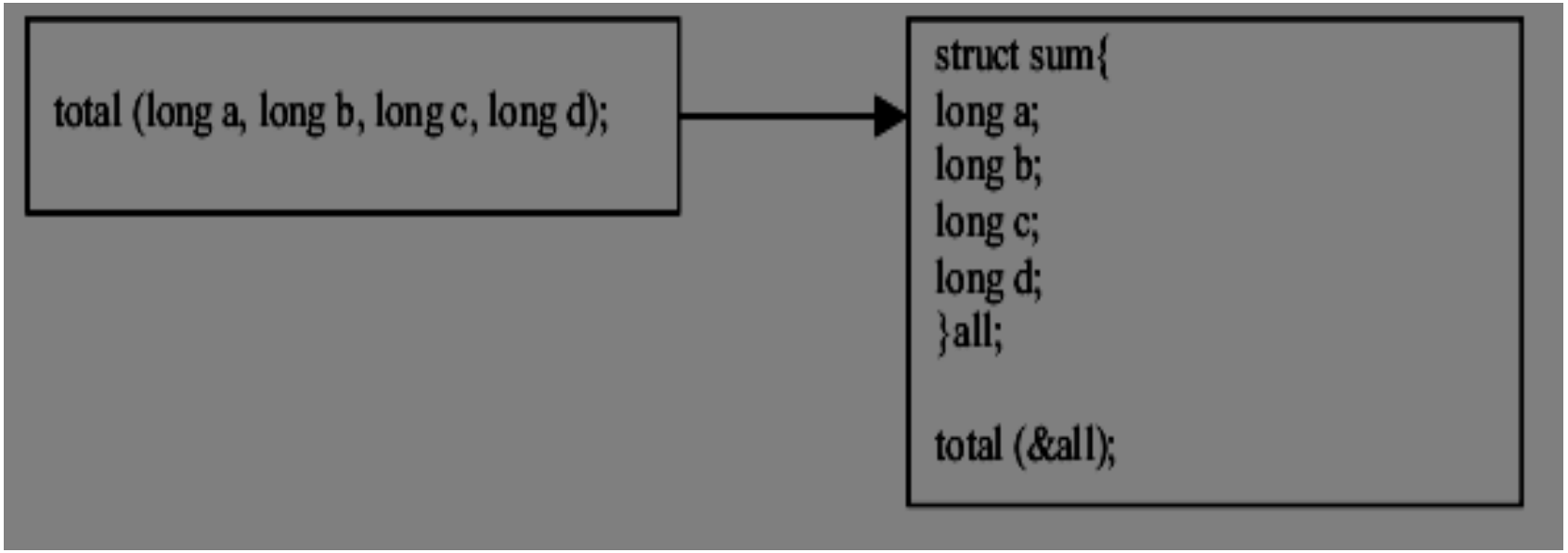
EMERTXE

Global and Local Variables

- ✓ Local variable is preferred over the global variable in a function.
- ✓ Generally, global variables are stored in the memory, whereas, local variables are stored in the register.
- ✓ Since register access is faster than the memory access, implementing local variables will improve speed operation.
- ✓ Moreover, code portability also encourages the use of local variables.
- ✓ However if there are more local variables than the available registers, the local variables will be temporary stored in the stack.

Passing Reference as Parameters

- ✓ Larger numbers of parameters may be costly due to the number of pushing and popping actions on each function call.
- ✓ It is more efficient to pass structure reference as parameters to reduce this overhead.

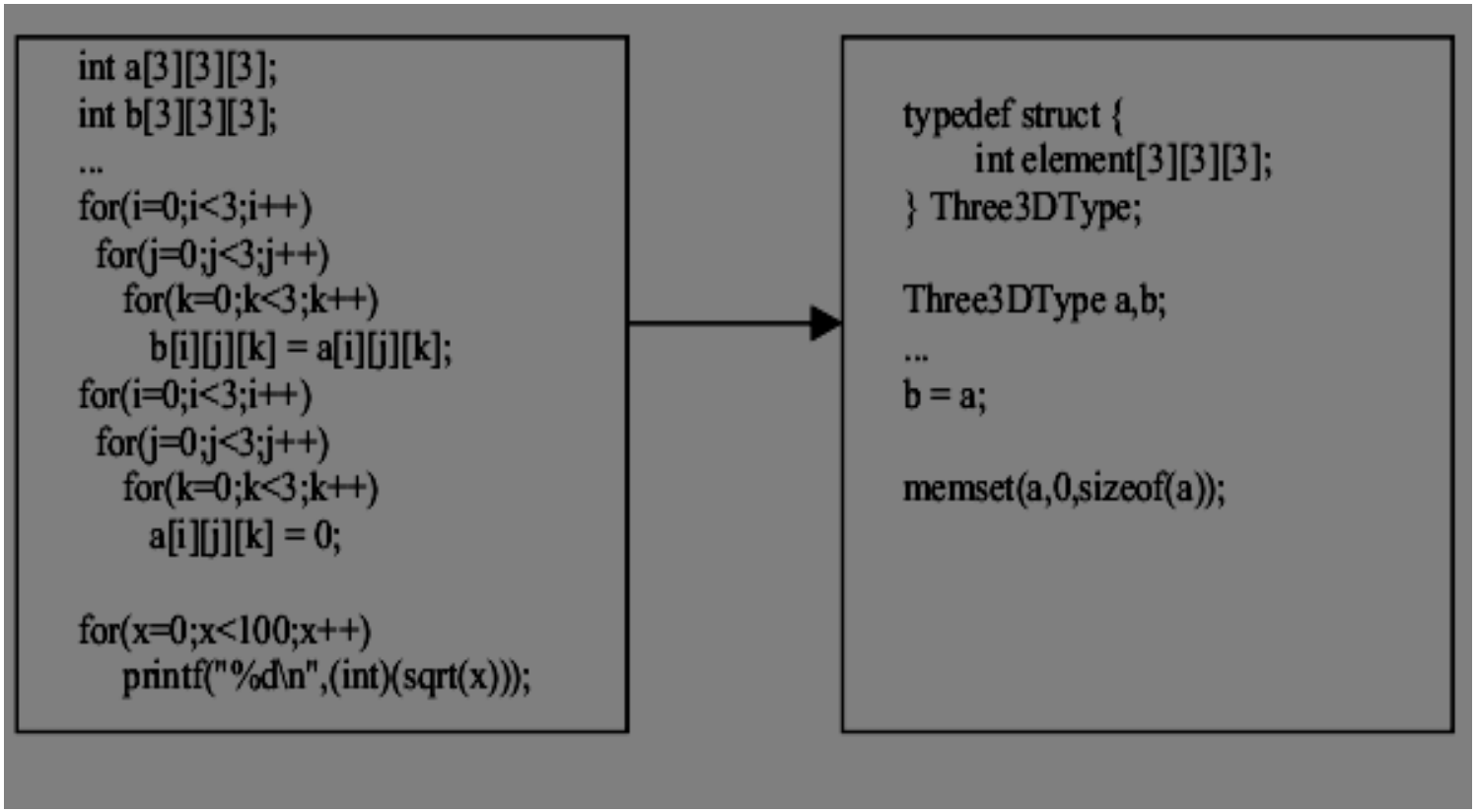


Return Value

- ✓ The return value of a function will be stored in a register.
- ✓ If this return data has no intended usage, time and space are wasted in storing this information.
- ✓ Programmer should define the function as “void” to minimize the extra handling in the function.

