# CS 4031
# Compiler Construction
# Lecture 15

Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

## Code Generator

- **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

## Code Generator

- Code generator is the last phase in the design of a compiler. It takes three address code or DAG representation of the source program as input and produces an equivalent target program as output.
- The code generator has many limitations regarding the generation of code that is of high quality, accurate, and efficient. In addition to this, the code generator should run efficiently. There are many issues that are to be considered while designing a code generator.

# Code Generator

- 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 instruction

## Example: Instruction Selection

- Intermediate Statement

t1 = a + b

- Instruction Selection

```
ADD EAX, EBX
MOV t1, EAX
```

## Example: Register Allocation

- Intermediate Statement

```
t1 = a + b
```

- Register Allocation

```
MOV R1, a
MOV R2, b
ADD R1, R2
MOV t1, R1
```

## Example: Instruction Ordering

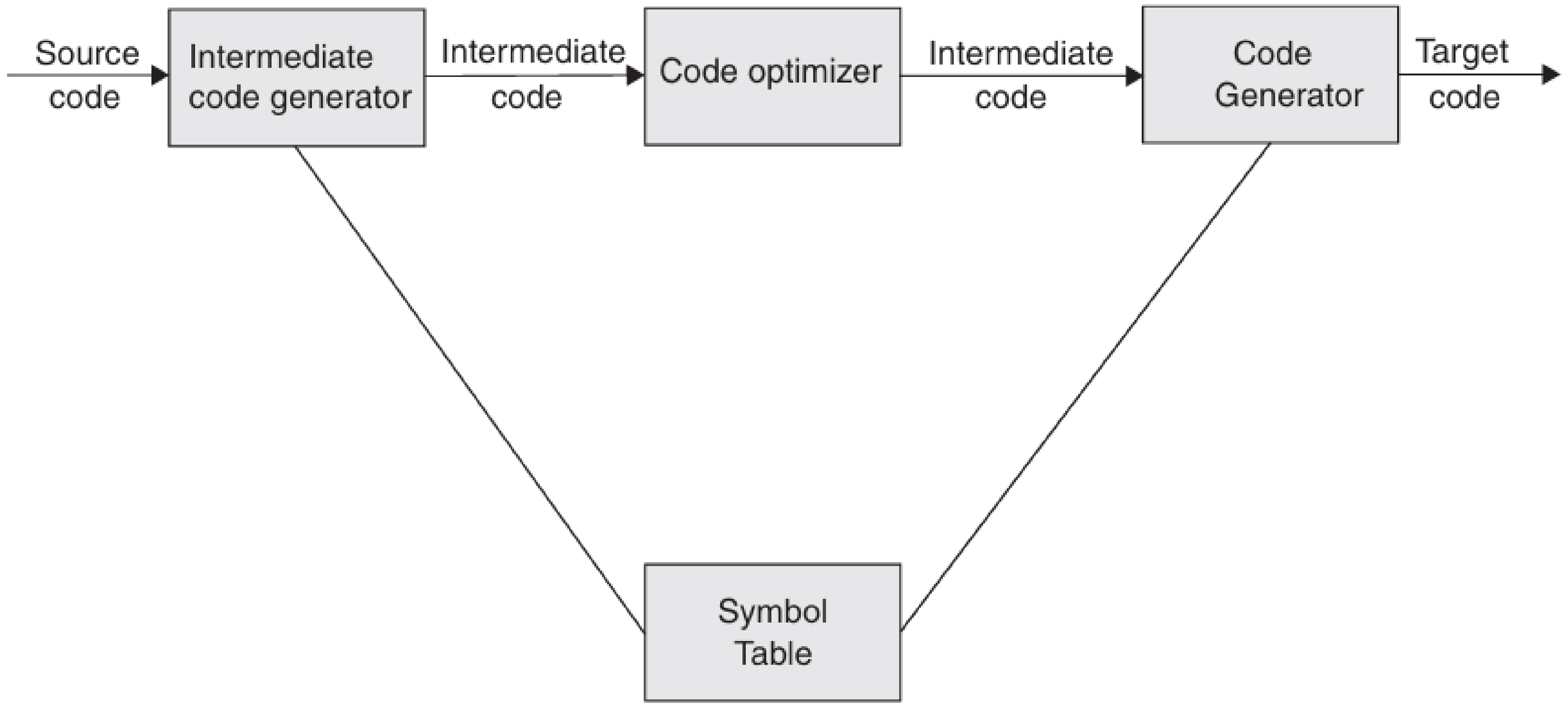
- Intermediate Statement

```
x = a + b;
y = x * c;
```

- Instruction Ordering

```
MOV R1, a
MOV R2, b
ADD R1, R2
MOV R3, c
MUL R1, R3
```

# Issues in the Design of a Code Generator



## Issues in the Design of a Code Generator

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

# Input 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

# Target Code

- 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)

