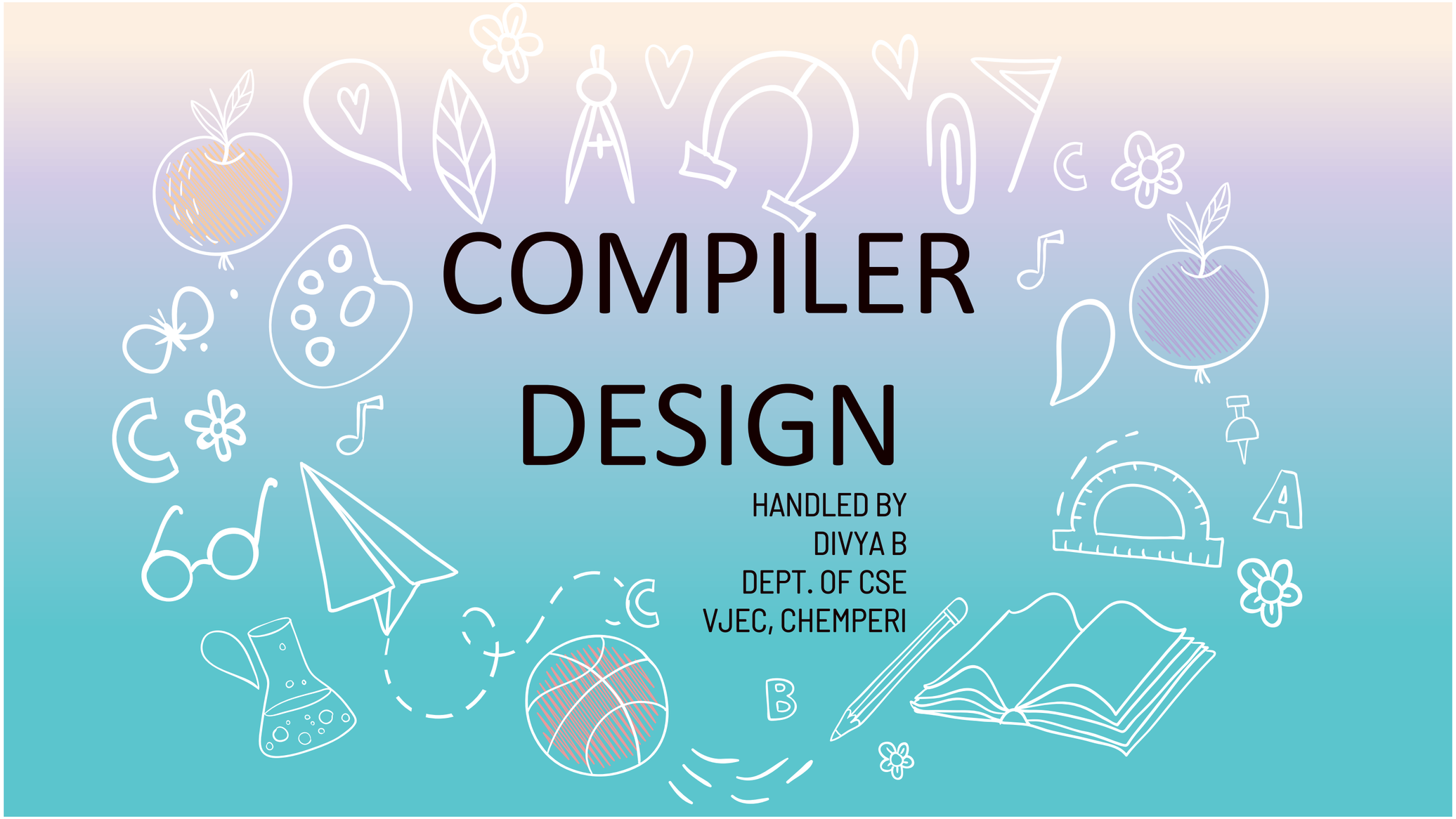


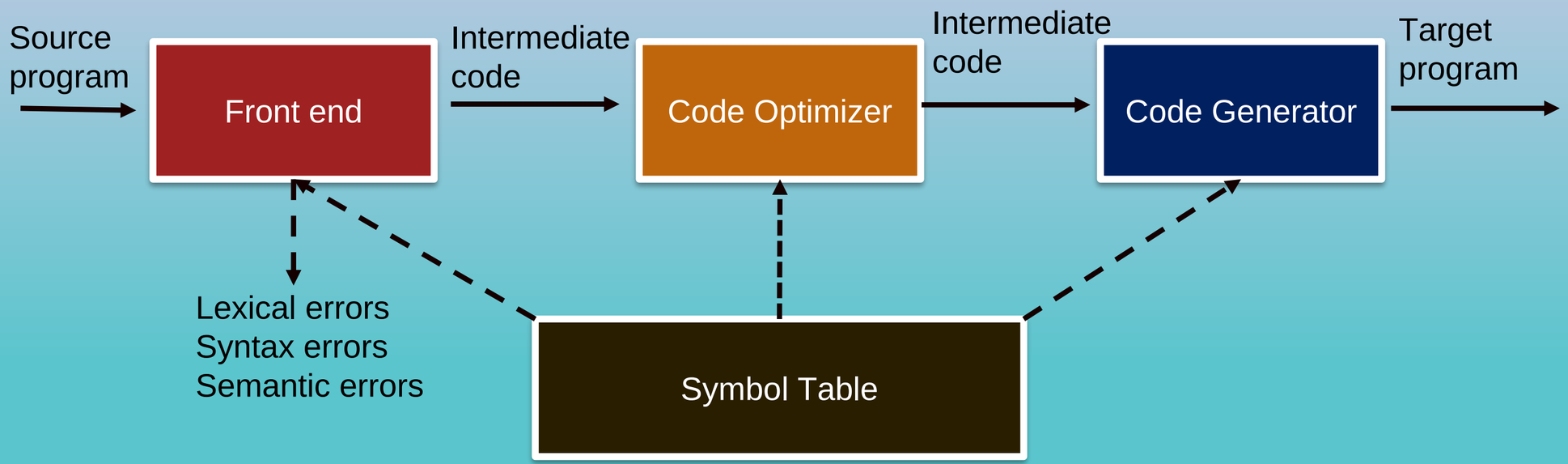
# COMPILER DESIGN

HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI

# CODE GENERATION

- The final phase in the compiler model is the code generator.
- It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

# CODE GENERATION



# CODE GENERATION

- The following issues arise in the code generation phase
  1. Input to the code generator
  2. Target program
  3. Memory management
  4. Instruction selection
  5. Register allocation
  6. Evaluation order
  7. Approaches to code generation

# CODE GENERATION

- **Input to the code generator**
  - ✓ The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the intermediate representation.

# CODE GENERATION

- **Input to the code generator**
  - ✓ There are several choices for the intermediate language including postfix notation, three address representation such as quadruple, virtual machine representations such as stack machine code, and graphical representations such as syntax trees and DAGs.

# CODE GENERATION

- **Input to the code generator**
  - ✓ Prior to the code generation the front end scanned, parsed and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language, type checking has taken place, so type conversion operators have been inserted wherever necessary.
  - ✓ The code generation phase can therefore proceed on the assumption that its input is free of errors.

# CODE GENERATION

- **Target Programs**
  - ✓ The output of the code generator is the target program.
  - ✓ This output may take on a variety of forms- absolute machine language, relocatable machine language or assembly language.
  - ✓ Producing an absolute machine language as output has the advantage that it can be placed in a fixed location in memory and immediately executed.

# CODE GENERATION

- **Target Programs**