Array & Structure Initialization

A simple illustration of implementation:



```
int a[3][3][3];
int b[3][3][3];
...
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    for(k=0;k<3;k++)
      b[i][j][k] = a[i][j][k];
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    for(k=0;k<3;k++)
      a[i][j][k] = 0;

for(x=0;x<100;x++)
    printf("%d\n",(int)(sqrt(x)));
```

```
typedef struct {
      int element[3][3][3];
} Three3DType;

Three3DType a,b;
...
b = a;

memset(a,0,sizeof(a));
```

Inline Function

- ✓ The technique will cause the compiler to replace all calls to the function, with a copy of the function's code.
- ✓ This will eliminate the runtime overhead associated with the function call.
- ✓ This is most effective if the function is called frequently, but contains only a few lines of code.

Inline function

```
#pragma inline (sum)
...
int sum(int a, int b)
{           return (a+b);
}
...
routine()
{           ...
            total  = sum (x,y);
            ...
            sub_total = sum (cost_a, cost_b)
            ...
}
```

SWITCH & and Non-contiguous Case Expressions

- ✓ Use if-else statements in place of switch statements that have noncontiguous case expressions.
- ✓ If the case expressions are contiguous or nearly contiguous integer values, most compilers translate the switch statement as a jump table instead of a comparison chain.

Switch contd...

Jump tables generally improve performance because:

- ✓ They reduce the number of branches to a single procedure call.
- ✓ The size of the control-flow code is the same no matter how many cases there are.
- ✓ The amount of control-flow code that the processor must execute is the same for all values of the switch expression.

Switch contd...

- ✓ However, if the case expressions are noncontiguous values, most compilers translate the switch statement as a comparison chain.
- ✓ Comparison chains are undesirable because:
  - They use dense sequences of conditional branches, which interfere with the processor's ability to successfully perform branch prediction.
  - The size of the control-flow code increases with the number of cases.
  - The amount of control-flow code that the processor must execute varies with the value of the switch expression.

Switch – Example 1

A switch statement like this one, provides good performance:

```
switch (grade)
{
    case 'A':
        ...
        break;
    case 'B':
        ...
        break;
    case 'C':
        ...
        break;
    case 'D':
        ...
        break;
    case 'F':
        ...
        break;
}
```

Switch – Example 2

Because the case expressions in the following switch statement are not contiguous values, the compiler will likely translate the code into a comparison chain instead of a jump table:

```
switch (a)
{
    case 8:
        // Sequence for a==8
        break;
    case 16:
        // Sequence for a==16
        break;
    ...
    default:
        // Default sequence
        break;
}
```

If-Else statements

To avoid a comparison chain and its undesirable effects on branch prediction, replace the switch statement with a series of if-else statements, as follows:

```
if (a==8) {
   // Sequence for a==8
}
else if (a==16) {
   // Sequence for a==16
}
...
else {
   // Default sequence
}
```

Arranging Cases by Probability of Occurrence

- ✓ Arrange switch statement cases by probability of occurrence, from most probable to least probable.
- ✓ Avoid switch statements such as the following, in which the cases are not arranged by probability of occurrence:

```
int days_in_month, short_months, normal_months, long_months;

switch (days_in_month) {
  case 28:
  case 29: short_months++; break;
  case 30: normal_months++; break;
  case 31: long_months++; break;
  default: printf("Month has fewer than 28 or more than 31 days.\n");
}
```

Switch case - Contd...

- ✓ Instead, arrange the cases to test for frequently occurring values first:

```
switch (days_in_month) {
  case 31: long_months++; break;
  case 30: normal_months++; break;
  case 28:
  case 29: short_months++; break;
  default: printf("Month has fewer than 28 or more than 31 days.\n");
}
```

Use of Function Prototypes

- ✓ In general, use prototypes for all functions.
- ✓ Prototypes can convey additional information to the compiler that might enable more aggressive optimizations.

Generic Loop Hoisting

- ✓ To improve the performance of inner loops, reduce redundant constant calculations
- ✓ Avoid:

```
for (i...) {
   if (CONSTANT0) {
      DoWork0(i);   // Does not affect CONSTANT0.
   }
   else {
      DoWork1(i);   // Does not affect CONSTANT0.
   }
}
```

Generic Loop Hoisting

- ✓ Preferred optimization

```
if (CONSTANT0) {
    for (i...) {
        DoWork0(i);
    }
}
else {
    for (i...) {
        DoWork1(i);
    }
}
```

Contd...

- ✓ Un-optimized code

```
for (i...) {
   if (CONSTANT0) {
      DoWork0(i);    // Does not affect CONSTANT0 or CONSTANT1.
   }
   else {
      DoWork1(i);    // Does not affect CONSTANT0 or CONSTANT1.
   }

   if (CONSTANT1) {
      DoWork2(i);    // Does not affect CONSTANT0 or CONSTANT1.
   }
   else {
      DoWork3(i);    // Does not affect CONSTANT0 or CONSTANT1.
   }
}
```

Contd...

```
#define combine(c1, c2) (((c1) << 1) + (c2))
switch (combine(CONSTANT0 != 0, CONSTANT1 != 0)) {
   case combine(0, 0):
      for(i...) {
         DoWork0(i);
         DoWork2(i);
      }
      break;
   case combine(1, 0):
      for(i...) {
         DoWork1(i);
         DoWork2(i);
      }
      break;
   case combine(0, 1):
      for(i...) {
         DoWork0(i);
         DoWork3(i);
      }
      break;
   case combine( 1, 1 ):
      for(i...) {
         DoWork1(i);
         DoWork3(i);
      }
      break;
   default:
      break;
}
```

Local Static Functions

- ✓ Declare as static functions that are not used outside the file where they are defined.
- ✓ Declaring a function as static forces internal linkage.
- ✓ Functions that are not declared as static default to external linkage, which may inhibit certain optimizations—for example, aggressive in-lining—with some compilers.

