

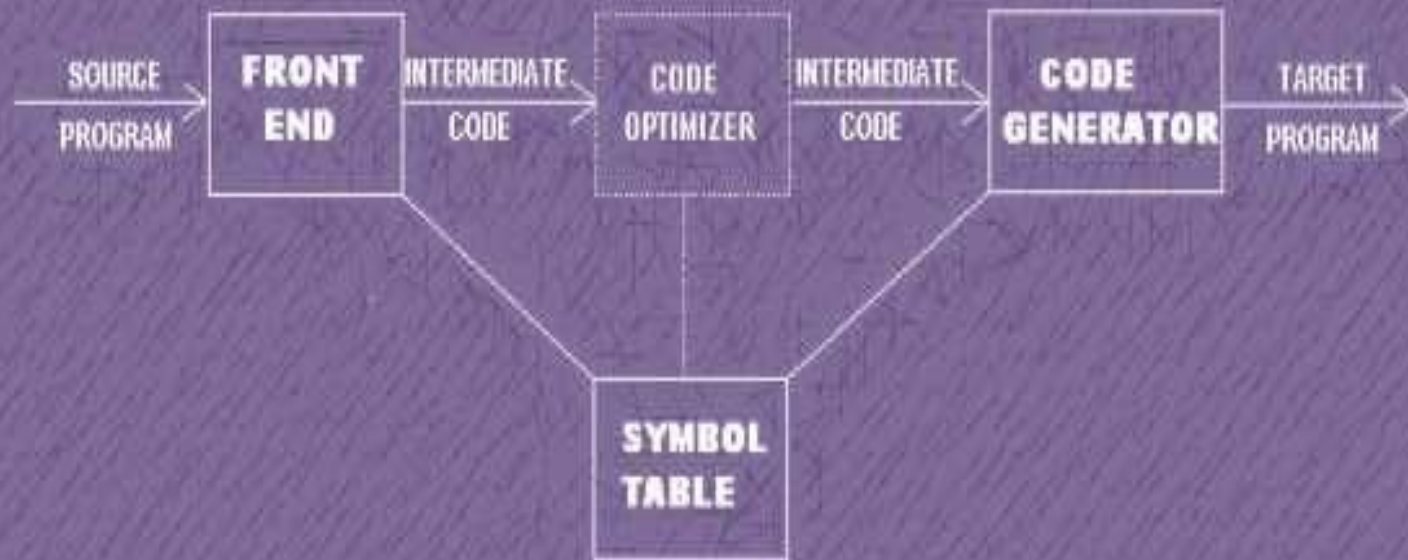
# CODE GENERATION

# Introduction

- The final phase of our compiler model is code generator.
- It takes input from the intermediate representation with supplementary information in symbol table of the source program and produces as output an equivalent target program.
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering

- Instruction selection
  - choose appropriate target-machine instructions to implement the IR statements
- Register allocation and assignment
  - decide what values to keep in which registers
- Instruction ordering
  - decide in what order to schedule the execution of instructions

## ***POSITION OF CODE GENERATOR***



# Issues in the design of code generator

- Input to the code generator
- Target program
- Memory management
- Instruction selection
- Register allocation
- Choice of evaluation order
- Approaches to code generation

# Issues in the design of code generator

- Input to the code generator
  - IR + Symbol table
  - IR has several choices
    - Postfix notation
    - Syntax tree
    - Three address code
  - We assume front end produces low-level IR, i.e. values of names in it can be directly manipulated by the machine instructions.
  - Syntactic and semantic errors have been already detected

# Issues in the design of code generator

- The target program

- The output of code generator is target program.
- Output may take variety of forms
  - Absolute machine language(executable code)
  - Relocatable machine language(object files for linker)
  - Assembly language(facilitates debugging)
- Absolute machine language has advantage that it can be placed in a fixed location in memory and immediately executed.
- Relocatable machine language program allows subprograms to be compiled separately.
- Producing assembly language program as o/p makes the process of code generation somewhat easier.

# Issues in the design of code generator

- **Memory management**
  - Mapping names in the source program to addresses of data objects in run time memory is done by front end & code generator.
  - A name in the three address statement refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.



# Issues in the design of code generator

- **Instruction selection**

- Uniformity and completeness of the instruction set are imp factors
- If we do not care about the efficiency of the target program, instruction selection is straightforward.
- The quality of the generated code is determined by its speed and size.
- For ex, this is statement by statement code generation.

