CODE GENERATION
MODULE 5

Prepared by

Jesna J S

Assistant Professor CSE, TKMCE

- The target program must preserve the semantic meaning of the source program and be of high quality.
- It must make use of the available resources of the target machine.
- It must **produce correct and efficient code**.

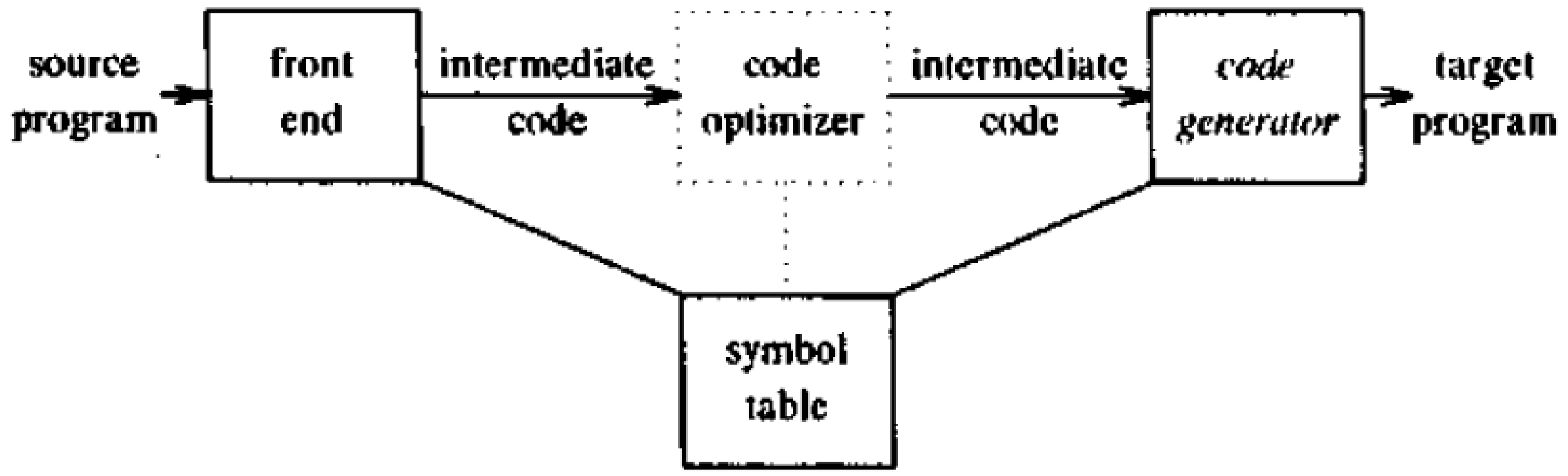


Fig. 9.1. Position of code generator.

Issues in Code generator

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

Input to code generator

- The input to the code generator is an intermediate representation of the source program.
- It includes **symbol table information for resolving data object addresses**.
- Inputs to the code generator are..
  - ✓**Linear representations (e.g., postfix notation).**
  - ✓**Three-address representations (e.g., quadruples).**
  - ✓**Virtual machine representations (e.g., stack machine code).**
  - ✓**Graphical representations (e.g., syntax trees, Directed Acyclic Graphs (DAGs))**
- Names in intermediate code are mapped to values that the target machine can handle (e.g., integers, floats, pointers) in the previous phases.
- The code generator assumes no errors in the input.

Target Program

- The **output of the code generator is the target program**.
- Types of outputs:
  - ✓*Absolute machine language*
  - ✓*Relocatable machine language*
  - ✓*Assembly language*
- If producing assembly-language program as output:
  - Advantage: **Code generation process will be easier**.
    - **Can generate symbolic instructions use the macro facilities of assembler** to help generate code.
- If producing relocatable machine language as output:
  - Can **compile subroutines separately**.
  - But **explicit relocation information must be provided to loader** to link separately compiled program segments.

Memory Management

- The front end and code generator cooperate **to map names in the program into memory address**.
- A name in a three-address statement refers to an entry in the symbol table.
- When a **declaration is made, a symbol-table entry is created**.
- The data type of a variable determines its storage size.
- From symbol-table data, a relative address is assigned to each variable.

✓***Static*** *allocation:* ***Fixed memory*** *locations are assigned.*

✓***Stack*** *allocation: Memory is* ***assigned dynamically*** *during execution.*

Backpatching and Label Resolution

- **Labels** in three-address code must be **converted to machine instruction addresses**.
- A technique called **backpatching** is used to handle **forward jumps** in code.

