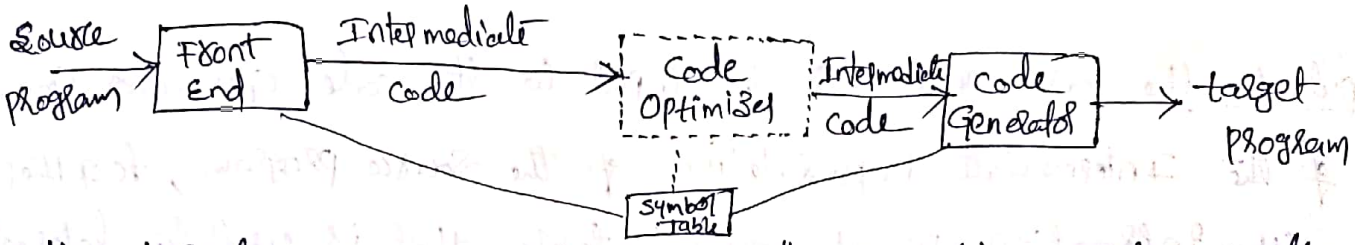
Code Generation:

* the final phase in our compiler model is the Code Generator.
* 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.



* the target program must preserve the semantic meaning of the source program and be of high quality i.e, it must make effective use of the available Resources of the Target machine.
* Moreover, (it must make effective) the Code Generator itself must Run efficiently.
* Compilers that need to produce efficient target programs, include an optimization phase prior to code generation.
* the optimizer maps the IR into IR from which more efficient code can be generated.
* In General, the code optimization and code generation phases of a compiler, often referred to as the back end.

Issues in the Design of a Code Generator:

The issues in the design of code generators are,

- → Input to the code generator.
- → Target programs.
- → Instruction selection
- → Register Allocation
- → Evaluation order.

Input to the code generator: The input to the code Generator Consists of the Intermediate Representation of the source program, together with information in the symbol table that is used to determine the Runtime addresses of the data objects denoted by the names in the IR.

* we assume that prior to code generator the i/p has been validated by the front end, ie, typechecking, syntax, semantics etc.

Target program: The output of the code generation is the Target program. The target program may take a variety of forms i.e; absolute machine language, Relocatable machine language (or) Assembly language.

absolute machine language: output that it can be placed in a fixed location in memory and immediately executed.

Relocatable machine code: output allows subprograms to be compiled separately. Although we must pay the added expense of linking and loading if we produce Relocatable object modules.

producing an assembly code as output makes the process of code generation easier as we can generate symbolic instructions.

Instruction selection: the nature of the instruction set of the target machine determine the difficulty of instruction selection. If each type of three address statement, we can design a code that outlines the target code to be generated for that construct. for example, every three address code statement of the form $x := y + z$, where x, y and z are statically allocated, can be translated into the code sequence.

```
LOAD  y, R0    /* load y into Register R0 */   R0 = y
ADD   z, R0    /* add z to R0 */               R0 = R0 + z
MOV   R0, x    /* store R0 into x */           x = R0
```

The quality of the generated code is determined by its speed and size. i.e,

For example the sequence of three-address stmt

$a = b + c$

$d = a + e$

would be Translated into

```
1, LD  R0, b      // R0 = b
   ADD R0, R0, c  // R0 = R0 + c
   ST  a, R0      // a = R0
   LD  R0, a      // R0 = a
   ADD R0, R0, e  // R0 = R0 + e
   ST  d, R0      // d = R0.
```

```
2,  a = a + 1
    LD  R0, a      // R0 = a
    ADD R0, R0, #1 // R0 = R0 + 1
    ST  a, R0      // a = R0
```

Here, the fourth statement is Redundant since it loads a value that has just been stored, and so is the third if 'a' is not subsequently used.

Instruction used i.e, which instructions should be used in case there are multiple instructions that do the same job.

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 do not have enough of them to hold all values.

* values not held in Registers need to Reside in memory.

* Instructions involving Register operands are invariably shorter and faster than those involving operands in memory, so efficient utilisation of Registers is particularly important.

- the use of Registers is often ~~set~~ subdivided into two sub-problems.

1. Register allocation: , during which we select the set of variables that will Reside in Registers at each point in the program.
2. Register assignment: , during which we pick the specific Register that a variable will Reside in.

* store long life time values that are often used in Registers.

Even odd Register pairs:

certain machine Require register pairs (even odd register) for some operands and Results.

Ex! Some machines, integer multiplication and integer division involve register pairs. the multiplication instruction is of the form

M x, y

where x, the multiplicand, is the even register of an even/odd Register pair and y, the multiplier, is the odd register.

- the product occupies the entire even/odd Register pair.
- the division instruction is of the form.

D X, y

where the dividend occupies an even/odd register pair whose even register is X; the divisor is y. After division, the even Register holds the Remainder and the odd register the quotient.

Evaluation order: the order in which computations are performed can affect the efficiency of the Target code.

* the order in which the instructions will be executed,
- this increases performance of the code.

Example:

a+b-(c+d)*e

⇓

$t_1 := a+b$	MOV a, R0
$t_2 := c+d$	ADD b, R0
$t_3 := e*t_2$	mov R0, t_1
$t_4 := t_1 - t_3$	mov c, R1
	ADD d, R1
	mov e, R0
	mul R1, R0
	mov t_1, R1
	SUB R0, R1
	mov R1, t_4

Re-ordered instructions and code.