  - ✓ Producing a relocatable machine language program as output allows subprograms to be compiled separately.
  - ✓ A set of relocatable object modules can be linked together and loaded for execution by a linking loader.
  - ✓ Producing an assembly language program as output makes the process of code generation somewhat easier.
  - ✓ We can generate symbolic instructions and use the macro facilities of the assembler to help generate code.

# CODE GENERATION

- **Target Programs**
  - ✓ The instruction set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high quality machine code.
  - ✓ The most common target machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer) and stack based.

# CODE GENERATION

- **Target Programs**
  - ✓ The RISC machine has many registers, three-address instructions, simple addressing modes and a relatively simple instruction set architecture.
  - ✓ In contrast, a CISC machine has few registers, two-address instructions, a variety of addressing modes, several register classes, variable length instructions and instructions with side effects.

# CODE GENERATION

- **Target Programs**
  - ✓ In a stack based machine, operations are done by pushing operands onto the stack and then performing the operations on the operands at the top of the stack.
  - ✓ To achieve high performance, the top of the stack is typically kept in registers.

# CODE GENERATION

- **Target Programs**
  - ✓ Stack based architectures were revived with the introduction of Java Virtual Machine (JVM).
  - ✓ The JVM is a software interpreter for Java bytecodes, an intermediate language produced by Java compiler.
  - ✓ The interpreter provides software compatibility across multiple platforms.
  - ✓ To overcome the high-performance penalty of interpretation, which can be on the order of a factor of 10, Just-In-Time java compiler.