Explicit Parallelism in Code

- ✓ Where possible, break long dependency chains into several independent dependency chains that can then be executed in parallel, exploiting the execution units in each pipeline.
- ✓ Avoid:

```
double a[100], sum;
int i;

sum = 0.0f;
for (i = 0; i < 100; i++) {
    sum += a[i];
}
```

Contd...

```
double a[100], sum1, sum2, sum3, sum4, sum;
int i;

sum1 = 0.0;
sum2 = 0.0;
sum3 = 0.0;
sum4 = 0.0;
for (i = 0; i < 100; i + 4) {
    sum1 += a[i];
    sum2 += a[i+1];
    sum3 += a[i+2];
    sum4 += a[i+3];
}
sum = (sum4 + sum3) + (sum1 + sum2);
```

Notice that the four-way unrolling is chosen to exploit the four-stage fully pipelined floating-point adder. Each stage of the floating-point adder is occupied on every clock cycle, ensuring maximum sustained utilization.

Extracting Common Sub expressions

- ✓ Manually extract common sub expressions where C compilers may be unable to extract them from floating-point expressions due to the guarantee against reordering of such expressions in the ANSI standard.

Contd...

Manually optimize – Example 1

```
double a, b, c, d, e, f;


e = b * c / d;
f = b / d * a;
```

```
double a, b, c, d, e, f, t;


t = b / d;
e = c * t;
f = a * t;
```

Contd...

Manually optimize – Example 2

```
double a, b, c, e, f;


e = a / c;
f = b / c;
```

```
double a, b, c, e, f, t;


t = 1 / c;
e = a * t
f = b * t;
```

Replacing Integer Division with Multiplication

- ✓ Replace integer division with multiplication when there are multiple divisions in an expression.
- ✓ This is possible only if no overflow will occur during the computation of the product. The possibility of an overflow can be determined by considering the possible ranges of the divisors.
- ✓ Integer division is the slowest of all integer arithmetic operations.

Contd...

Example - Replace division by multiplication

Avoid code that uses two integer divisions:

```
int i, j, k, m;

m = i / j / k;
```

Instead, replace one of the integer divisions with the appropriate multiplication:

```
m = i / (j * k);
```

Sign of Integer Operands

- ✓ Where there is a choice of using either a signed or an unsigned type, take into consideration that some operations are faster with unsigned types while others are faster for signed types.

Contd…

```
double x;           ====>   mov [temp+4], 0
unsigned int i;             mov eax, i
                            mov [temp], eax
x = i;                      fild QWORD PTR [temp]
                            fstp QWORD PTR [x]
```

```
double x;   ====>   fild DWORD PTR [i]
int i;              fstp QWORD PTR [x]


x = i;
```

Contd...

Signed vs. Unsigned int division

```
int i;          ====>   mov eax, i
                        cdq
i = i / 4;              and edx, 3
                        add eax, edx
                        sar eax, 2
                        mov i, eax
```

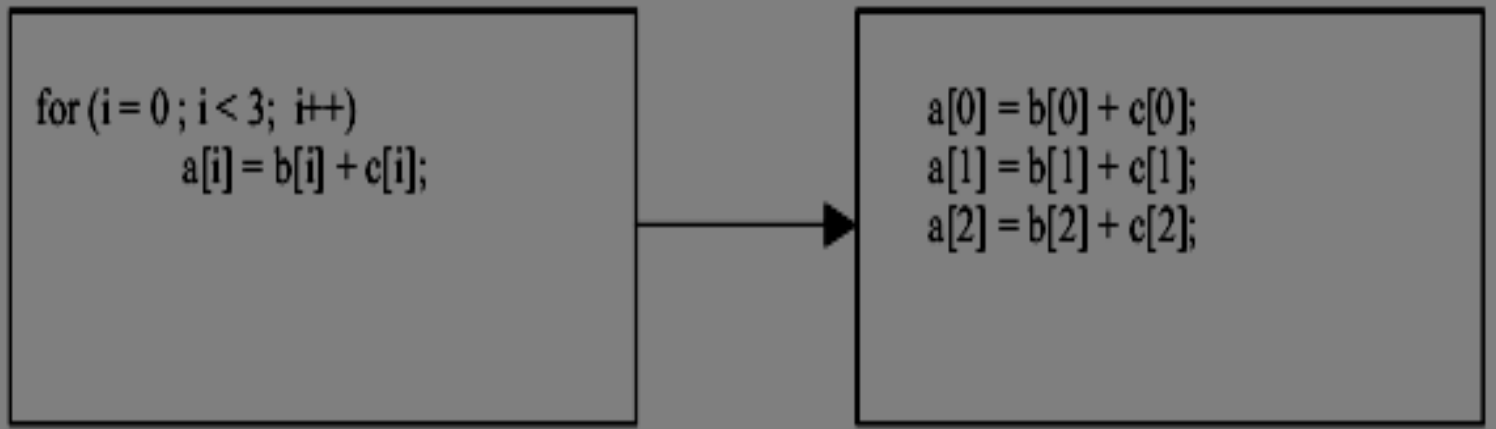
```
unsigned int i;   ====>   shr  i, 2


i = i / 4;
```

Contd...

- ✓ In summary, use unsigned types for:
  - Division and remainders
  - Loop counters
  - Array indexing
- ✓ Use signed types for:
  - Integer-to-floating-point conversion

Loop Unrolling

If the loop is small, overhead will be higher.



Contd…

If the loop is larger, the overhead can also be reduced by:

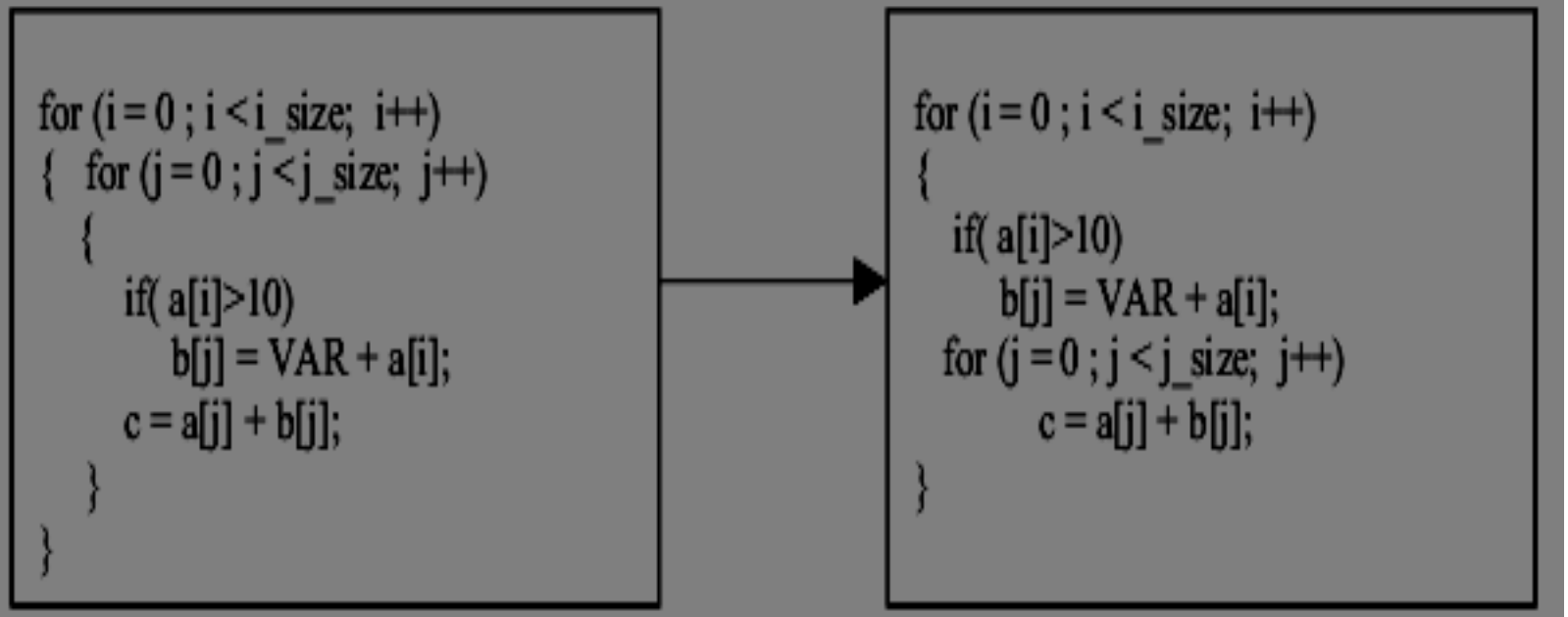
```
for (i = 0 ; i < 3*n;  i++)
        a[i] = b[i] + c[i];
```

```
for (i = 0 ; i < 3*n;  i +=3)
{   a[i+0] = b[i+0] + c[i+0];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
}
```