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

# This may produces poor code

## Example #1

$a = b + c$

$d = a + e$

```
MOV    B,R0
ADD    c, R0
MOV    R0,a
MOV    a,R0
ADD    e, R0
MOV    R0,d
```

## Example #2

INC a

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

# Issues in the design of code generator

- **Register allocation**

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- Two subproblems
  - Register allocation and assignment : A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
- Complications imposed by the hardware architecture
  - Example: register pairs for multiplication and division

Example:

$t = a + b$

$t = t * c$

$T = t / d$

L	R1, a
A	R1, b
M	R0, c
D	R0, d
ST	R1, t

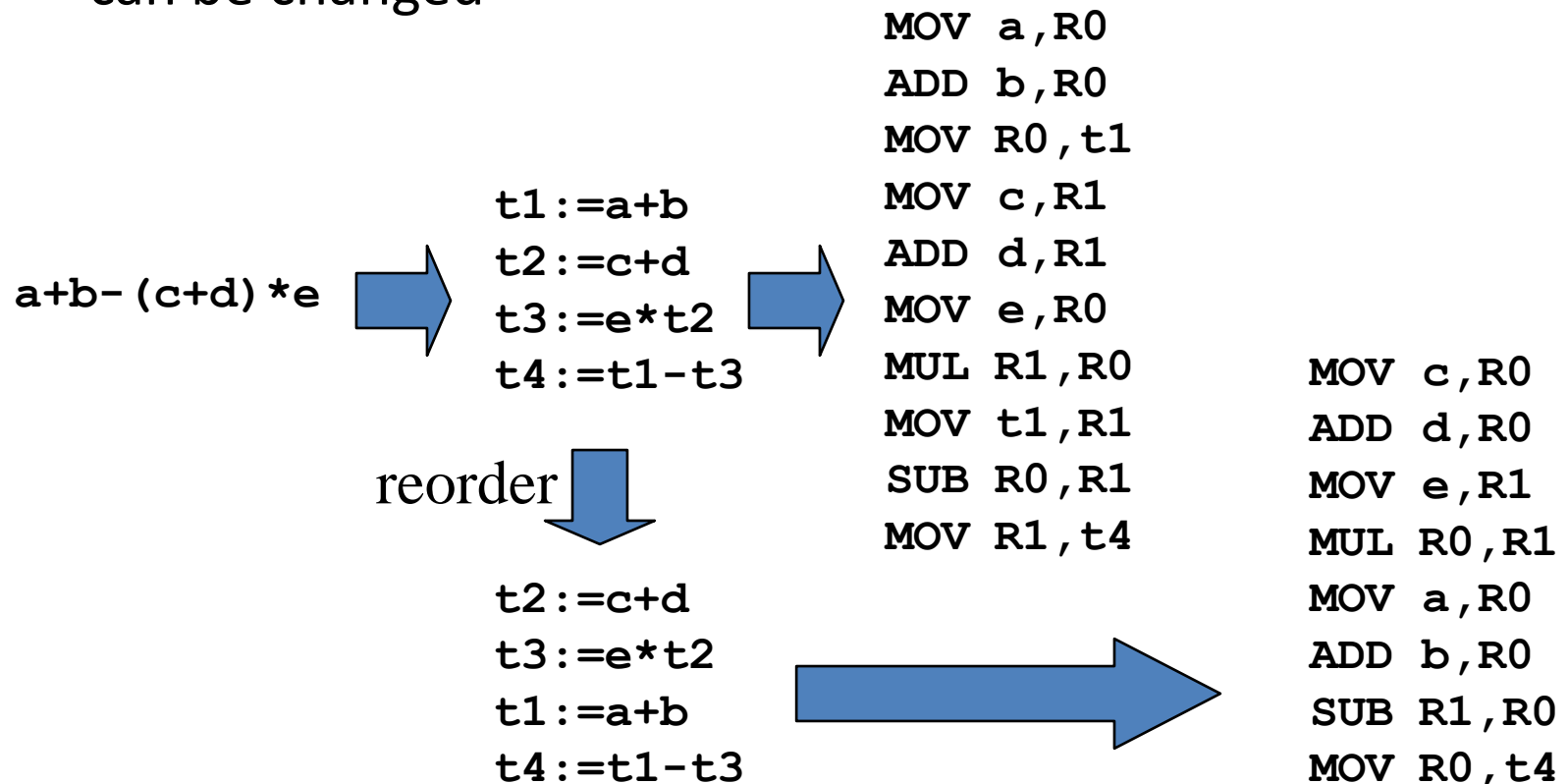
# Issues in the design of code generator

- Choice of Evaluation Order
  - The order in which the computations are performed can affect the efficiency of a target code.
  - Some computations required fewer registers to hold intermediate results than other.
  - Picking the best order is another difficult problem

# Issues in the design of code generator

- Choice of evaluation order

- The order in which computations are performed can affect the efficiency of the target code.
- When instructions are independent, their evaluation order can be changed



# Issues in the design of code generator

- Approaches to code generator
  - Criterion for a code generator is to produce correct code.
  - Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

# Target machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set
- Our (hypothetical) machine:
  - Byte-addressable (word = 4 bytes)
  - Has  $n$  general purpose registers **R0**, **R1**, ..., **R $n$ -1**
  - Two-address instructions of the form  
$$op\ source,\ destination$$
  - *Op* – op-code
  - *Source, destination* – data fields



# The Target Machine: Op-codes

- Op-codes (op), for example
- MOV (move content of source to destination)
- ADD (add content of source to destination)
- SUB (subtract content of source from  
destination)
- There are also other ops

# The Target Machine: Address modes

- **Addressing mode:** Different ways in which location of an operand can be specified in the instruction.