## Target Code

- Absolute machine language has advantage that it can be placed in a fixed location in memory and immediately executed.
  - Example:
    - Early microcontroller programming
    - Programs written in raw binary format for early processors like Intel 8080.
- Relocatable machine language program allows subprograms to be compiled separately.
  - Example:
    - C and C++ with object files
    - A .obj file generated from a C program can be linked with other .obj files to form an executable

## Target Code

- Producing assembly language program as o/p makes the process of code generation somewhat easier.
  - Example:
  - GNU Assembly (GAS) output from compilers
  - LLVM or GCC producing .s assembly files from C/C++ code before assembling them

## Memory Management

- Mapping names in the source program to addresses of data objects in run time memory is done by front end & code generator.
- If a machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to “backpatching” technique.

# 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, …, Rn-1
  - Two-address instructions of the form

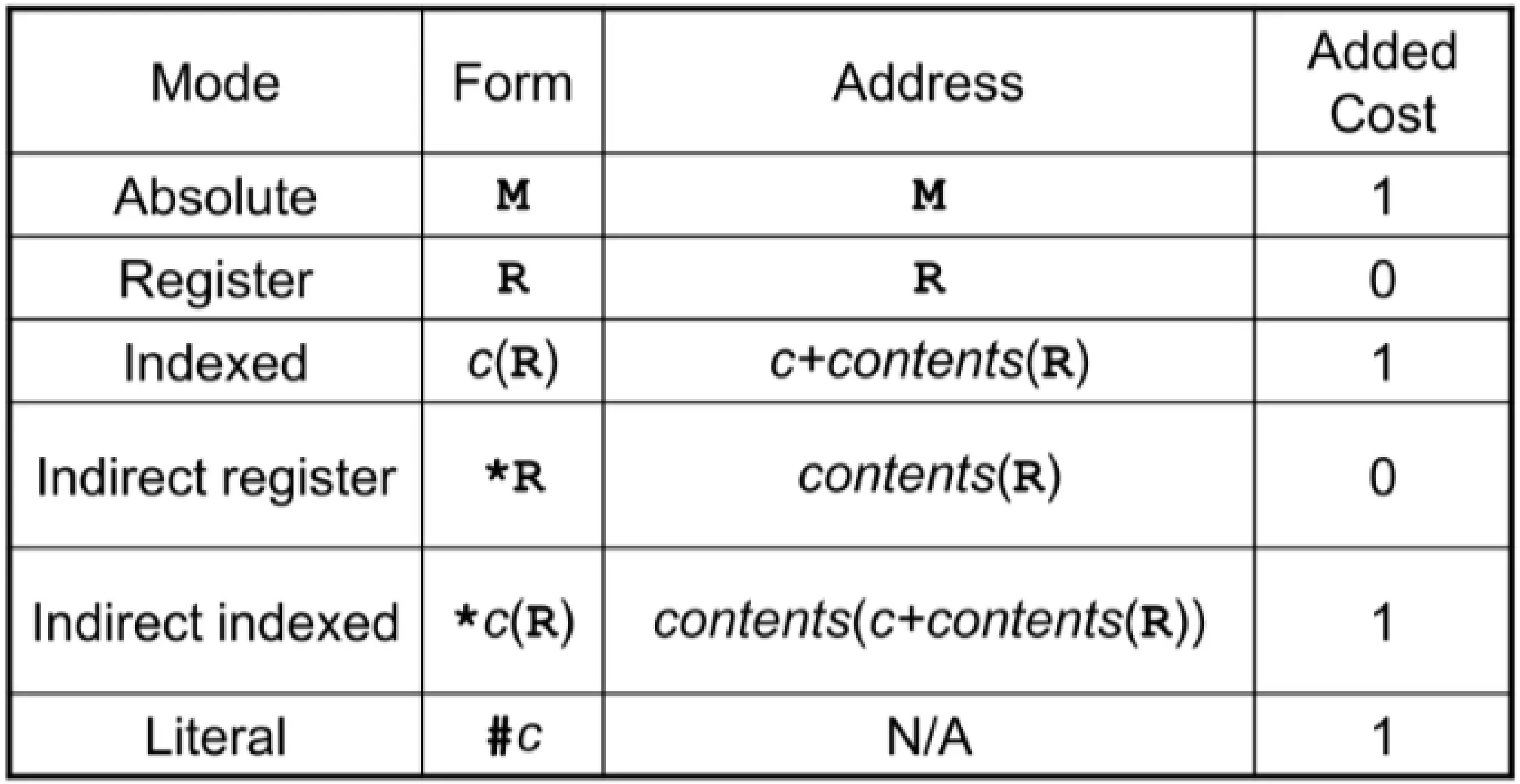
    op source, destination
  - Op – op-code
  - Source, destination
  - data fields

## OP Codes

- Op-codes Op-codes (op)
- 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 18

## Target Machine Addressing Modes



| 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 |
| Literal | **#***c* | N/A | 1 |