Loop Invariant
IF Code Floating

Incorrect

'IF' statements that do not change from iteration to iteration may be moved out of the loop

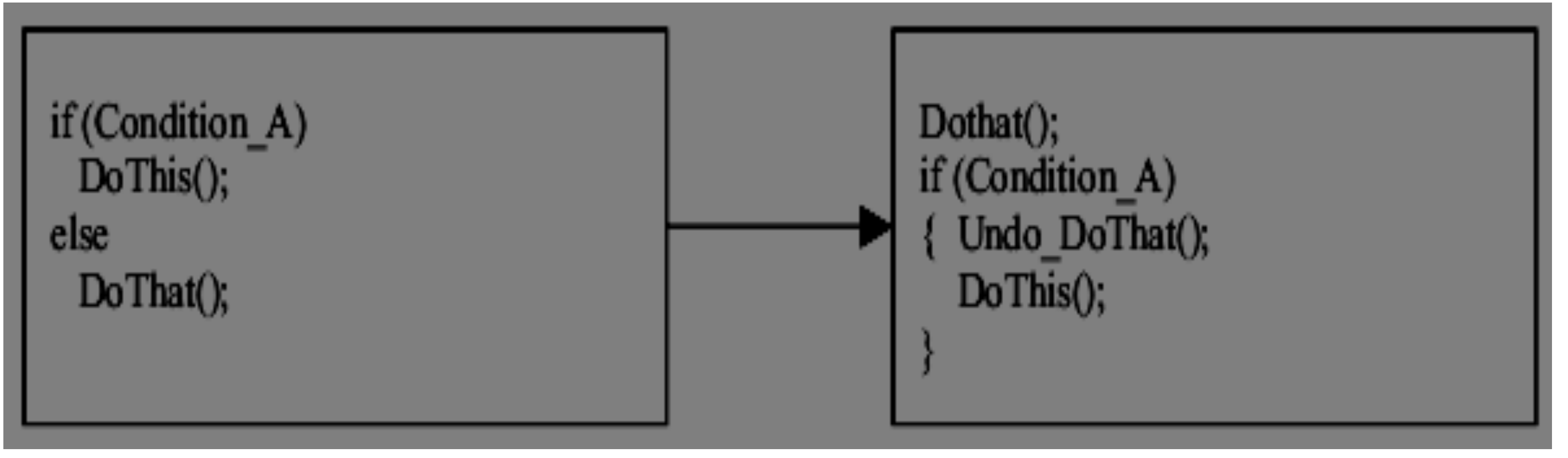


```
for (i = 0 ; i < i_size;  i++)
{   for (j = 0 ; j < j_size;  j++)
    {
        if( a[i]>10)
            b[j] = VAR + a[i];
        c = a[j] + b[j];
    }
}
```

```
for (i = 0 ; i < i_size;  i++)
{
    if( a[i]>10)
        b[j] = VAR + a[i];
   for (j = 0 ; j < j_size;  j++)
            c = a[j] + b[j];
}
```

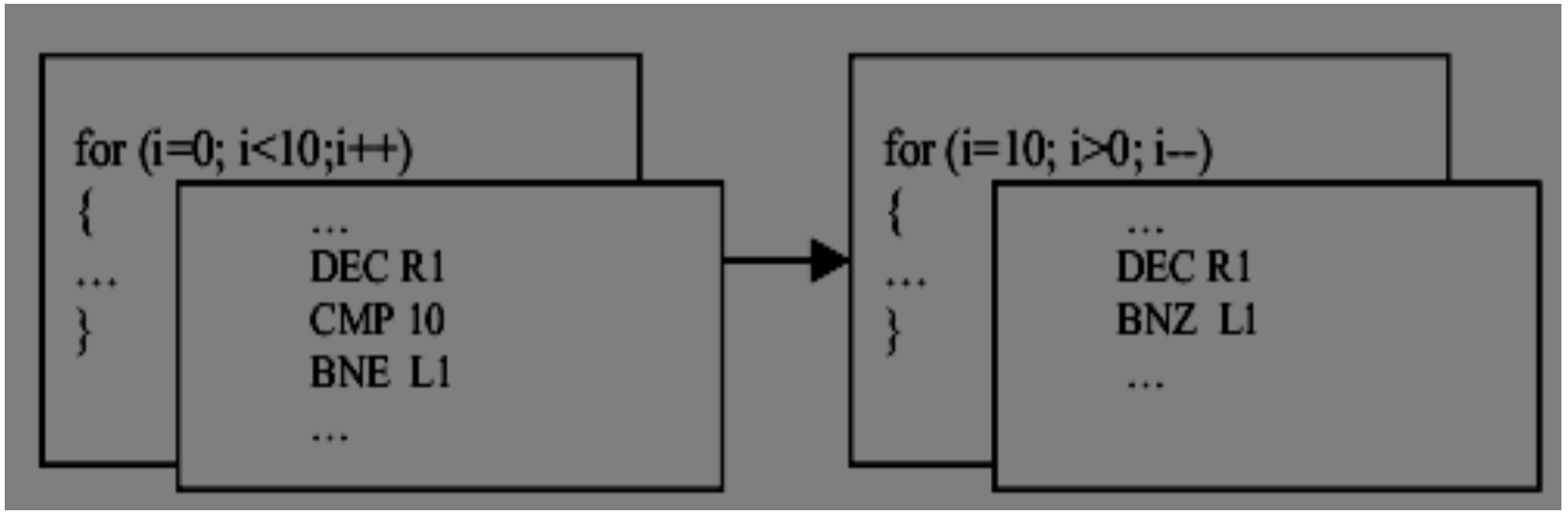
Else Clause Removal

- ✓ A jump condition is inevitable in the first case, whereas the second case gives the higher possibility process (Condition_A) to have a sequential execution. This technique is possible only if Undo "DoThat()" process is possible.



