# CSD 3202-COMPILER DESIGN

## MODULE IV

## ISSUES IN THE DESIGN OF CODE GENERATOR

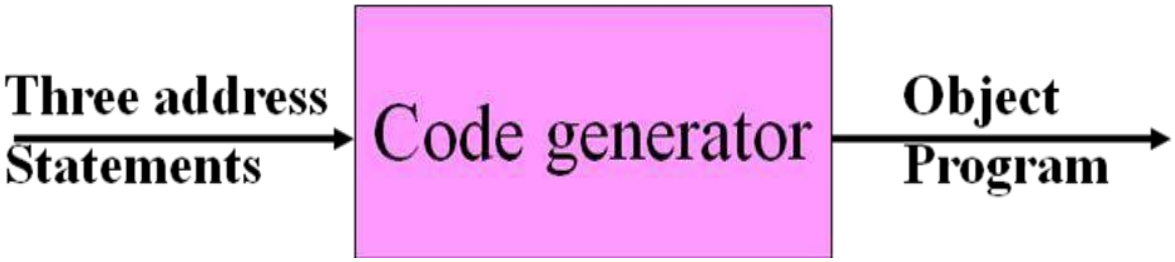
## THE TARGET LANGUAGE

## ADDRESSES IN THE TARGET CODE

## REGISTER ALLOCATION AND ASSIGNMENT

# Code Generation

- It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program



# Issues in the Design of Code Generator

- **The following issue arises during the code generation phase:**

- Input to the Code Generator
- The Target Program
- Instruction Selection
- Register Allocation
- Evaluation Order

# Input to code generator

- The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation.
- Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc.
- The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary

# Target program

- **The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.**
  - **Absolute** machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
  - **Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.**
  - **Assembly** language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

# Instruction selection



- Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.
- For example, three-address statements would be translated into the latter code sequence as shown below:

```
P:=Q+R
S:=P+T
MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

- Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement.
- It leads to an inefficient code sequence.
- A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations.
- A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

# Register allocation issues

- Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:
- During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.
- During a subsequent **Register assignment** phase, the specific register is picked to access the variable.
- As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

  M a, b

- These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

# Evaluation order

- The code generator decides the order in which the instruction will be executed.
- The order of computations affects the efficiency of the target code.
- Among many computational orders, some will require only fewer registers to hold the intermediate results.
- However, picking the best order in the general case is a difficult NP-complete problem.

# Approaches to code generation issues:

- Code generator must always generate the correct code.
- It isessential because of the number of special cases that a code generator might face.
- Some of the design goals of code generator are:
  - Correct
  - Easily maintainable
  - Testable
  - Efficient

# Register allocation

- Two subproblems
  - **Register allocation**: selecting the set of variables that will reside in registers at each point in the program
  - **Register assignment**: selecting specific register that a variable reside in
- Complications imposed by the hardware architecture
  - Example: register pairs for multiplication and division