Backpatching Mechanism

- Labels refer to **quadruple numbers** in a quadruple array.
- The compiler scans quadruples and tracks the **machine instruction count**.
- If a label points to a **previous instruction**, the **jump address is directly assigned**.
- If a label points to a **future instruction**, it is **stored and later resolved** when the target instruction is encountered.

Instruction Selection

- The **target machine's instruction set** affects how instructions are selected.
- If the machine does not handle data types uniformly ; **special handling** is needed.

Factors Affecting Instruction Selection

- **Instruction speed and machine-specific optimizations are important.**
- *If efficiency is ignored, **straightforward code generation** can be done.*
- Given a three-address statement:

  x := y + z

- It can be translated into machine code:

```
MOV y, R0   /* Load y into register R0 */
ADD z, R0   /* Add z to R0 */
MOV R0, x   /* Store result in x */
```

- *This approach is simple but may generate **redundant instructions***

- Example of Redundant Code

```
a := b + c
d := a + e
```

Translated machine code

```
MOV b, R0
ADD c, R0
MOV R0, a
MOV a, R0  /* Redundant */
ADD e, R0
MOV R0, d
```

Optimizing Instruction Selection

- Efficient code requires minimizing unnecessary instructions.
- Machines with advanced instruction sets can implement operations in fewer steps.

  MOV a, R0

  ADD #1, R0 // The **increment instruction (INC)** can be used: INC a

  MOV R0, a

  This reduces execution time and improves efficiency.

Challenges in Instruction Selection

1. **Finding the best machine-code sequence requires context awareness.**
2. **Accurate timing information is often hard to obtain.**

Register Allocation

Why Register Allocation is Important?

- Register-based instructions are faster than memory-based ones.
- Efficient utilization of registers improves performance.

Challenges:

- Finding an optimal assignment is NP-complete (computationally difficult).
- Some machines have special register usage conventions.

Optimization Techniques:

- Reduce register moves.
- Use machine-specific instructions (SRDA for shifting, etc.).
- Load variables only when needed.

Evaluation Order

Importance of Evaluation Order

- **The sequence in which computations are performed impacts code efficiency.**
- Some orders reduce the number of registers required for intermediate results.

Challenges in Choosing the Best Order

- Finding the most efficient order is **NP-complete** (computationally difficult).

To simplify, the compiler initially **generates code in the order in which three-address statements are produced** by the intermediate code generator.

The order of evaluation affects register usage and efficiency, but finding an optimal order is complex. Initially, the compiler follows a straightforward approach.

CODE GENERATION(THE TARGET LANGUAGE)

- **Familiarity with the target machine and instruction set is essential** for code generation.
- A register-based machine is used as a reference.
- The target machine has:
  - Byte-addressable memory (4 bytes per word).
  - General-purpose registers (R0, R1, ..., Rn-1).
  - Two-address instructions in the form:

    Op source, destination

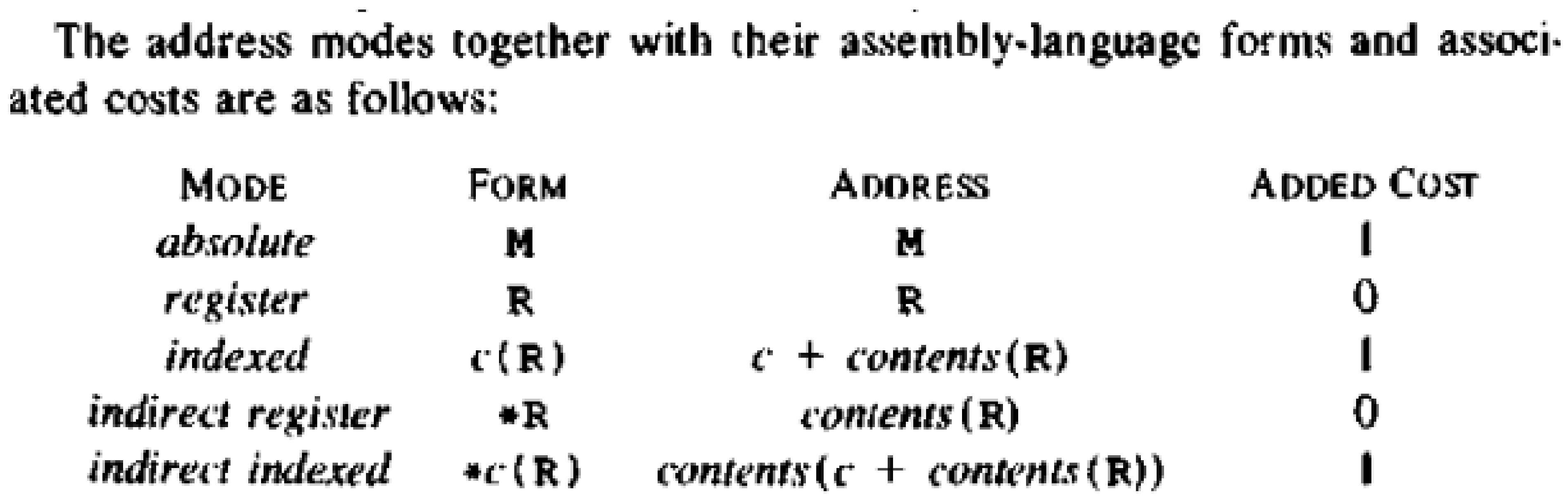
    ✓It has the following opcodes:

    ✓MOV (move source to destination)

    ✓ADD (add source to destination)

    ✓SUB (sub source from destination)

- The source and destination fields are not long enough to hold memory addresses so certain bit patterns in these 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 modes.
- *In the following description,* ***contents(a) denotes the contents of the register or memory address represented by a***



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

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

- A memory location M or a register R represents itself when used as a source or destination. For example, the instruction.

  MOV R0, M - stores the contents of register R0 into memory location M

  MOV 4(R0), M - stores the value of contents(4 + contents(R0)) into memory location M

  MOV #1, R0 - loads the constant 1 into register R0.

Instruction Cost

- The cost of an instruction is **one plus the costs associated with the source and destination address modes**.
- This cost corresponds to the length (in words) of the instruction.
- **Register address cost is zero**.
- But with a memory location or literal, the cost is one [Because such operands have to be stored with the instruction]
- To reduce the space the instruction length should be minimized.
- Minimizing the instruction length we also 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.

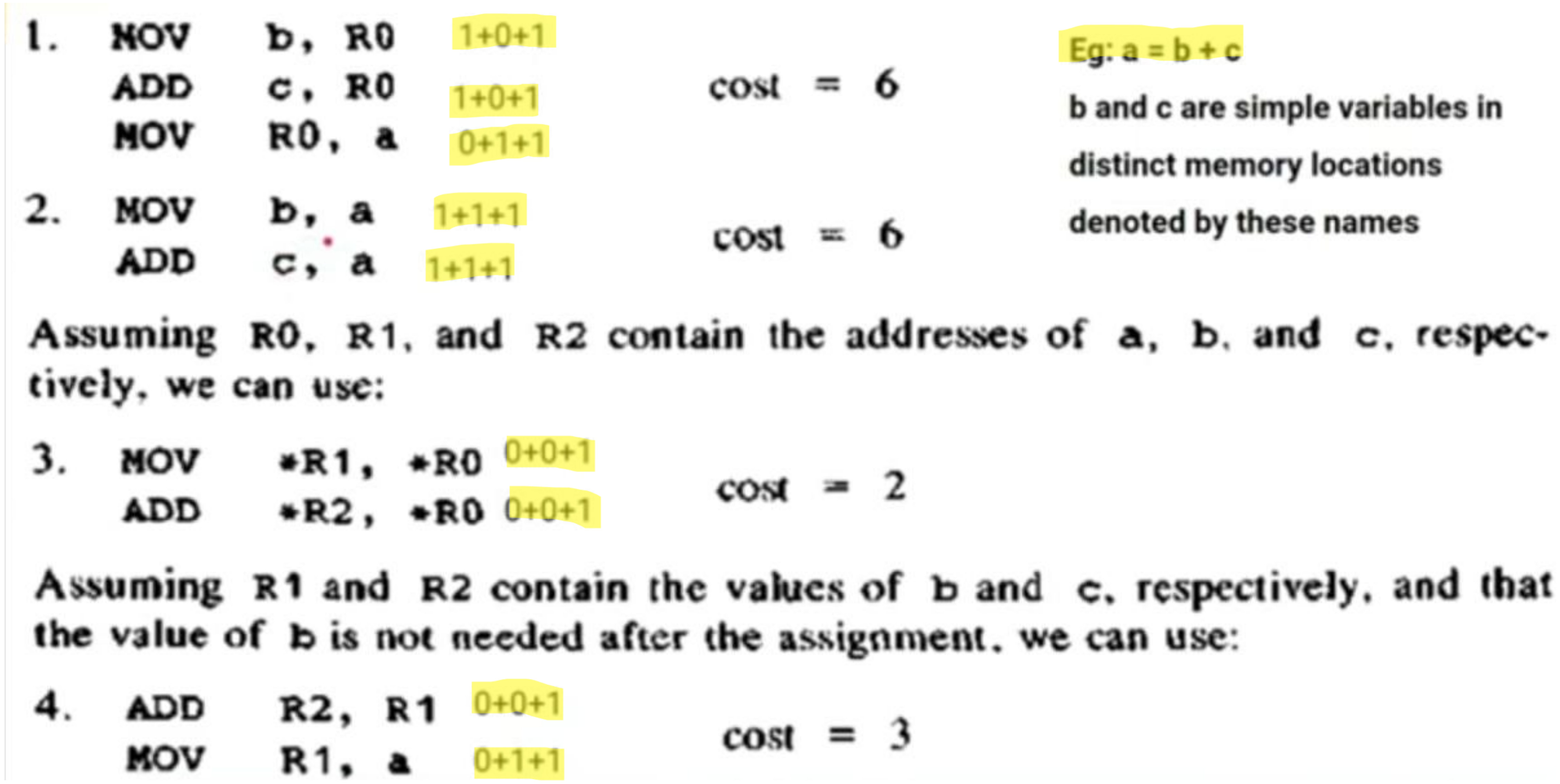
2. The (store) instruction `MOV R5,M` copies the contents of register `R5` into memory location `M`. This instruction has cost two, since the address of memory location `M` is in the word following the instruction.