Mode	Form	Address	Added Cost
Absolute	<b>M</b>	<b>M</b>	1
Register	<b>R</b>	<b>R</b>	0
Indexed	$c(\mathbf{R})$	$c + \text{contents}(\mathbf{R})$	1
Indirect register	$*\mathbf{R}$	$\text{contents}(\mathbf{R})$	0
Indirect indexed	$*c(\mathbf{R})$	$\text{contents}(c + \text{contents}(\mathbf{R}))$	1
Literal	<b>#c</b>	N/A	1

# Instruction Costs

- Machine is a simple processor with fixed instruction costs
- In most of the machines and in most of the instructions the time taken to fetch an instruction from memory exceeds the time spent executing the instruction. So reducing the length of the instruction has an extra benefit.

# Examples


Instruction	Operation	Cost
MOV R0 , R1	Store <i>content</i> ( <b>R0</b> ) into register <b>R1</b>	1
MOV R0 , M	Store <i>content</i> ( <b>R0</b> ) into memory location <b>M</b>	2
MOV M , R0	Store <i>content</i> ( <b>M</b> ) into register <b>R0</b>	2
MOV 4 (R0) , M	Store <i>contents</i> (4+ <i>contents</i> ( <b>R0</b> )) into <b>M</b>	3
MOV *4 (R0) , M	Store <i>contents</i> ( <i>contents</i> (4+ <i>contents</i> ( <b>R0</b> ))) into <b>M</b>	3
MOV #1 , R0	Store 1 into <b>R0</b>	2
ADD 4 (R0) , *12 (R1)	Add <i>contents</i> (4+ <i>contents</i> ( <b>R0</b> )) to value at location <i>contents</i> (12+ <i>contents</i> ( <b>R1</b> ))	3

# Instruction Selection

- Instruction selection is important to obtain efficient code
- Suppose we translate three-address code

$x := y + z$

to: **MOV**  $y, R0$   
**ADD**  $z, R0$   
**MOV**  $R0, x$

$a := a + 1$   **MOV**  $a, R0$   
**ADD**  $\#1, R0$   
**MOV**  $R0, a$   
Cost = 6

Better



**ADD**  $\#1, a$   
Cost = 3

Best



**INC**  $a$   
Cost = 2

# Instruction Selection: Utilizing Addressing Modes

- Suppose we translate  $a := b + c$  into

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

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

- Assuming addresses of  $a$ ,  $b$ , and  $c$  are stored in  $R0$ ,  $R1$ , and  $R2$

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

- Assuming  $R1$  and  $R2$  contain values of  $b$  and  $c$

```
ADD R2, R1
MOV R1, a
```

# Run Time Storage Management

- The information needed during an execution of a procedure is kept in a block of storage called activation record.
- Whenever a procedure is called a stack frame is allocated on the stack
- The information stored in the stack are return address, local variables, temporaries, return values etc.
- 2 standard storage allocation strategies namely
  - Static allocation
  - Stack allocation

- In static allocation the position of an activation record in memory is fixed at compile time.
- In stack allocation a new activation record is pushed onto the stack for each execution of the procedure. The record is popped when the activation ends.



# Peephole Optimization

- Statement by statement code generation strategy often produces target code that contains redundant instructions and suboptimal constructs.
- No guarantee that resulting code is optimal under any mathematical measure.
- ***Peephole optimization***: a short sequence of target instructions(peephole instructions) that can be replaced by shorter or faster sequence instructions.

- Repeated passes over the target code are necessary to get the maximum benefit.
- Characteristics of peephole optimization
  - Redundant- instruction elimination
  - Flow of control optimizations
  - Algebraic simplifications
  - Use of machine idioms

# Redundant Loads and Stores

```
MOV R0 a
```

```
MOV a R0
```

# Unreachable code

- Unlabeled instruction following an unconditional jump may be eliminated

```
#define DEBUG 0
...
if (debug) {
    /* print stmts */
}

if debug = 1 goto L1
goto L2
L1: /* print */
L2:

⇓

if 0 != 1 goto L2
/* print stmts */
L2:

⇐

if debug != 1 goto L2
/* print stmts */
L2:

eliminate
```

# Flow of control optimizations

```
    goto L1
    ...
L1:  goto L2
    ↓
    goto L2
    ...
L1:  goto L2
```

```
    if a < b goto L1
    ...
L1:  goto L2
    ↓
    if a < b goto L2
    ...
L1:  goto L2
```

goto L1

...

L1: if a < b goto L2

L3:

if a < b goto L2

goto L3

...

L3:

# Algebraic simplification

- Eliminate the instructions like the following
  - $x = x + 0$
  - $x = x * 1$

These type of instructions are often produced by straightforward intermediate code generation algorithms, and they can be easily eliminated by peephole optimization.

# Reduction in strength

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

ex,

$$X^2 = X * X$$



# Use of machine idioms

- The target machine may have hardware instructions to implement certain specific operations efficiently.
- Detecting situations that permit the use of these instructions can reduce execution time significantly.
- Ex,
  - Increment and decrement addressing modes