Loop Overhead

- ✓ The MCU have a conditional branch mechanism that works well when counting down from positive number to zero.
- ✓ It is easier to detect “Zero” (Branch non zero – single instruction) than a “predefined number” (Compare and perform jump – two instructions)



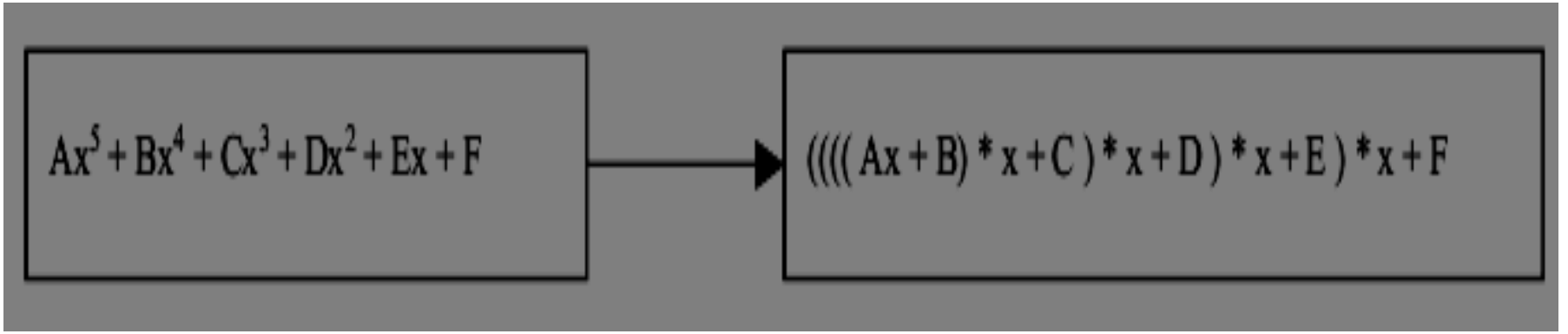
Specify Optimization Type for Each Module

#pragma option can be used to limit and control the optimization regions.

```
#pragma option speed                    // From this point, code will be optimized based on Speed
void function_A(void)
{
...
}
#pragma option size                     // From this point, code will be optimized based on Size
void function_A(void)
{
...
}
```

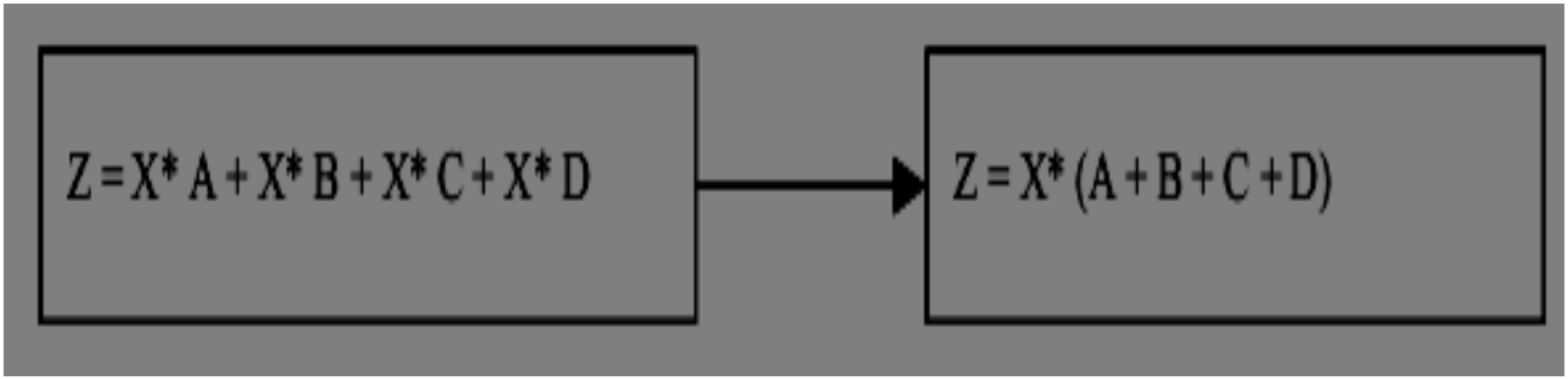
Horner's Rule of Polynomial Evaluation

The rules state that a polynomial can be rewritten as a nested factorization. The reduced arithmetic operations will have better efficiency and execution speed.