# CODE GENERATION

- **Memory Management**
  - ✓ Mapping of variable names to address is done cooperatively by the front end and code generator.
  - ✓ Name and width are obtained from symbol table.
  - ✓ Width is the amount of storage needed for that variable.
  - ✓ Each three-address code is translated to addresses and instructions during code generation.

# CODE GENERATION

- **Memory Management**
  - ✓ A relative addressing is done for each instruction.
  - ✓ All the labels should be addressed properly.
  - ✓ Backward jump is easier to manage than the forward jump.

# CODE GENERATION

- **Instruction Selection**
  - ✓ The code generator must map the IR (Intermediate Representation) program into a code sequence that can be executed by the target machine.
  - ✓ If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates.
  - ✓ Such statement-by-statement code generation often produces poor code that needs further optimization.

# CODE GENERATION

- **Instruction Selection**
  - ✓ If IR reflects some of the low level details of the underlying machine, then the code generator can use this information to generate more efficient code sequence.
  - ✓ The **nature** of the instruction set of the target machine has a strong effect on the difficulty of instruction selection.
  - ✓ **Uniformity** and **completeness** of the instruction set are important factors.

# CODE GENERATION

- **Instruction Selection**
  - ✓ If the target program does not support each data type in a uniform manner, then each exception to the general rule requires special handling.
  - ✓ E.g. In some machines floating point operations are done using separate registers.
  - ✓ **Instruction speed** and **machine idioms** are other important factors.

# CODE GENERATION

- **Instruction Selection**
  - ✓ If we do not care about the efficiency of the target program, instruction selection is straightforward.
  - ✓ For each common three-address statement, a general code can be designed.

# CODE GENERATION

- **Instruction Selection**

```
x = y + z
MOV y, R0
ADD z, R0
MOV R0, x
```

```
a = b + c
d = a + e
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0   <-- can be avoided.
ADD e, R0
MOV R0, d
```

# CODE GENERATION

- **Instruction Selection**
  - ✓ The **quality** of the generated code is usually determined by its speed and size.
  - ✓ On most machines, a given IR program can be implemented by many different code sequence, with significant cost difference between the different implementations.

# CODE GENERATION

- **Instruction Selection**
  - ✓ E.g. if the target machine has an increment instruction INC, then the three-address statement a = a+1 may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into a register, adds one to the register, and then store the result back into a.

# CODE GENERATION

- **Instruction Selection**

**a = a+1**

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

```
INC a
```

# CODE GENERATION

- **Register Allocation**
  - ✓ A key problem in code generation is deciding what values to hold in what registers.
  - ✓ Registers are the fastest computational unit on the target machine, but we usually not have enough of then to hold all values.

# CODE GENERATION

- **Register Allocation**
  - ✓ The use of registers is often subdivided into two sub problems.
  - ✓ **Register allocation**, during which we select the set of variables that will reside in registers at each point in the program.
  - ✓ **Register assignment**, during which we pick the specific register that a variable will reside in.

# CODE GENERATION

- **Register Allocation**
  - ✓ Finding on optimal assignment of registers to variables is difficult, even with single-register machines and it is an NP-complete problem.
  - ✓ This problem becomes more complicated, if the target machine has certain conventions on register use.
  - ✓ E.g. in 8085, one of the operand of some operations should be placed in register A.

# CODE GENERATION

- **Choice of evaluation order**
  - ✓ The order of evaluation can affect the efficiency of target code. Some order requires fewer registers and instructions than others.
  - ✓ Picking the best order is an NP-complete problem. This can be solved up to an extend by code optimization in which the order of instruction may change.

# CODE GENERATION

- **Approaches to code generation**
  - ✓ The target code generated should be correct.
  - ✓ Correctness depends on the number of special cases the code generator might face.
  - ✓ Other design goals of code generator are, it should be easily implemented, tested and maintained.

# TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- Our target computer is a byte-addressable machine with four bytes to a word and n general purpose registers, R0, R1, R2…. Rn-1.
- It has two address instructions of the form

  ***op source, destination***

  in which op is an opcode and source and destination are data fields.

# TARGET MACHINE

- It has the following opcodes
  - ✓ MOV (move source to destination)
  - ✓ ADD (add source to destination)
  - ✓ SUB (subtract source from destination)
  - .

# TARGET MACHINE

- The source and destination fields are not long enough to hold memory addresses, so certain bit patterns in these fields specify that words following an instruction contain operands and/or addresses.
- The source and destination of an instruction are specified by combining registers and memory locations with address mode.
- contents(a) denotes the contents of the register or memory address represented by a.

.