```
t=a+b
t=t*c
T=t/d
```

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

```
t=a+b
t=t+c
T=t/d
```

```
L     R0, a
A     R0, b
M     R0, c
SRDA  R0, 32
D     R0, d
ST    R1, t
```

# The target Language

- A Simple Target Machine Model
- Program and Instruction Costs

# A simple target machine model

- op source, destination

- Where, op is used as an op-code and source and destination are used as a data field.

It has the following op-codes:

- ADD (add source to destination)
- SUB (subtract source from destination)
- MOV (move source to destination)

The source and destination of an instruction can be specified by the combination of registers and memory location with address modes.

# A simple target machine model

- Load operations: LD r,x and LD r1, r2
- Store operations: ST x,r
- Computation operations: OP dst, src1, src2
- Unconditional jumps: BR L
- Conditional jumps: Bcond r, L like BLTZ r, L

# A simple target machine model

| MODE | FORM | ADDRESS | EXAMPLE | ADDED COST |
|---|---|---|---|---|
| Absolute | M | M | Add R0, R1 | 1 |
| Register | R | R | Add temp, R1 | 0 |
| indexed | c(R) | C+ contents(R) | ADD 100 (R2), R1 | 1 |
| indirect register | *R | contents(R) | ADD * 100 | 0 |
| indirect indexed | *c(R) | contents(c+ contents(R)) | (R2), R1 | 1 |
| literal | #c | c | ADD #3, R1 | 1 |

- Here, cost 1 means that it occupies only one word of memory.
- Each instruction has a cost of 1 plus added costs for the source and destination.
- Instruction cost = 1 + cost is used for source and destination mode.

# Program and Instruction Costs

- Cost of an instruction to be one plus the costs associated with the addressing modes of the operands . This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction.

Examples:

- The instruction LD RO, Rl copies the contents of register Rl into register RO. This instruction has a cost of one because no additional memory words are required.
- The instruction LD RO, M loads the contents of memory location M into register RO. T h e cost is two since the address of memory location M is in the word following the instruction.
- The instruction LD R l , *100(R2) loads into register Rl the value given by contents(contents(100 + contents(K2))). The cost is three because the constant 100 is stored in the word following the instruction.

# Example:

1. Move register to memory R0 → M
- MOV R0, M
- cost = 1+1+1 (since address of memory location M is in word following the instruction)

2. Indirect indexed mode: MOV * 4(R0), M
- cost = 1+1+1 (since one word for memory location M, one word result of *4(R0) and one for instruction)

3. Literal Mode:
- MOV #1, R0
- cost = 1+1+1 = 3 (one word for constant 1 and one **for** instruction)

# ADDRESSES IN THE TARGET CODE

- The information which required during an execution of a procedure is kept in a block of storage called an activation record. The activation record includes storage for names local to the procedure.
- We can describe address in the target code using the following ways:
    - Static allocation
    - Stack allocation
- In static allocation, the position of an activation record is fixed in memory at compile time.
- In the stack allocation, for each execution of a procedure a new activation record is pushed onto the stack. When the activation ends then the record is popped.

# ADDRESSES IN THE TARGET CODE

For the run-time allocation and deallocation of activation records the following three-address statements are associated:

- Call
- Return
- Halt
- Action, a placeholder for other statements

Assume that the run-time memory is divided into areas for:

- Code
- Static data
- Stack

# *Static allocation:*

1. Implementation of call statement:

The following code is needed to implement static allocation:

MOV #here + 20, callee.static_area /*it saves return address*/</p>

GOTO callee.code_area /* It transfers control to the target code for the called procedure*/

Where,

**callee.static_area** shows the address of the activation record.

**callee.code_area** shows the address of the first instruction for called procedure.

**#here + 20** literal are used to return address of the instruction following GOTO.

# *Static allocation:*

2. Implementation of return statement:

- The following code is needed to implement return from procedure callee: GOTO * callee.static_area
- It is used to transfer the control to the address that is saved at the beginning of the activation record.

3. Implementation of action statement:

The HALT statement is the final instruction that is used to return the control to the operating system.

# *Stack allocation:*

- Using the relative address, static allocation can become stack allocation for storage in activation records.
- In stack allocation, register is used to store the position of activation record so words in activation records can be accessed as offsets from the value in this register.
- The following code is needed to implement stack allocation:

1. **Initialization of stack:**

```
MOV #stackstart , SP   /*initializes stack*/
HALT                   /*terminate execution*/
```

# *Stack allocation:*



2. **Implementation of Call statement:**

```
ADD #caller.recordsize, SP      /* increment stack pointer */
MOV #here + 16, *SP             /*Save return address */
GOTO callee.code_area
```

Where,

**caller.recordsize** is the size of the activation record

**#here + 16** is the address of the instruction following the **GOTO**

3. **Implementation of Return statement:**

```
GOTO *0 ( SP )          /*return to the caller */
SUB #caller.recordsize, SP    /*decrement SP and restore to previous value */
```