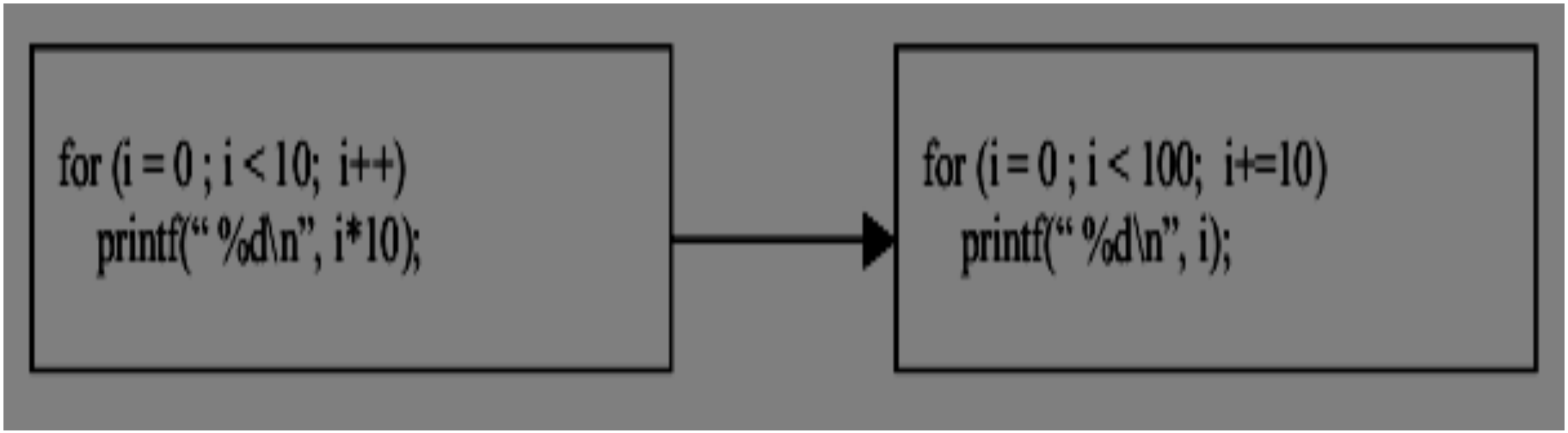


Factorization

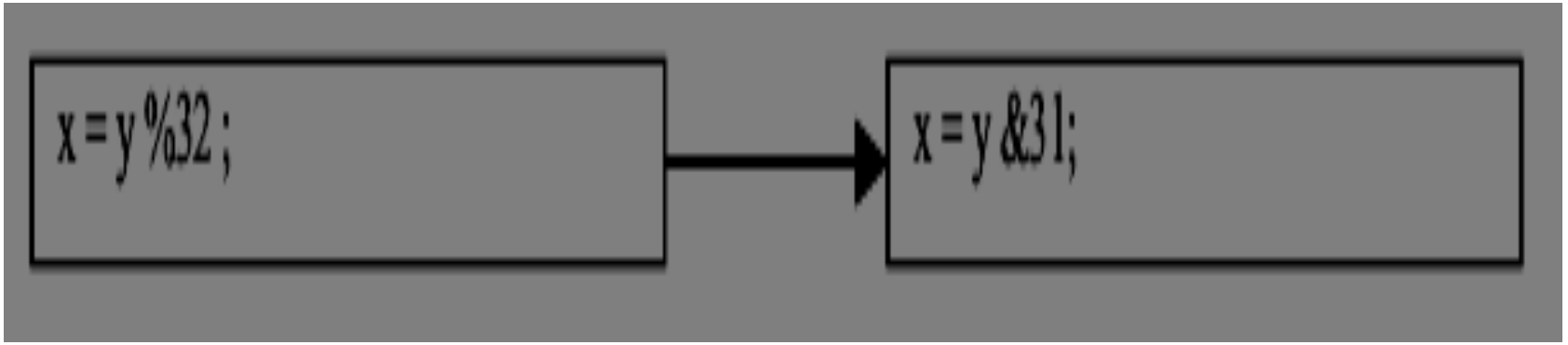
The compiler may be able to perform better when the formula



Use Finite Differences to Avoid Multiplies

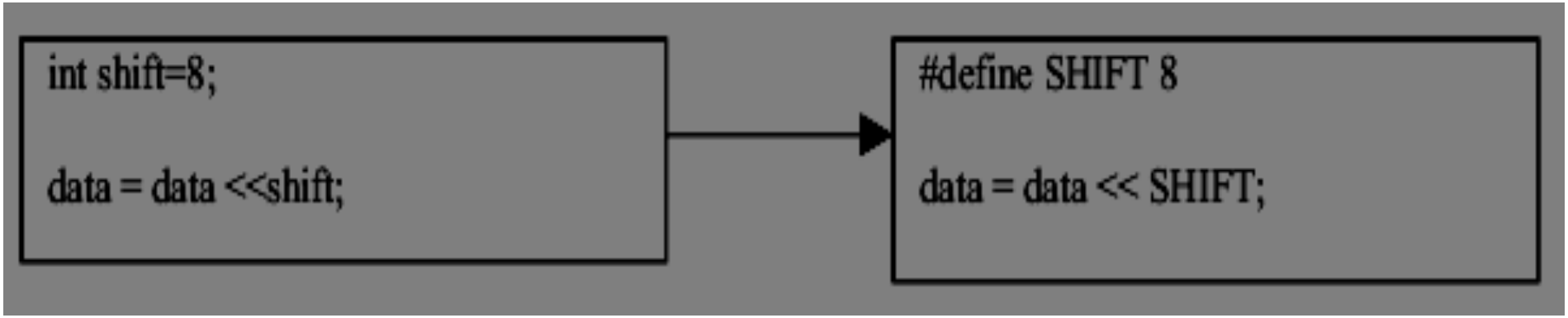


Modula



Constant in Shift Operations

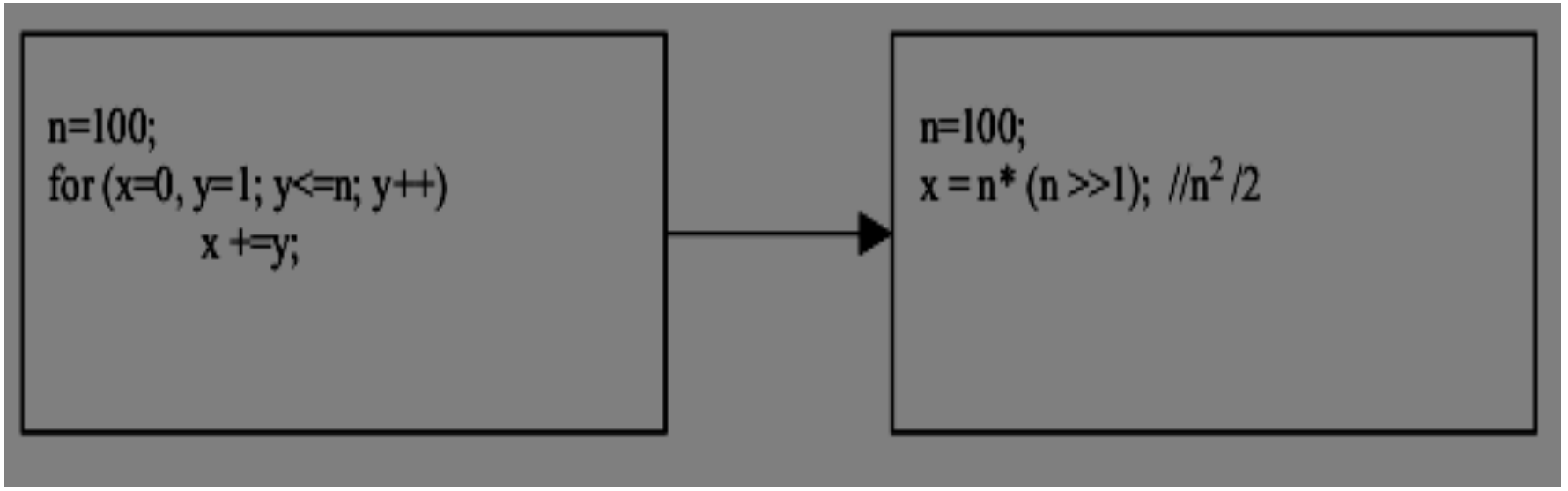
- ✓ For shift operations, if the shift count is a variable, the compiler calls a runtime routine to process the operation.
- ✓ If the shift count is a constant, the compiler does not call the routine, which give significant speed improvement.



Use Formula

- ✓ Use formula instead of loops
- ✓ What is the formula for sum of naturals numbers?