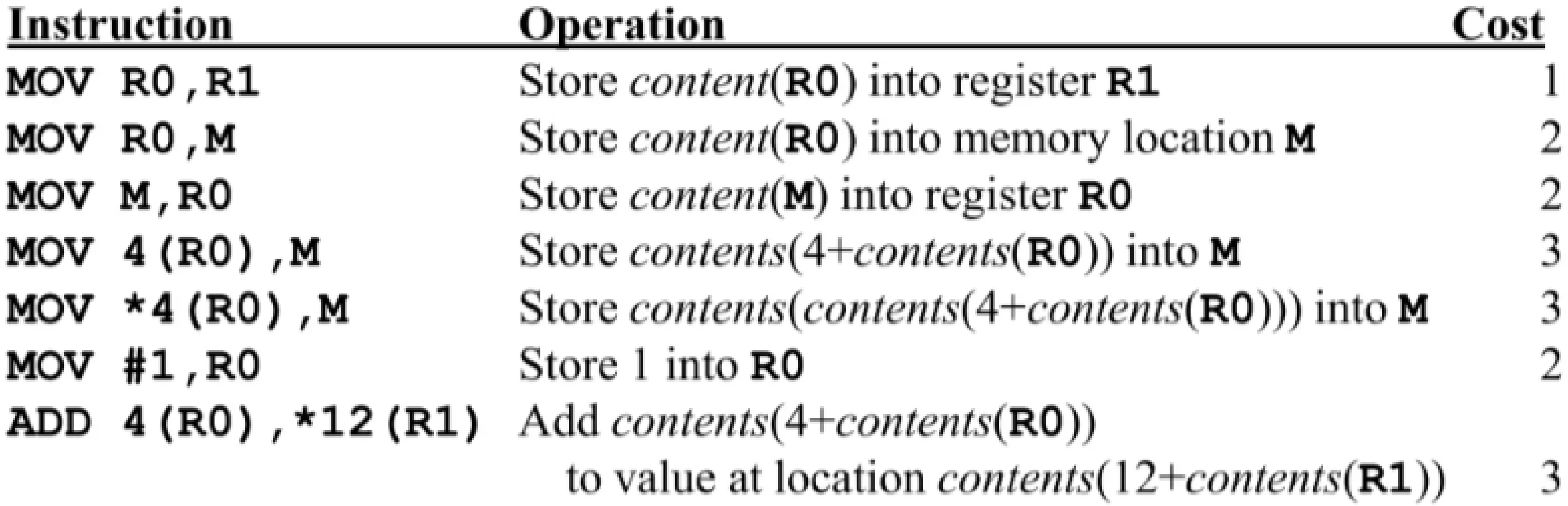
# Target Machine Addressing Modes

| Mode | Purpose |
|---|---|
| Absolute | A memory location is specified explicitly<br>data is retrieved from that exact location |
| Register | The operand is stored in a register<br>instruction accesses data directly from a CPU register, leading to faster execution. |
| Indexed | memory address is computed by adding a constant<br>useful for accessing array elements, |
| Indirect Register | The register contains the actual memory address where data is stored<br>pointer to the memory location, enabling indirect access. |
| Indirect Indexed | The computed address holds the final operand’s location |
| Literal Addressing | A constant (c) is directly included as an operand.<br>The instruction itself contains a hardcoded value instead of a memory reference, avoiding memory access entirely. |

## Instruction Cost

- **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.

## Example



| Instruction | Operation | Cost |
|---|---|---|
| `MOV R0,R1` | Store *content*(**R0**) into register **R1** | 1 |
| `MOV R0,M` | Store *content*(**R0**) into memory location **M** | 2 |
| `MOV M,R0` | Store *content*(**M**) into register **R0** | 2 |
| `MOV 4(R0),M` | Store *contents*(4+*contents*(**R0**)) into **M** | 3 |
| `MOV *4(R0),M` | Store *contents*(*contents*(4+*contents*(**R0**))) into **M** | 3 |
| `MOV #1,R0` | Store 1 into **R0** | 2 |
| `ADD 4(R0),*12(R1)` | Add *contents*(4+*contents*(**R0**)) to value at location *contents*(12+*contents*(**R1**)) | 3 |

# Calculate the Cost

- Mov R0, R1
  - Operation: - Copy the contents of register R0 into register R1.
  - Read value from R0 → Cost = 0
  - Store value into R1 → Cost = 1
  - Total Cost = 1

- MOV R0, M
  - Operation: Store the contents of register R0 into memory location M.
  - Read value from R0 → Cost = 0
  - Find memory location M → Cost = 1
  - Store value into M → Cost = 1
  - Total Cost = 2

## Running Procedure

- llvm-as name_file_LLVM –o OutputfileName
- llc OutputfileName -filetype=obj -o objectfileName

- clang objectfileName -o YourCompiler