# TARGET MACHINE

- The address modes together with their assembly-language forms and associated costs are

| MODE | FORM | ADDRESS | ADDED COST |
|---|---|---|---|
| Absolute | M | M | 1 |
| Register | R | R | 0 |
| Indexed | c(R) | c+contents(R) | 1 |
| Indirect register | *R | contents(R) | 0 |
| Indirect indexed | *c(R) | contents(c+contents(R)) | 1 |

# TARGET MACHINE

- **MOV R0, M**
  - Stores the contents of register R0 into memory location M.
- **MOV 4(R0), M**
  - Stores the value contents(4 + contents(R0))
- **MOV *4(R0), M**
  - Stores the value contents(contents(4 + contents(R0))) .

# TARGET MACHINE

| MODE | FORM | ADDRESS | ADDED COST |
|---|---|---|---|
| Immediate or literal | #C | C | 1 |

- MOV #1, R0
  - Load constant 1 into register R0.

# TARGET MACHINE

- **Instruction Cost**
  - ✓ Cost of an instruction is one plus the costs associated with the source and destination address modes, indicated by add cost in the above table.
  - ✓ This cost corresponds to the length of the instruction.
  - ✓ Address modes involving registers have cost zero, while those with a memory location or literal in them have cost one, because such operands have to be stored with the instruction.

# TARGET MACHINE

- **Instruction Cost**
  - ✓ Minimizing the instruction length will tend to minimize the time taken to perform the instruction as well.
    1. The instruction MOV R0, R1 copies the contents of register R0 into register R1. This instruction has cost one, since it occupies only one word of memory.

# TARGET MACHINE

- **Instruction Cost**
  - ✓ Minimizing the instruction length will tend to minimize the time taken to perform the instruction as well.
    2. The (store) instruction MOV R5 , M copies the contents of register R5 into memory location M. This instruction has cost two, since the address of memory location M is in the word following the instruction.

# TARGET MACHINE

- **Instruction Cost**
  - ✓ Minimizing the instruction length will tend to minimize the time taken to perform the instruction as well.
    3. The instruction ADD #1 , R3 adds the constant 1 to the contents of register 3, and has cost two, since the constant 1 must appear in the next word following the instruction.

# TARGET MACHINE

- **Instruction Cost**
  - ✓ Minimizing the instruction length will tend to minimize the time taken to perform the instruction as well.

4. The instruction SUB 4 (R0) , *12 (R1) stores the value ***contents (contents (12+ contents (R1))) - contents (contents (4 +R0))*** into the destination *12 (R1) .

The cost of this instruction is three, since the constants 4 and 12 are stored in the next two words following the instruction.

# TARGET MACHINE

- **Instruction Cost**

```
MOV b, R0
ADD c, R0
MOV R0, a
```

**cost = 6**

# TARGET MACHINE

- **Instruction Cost**

```
MOV b, a
ADD c, a
```

**cost = 6**

Assuming R0, R 1 , and R2 contain the addresses of a, b, and c respectively,

```
MOV *R1, *R0
ADD *R2, *R0
```

**cost = 2**

# TARGET MACHINE

- **Instruction Cost**

  Assuming R1 and R2 contain the values of b and c, respectively, and that the value of b is not needed after the assignment,

  ADD R2, R1
  MOV R1, a

  **cost = 3**

# SIMPLE CODE GENERATOR

- The code generation strategy is the generation of target code for a sequence of three-address statement.
- It is assumed that computed result is in registers as long as possible, storing them only
  a) if the register is needed for another computation

  or

  b) just before a procedure call, jump or labelled statement.

# SIMPLE CODE GENERATOR

- For a three-address statement a = b + c, generate instruction ADD Rj, Ri with cost one, leaving the result a in register Ri.
- This sequence is possible only if register Ri contains b, Rj contains c and b is not live after the statement; that is, b is not used after the statement.

# SIMPLE CODE GENERATOR

- If Ri contains b and c is in a memory location,

ADD c, Ri **cost =2**

Or

MOV c, Rj
ADD Rj, Ri **cost =3**

# SIMPLE CODE GENERATOR

- **Register and Address Descriptors**
  - ✓ The code generation algorithm uses descriptors to keep track of register contents and addresses for names.
    **A Register Descriptor -** keeps track of what is currently in each register. It is consulted whenever a new register is needed.

# SIMPLE CODE GENERATOR

- **Register and Address Descriptors**
  **Address Descriptor -** keeps track of the location where the current value of the name can be found at run time. The location might be a register, a stack location or a memory address. This information can be stored in the symbol table and is used to determine the accessing method for a name.

# A CODE GENERATION ALGORITHM

- Code generation algorithm takes input as a sequence of three-address statements constituting a basic block. Statement of the form x = y op z performs the following actions.