Contd…



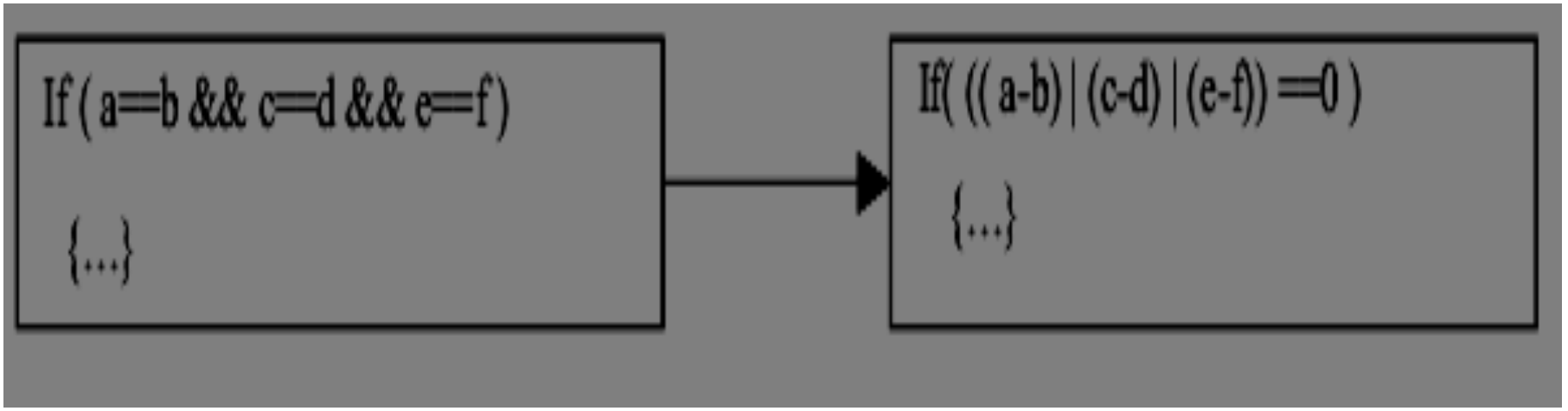
Simplify Condition

Un-optimized – Example 1

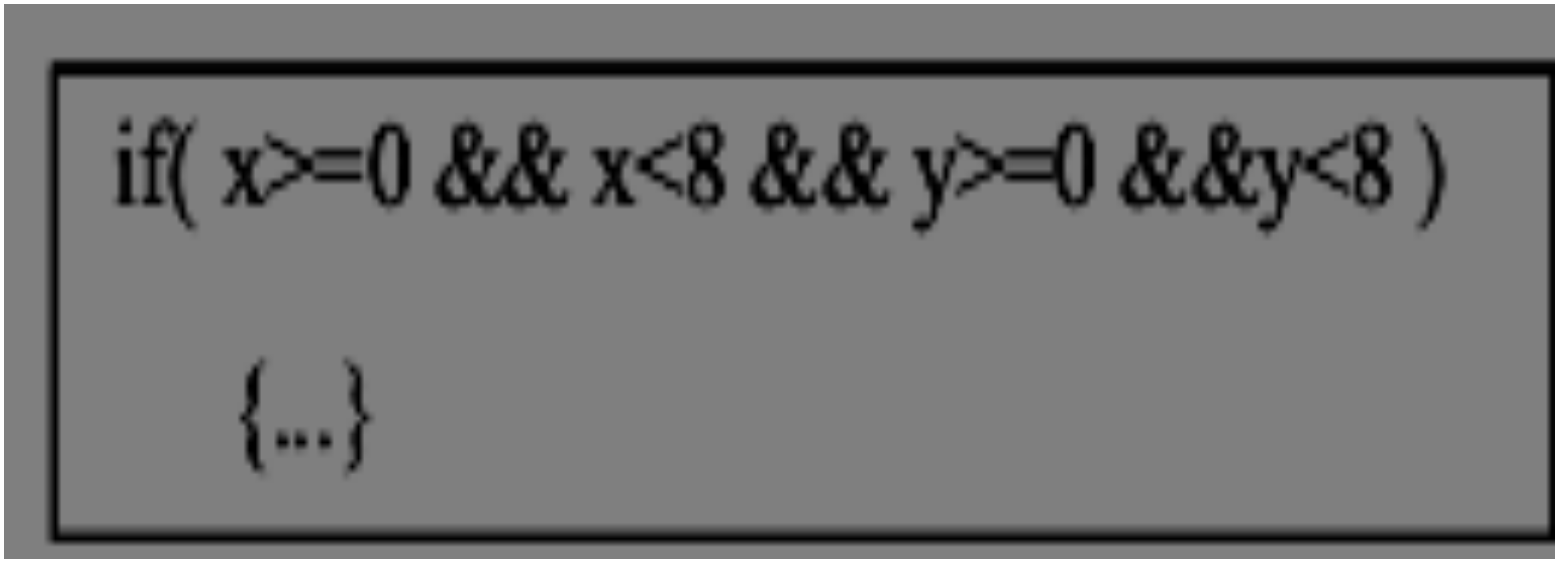
```
If ( a==b && c==d && e==f )

  {...}
```

Contd..