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.

4. The instruction `SUB 4(R0),*12(R1)` stores the value

   $$contents(contents(12 + contents(\mathtt{R1}))) - contents(contents(4 + \mathtt{R0}))$$

   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.



Eg: a = b + c

b and c are simple variables in distinct memory locations denoted by these names

```
1.  MOV   b, R0     1+0+1
    ADD   c, R0     1+0+1        cost = 6
    MOV   R0, a     0+1+1

2.  MOV   b, a      1+1+1
    ADD   c, a      1+1+1        cost = 6
```

Assuming R0, R1, and R2 contain the addresses of a, b, and c, respectively, we can use:

```
3.  MOV   *R1, *R0  0+0+1
    ADD   *R2, *R0  0+0+1        cost = 2
```

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

```
4.  ADD   R2, R1    0+0+1
    MOV   R1, a     0+1+1        cost = 3
```

Register and Address Descriptors

- A **register descriptor** *keeps track of what is currently in each register*.
- It is consulted whenever a new register is needed.
- An **address descriptor** *keeps track of the location where the current value of the name can be found at run time(whether it's in memory, a register, or both).*

getreg()

How storage location for the result is chosen?

x = y op z

- *If ‘y’ is in a register and not live and has no next use after the instruction, then the register stored y will be taken.*
- *Else if a new register is available , then it will be taken.*
- *If no new register will be available, then select an already occupied register and copy its value to any memory location and that register will be taken.*

Algorithm: Code Generation for Three-Address Statements

Input: A sequence of three-address statements forming a basic block. Each statement is of the form: **x = y op z**

Output: A sequence of assembly instructions to compute the result and store it efficiently.

Method:

Step1: Determine the Storage Location for the Result:

- Invoke the function *getreg()* to determine where the result of *y op z* should be stored.
- Preferably, this location should be a register, but it may also be a memory location.

Step2: Retrieve Operand y:

- Use the address descriptor to find one of the current locations of y.
- If y is in both memory and a register, prefer the register.
- If y is **not in the chosen destination L**, generate the instruction: MOV y’, L

Step3: Perform the Operation:

- Generate the instruction: op z’ , L where z′ is the current location of z.
- Prefer a register over memory if z is available in both.
- Update the address descriptor of x to indicate it is now in L.
- If L is a register, remove x from all other register descriptors.

Step4: Handle Dead Variables:

- If y and/or z are not used later in the block and are in registers, remove them from the register descriptor to free up registers.

Handle Assignment Statements:

- If the three-address statement is of the form x:=y
  - If y is in a register, simply update the descriptors.
  - If y is in memory, load it into a register first

Store Live Variables at Block Exit

- At the end of the basic block, ensure variables that are live at exit and not in memory are stored back using MOV instructions.

Example 9.5. 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
```

STATEMENTS	CODE GENERATED	REGISTER DESCRIPTOR	ADDRESS DESCRIPTOR
		registers empty	

`t := a - b`	`MOV a, R0`
	`SUB b, R0`
		R0 contains t	t in R0

`u := a - c`	`MOV a, R1`
	`SUB c, R1`
		R0 contains t
		R1 contains u
			t in R0
			u in R1

`v := t + u`	`ADD R1, R0`	R0 contains v R1 contains u	u in R1 v in R0
`d := v + u`	`ADD R1, R0`	R0 contains d	d in R0
	`MOV R0, d`		d in R0 and memory