**Step I:** `x = y op z`

1. Let $L = getreg()$.
2. Let $y' = location(y)$ (preferably register). If $y' \neq L$, generate

   `MOV y' L`

3. Let $z' = location(z)$ (as above). Generate

   `OP z' L`

4. Update address descriptor of `x` to $\{L\}$; remove `x` from all RDs.
5. If $L$ is a register, update its RD.
6. If `y` (or `z`) is
   (i) in a register
   (ii) has no next use and is not live on exit from the block
   change RD to indicate that the register no longer contains `y` (or `z`).

**Step I (special case):** `x = y`

1. If `y` is in register $R_i$:
   (i) change RDs and AD for `x` to indicate that `x` is now only in $R_i$;
   (ii) if `y` has no next use and is not live on exit from block, delete `y` from RD for $R_i$.
2. If `y` is in memory:
   (i) load `y` into a register (obtained using getreg()), and proceed as above; OR
   (ii) generate `MOV y x`
   (preferable if `x` has no next use in the block).

**Step II:** after processing all stmts in the basic block, generate `MOV` instructions to store all variables that are live on exit, but not currently in their memory locations.

for each variable `x` in each register
    check AD for `x` to determine whether its current value is in memory
    if not, generate suitable `MOV` instruction

## The Function *getreg* M

The function getreg returns the location L to hold the value of x for the assignment **x= y op z.**

## The Function *getreg* M

1. If y is in a register $R$ and
   - $R$ holds no other names
   - y is not live / no next use after this statement

   then
   (i) delete $R$ from AD for y; (ii) return $R$.

# Generating Code For Assignment Statements

The assignment d = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence

**t = a – b**

**u = a – c**

**v = t + u**

**d = v + u** with d live at the end.

| Statements | Code Generated | Register Descriptor | Address Descriptor |
|---|---|---|---|
| | | Register Empty | |
| t = a - b | | | |
| u = a - c | | | |
| v = t + u | | | |
| d = v + u | | | |

# Generating Code For Other Type Of Statements

| Statements | Code Generated | Cost |
|---|---|---|
| a = b[i] | MOV b(Ri), R | |
| a[i] = b | | |

- ✓ i is in Ri
- ✓ R is the location returned by getreg()

# Generating Code For Other Type Of Statements

| Statements | Code Generated | Cost |
|---|---|---|
| a = *p | MOV *Rp, a | |
| *p = a | | |

- ✓ p is in Rp

# Generating Code For Other Type Of Statements

| Statements | Code Generated |
|---|---|
| if x < y goto z | CMP x, y<br>CJ < z /*Jump to z if condition code is negative*/ |
| x = y + z<br>if x < 0 goto z | MOV y, R0<br>ADD z, R0<br>MOV R0, x<br>CJ < z |

# PEEPHOLE OPTIMIZATION

- Peephole optimization is a simple and effective technique for locally improving target code.
- This technique is applied to improve the performance of the target program by examining the short sequence of target instructions (called the peephole) and replace these instructions replacing by shorter or faster sequence whenever possible.
- **Peephole is a small, moving window on the target program.**

# PEEPHOLE OPTIMIZATION

- **Characteristics**
  1. Redundant instruction elimination
  2. Unreachable code elimination
  3. Flow of control optimization
  4. Algebraic simplification
  5. Reduction in strength
  6. Machine idioms

1. **Redundant instruction elimination**
   - ✓ Redundant loads and stores can be eliminated in this type of transformations.

     E.g. **MOV R0, x**

     **MOV x, R0**

     We can eliminate the second instruction since x is in already R0.

     But if MOV x, R0 is a label statement then we cannot remove it.

## 2. Unreachable code elimination

- ✓ Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

E.g.
```
void add_ten(int x)
    {
        return x + 10;
        printf("value of x is %d", x);
    }
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

## 3. The flow of control optimization

- ✓ There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed.

E.g.

```
...
MOV R1, R2
GOTO L1
...
L1 : GOTO L2
L2 : INC R1
```

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2.

## 4. Algebraic simplification

- ✓ There are occasions where algebraic expressions can be made simple. For example, the expression **a = a + 0** can be replaced by **a** itself and the expression a = a + 1 can simply be replaced by INC a.

## 5. Reduction in strength

- ✓ There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.
- ✓ For example, **x * 2** can be replaced by **x << 1**, which involves only one left shift.

## 6. Machine Idioms

- ✓ We can replace some target instructions by their equivalent machine instructions in order to improve the efficiency.
- ✓ E.g. some machines have auto-increment or auto-decrement operators. This can be used in a code for a statement like i=i+1.