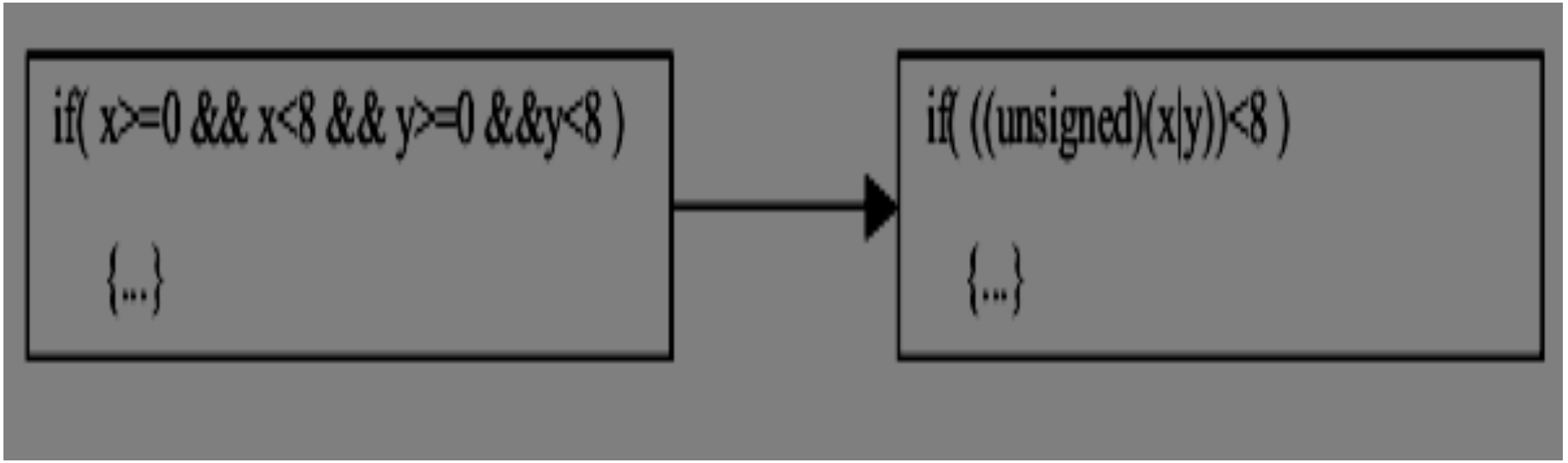
Optimized



Contd…



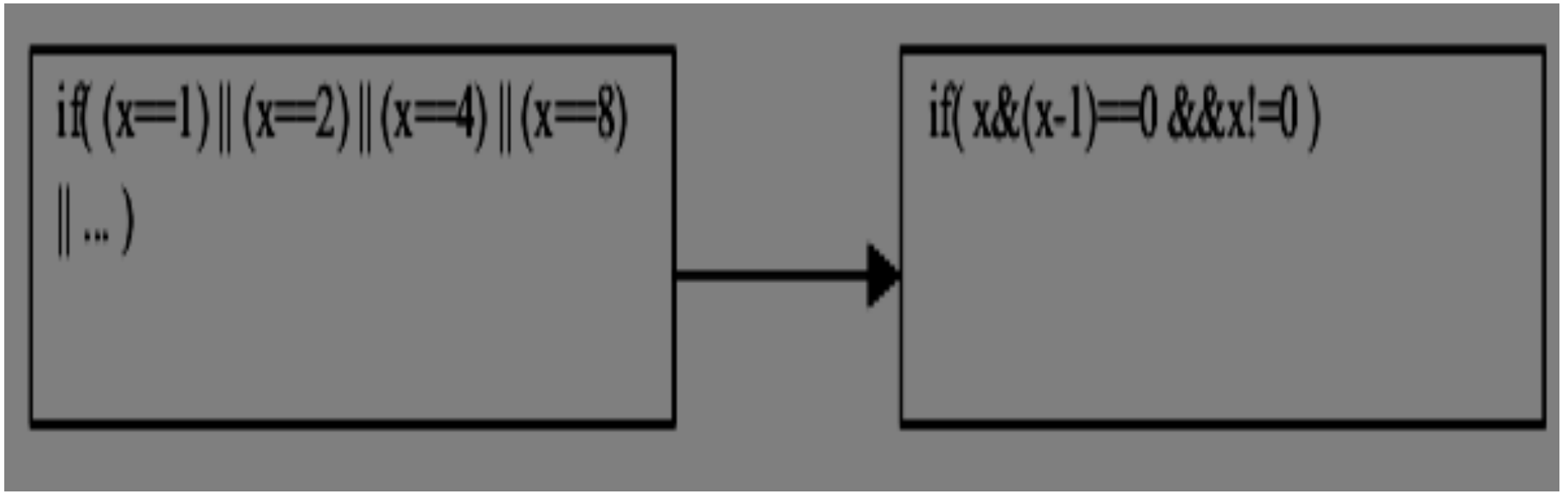
Contd…



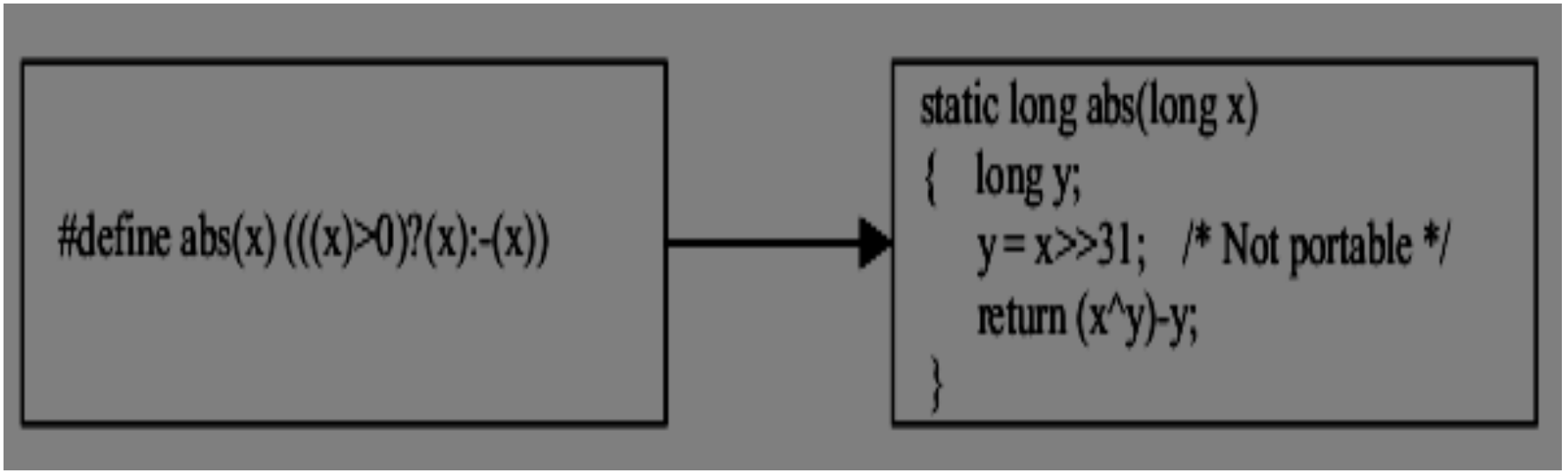
Contd...

```
if( (x==1) || (x==2) || (x==4) || (x==8)
|| ... )
```

Contd…



Absolute value



Profiling

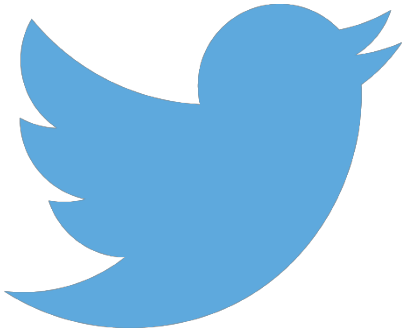
- ✓ What is profiling
- ✓ Memory profiling
- ✓ Leaks
- ✓ Memory usage
- ✓ CPU Usage
- ✓ Cache

Stay connected

About us: Emertxe is India's one of the top IT finishing schools & self learning kits provider. Our primary focus is on Embedded with diversification focus on Java, Oracle and Android areas

Emertxe Information Technologies,
No-1, 9th Cross, 5th Main,
Jayamahal Extension,
Bangalore, Karnataka 560046
T: +91 80 6562 9666
E: training@emertxe.com

https://www.facebook.com/Emertxe



https://twitter.com/EmertxeTweet

https://www.slideshare.net/EmertxeSlides

EMERTXE

THANK YOU