$t_2 := c+d$	Mov c, R0
$t_3 := e*t_2$	ADD d, R0
$t_1 := a+b$	mov e, R1
$t_4 := t_1 - t_3$	mul R0, R1
	mov a, R0
	ADD b, R0
	SUB R1, R0
	mov R0, t_4

when instructions are independents, their evaluation order can be changed to utilize and save instruction cost.

Peephole optimization:-

* while most production compilers produce good code through careful instruction selection and Registes allocation;

* A simple but effective technique for locally improving the target code is "peephole optimization", which is done by examining a sliding window of target instructions and Replacing instruction sequences within the peephole by a shorter or faster sequence.

* peephole optimization can also be applied directly after intermediate code generation to improve the intermediate Representation.

* the peephole is a small, sliding window on a program. the code in the peephole need not be contiguous, although some implementation do Require this.

characteristics of peephole optimation are:

* Redundant-instruction elimination
* Flow-of-control optimizations
* Algebraic simplifications
* use of machine idioms.

Redundant-instruction Elimination: If the target code contains the instruction sequence

```
MOV R, a
MOV a, R
```

We can Delete the second instruction if it is an unlabeled instruction, because first instruction ensure that the value

'a' is already in Register R. If it is labeled there is no guarantee that instruction ① will always be executed before instruction ②.

```
LD   a, R0
ST   R0, a
```

In a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of 'a' has already been loaded into Register R0.

Eliminating unreachable code:

* Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instructions immediately following an unconditional jump may be Removed.
* This operation can be Repeated to eliminate a sequence of instructions.

For example:

```
Sum=0
if(Sum)
printf("%d", Sum);
```

In above source code `if(Sum)` statement, it never get Executed, so, it is say that unReachable instruction and we can eliminate it.

Ex:-2

```
if a<b goto L1
goto L2
a=b
L1: a=b+2
L2: b=a+2
```

Flow-of-control optimisations: If we have jumps to jumps, then these unnecessary jumps can be eliminated in either intermediate code (or) Target code.

Ex-1 If we have the jump sequence as shown below.

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

then this can be Replaced by

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

if there are now no jumps to L1, then it may be possible to eliminate the statement L1: goto L2 provided by an unconditional jump.

Similarly, the sequence

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

can be Replaced by,

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

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence.

```
    goto L1
    ....
L1: if a<b goto L2
L3:
```

may be Replaced by the sequence

```
    goto L3
    ....
L3: if a<b goto L2
```

Ex: if x>y goto L4

```
L1: goto L2
L2: goto L3.
L3: goto L4
```

(unnecessary jumps)

```
    :
L9: goto L10
```

⇒ if x>y goto L6

Algebraic Simplification and Reduction in Strength:

* The algebraic identities that could be used to simplify DAG's. These algebraic identities can also be used by a peephole optimiser to eliminate three-address statements such as.

$$x = x + 0$$

(or)

$x = x * 1$ in the peephole.

Similarly, Reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones on the target machine. certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. for example, x^2 is invariably cheaper to implement as $x*x$ than as a call to an exponentiation routine.

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. for example some machines have auto-increment and auto-decrement addressing modes. these add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. these modes can also be used for implementation of statements like

$x := x + 1$.	$i = i - 1$
inc i	Dec i

Target Machine:-

Our target computer models a three-address machine with load and store operations, computation operations, jump operations and conditional jumps.

* the underlaying computer is a byte-addressable machine with n general-purpose Registers, $R_0, R_1, \ldots R_{n-1}$.

It has Two-address instruction of the form.

opcode source, destination.

in which opcode is an operation, and source and destination address data fields.

opcode-operations, it is typically

MOV (move content of source to destination)

ADD (add content of source to destination)

SUB (subtract content of sour. from destination)

in this, the Result will be in store destination field.

* the source and destination of instruction are specified by combining Registers and memory locations with address modes.

the address modes together with their Assembly-language forms and associated costs.

* Addressing modes Pnvolving Registers have zero additional cost, mde over, essentially each instruction having cost is 1 unit.

* we shall assume each target-language instruction has an associated cost. for simplicity, we take the cost f an Pnstruction to be one plus the costs Associated wilt the addressing modes f the operands. This cost corresponds to the length Pn words f the instruction.

Cost f Instruction:

1 + cost (source mode) + cost (Destination mode)

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	source to be a constant	1

It is Immediate Address mode.

Examples:

S.N	Instructions	operation	Cost
1	MOV R0, M	Store content of Register (R0) into memory location.	2
2	MOV R0, R1	Store content of Register (R0) into Register (R1)	1
3	MOV M, R0	Store content of (M) into Register R0	2
4	MOV 4(R0), M	Store contents (4+contents (R0)) into M	3
5	MOV #1, R0	Loads the constant 1 into Register R0	2

Some of the difficulties in generating code for this machine can be seen by considering what code to generate for a three-address statement of the form $a := b + c$. This statement can be implemented by many different instruction sequences.

	1+address mode cost	Instruction cost
1) MOV b, R0	1+1	2
ADD c, R0	1+1	2
MOV R0, a	1+1	2
	total cost	6
2). MOV b, a	1+2	3
ADD c, a	1+2	3
	total cost	6

Assuming R0, R1 and R2 contain the address of a, b and c Respectively.

3) MOV *R1, *R0	1+0	1
ADD *R2, *R0	1+0	1
	total cost	2
4) ADD R2, R1	1+0	1
MOV R1, a	1+1	2
	total cost	3

** In addl to generate good for this machine, we must utilize its addressing capabilities efficiently.

Examples:

1, the three-address statement $x = y - z$ can be implemented by the machine instructions.

```
LD  R1, y      // R1 = y
LD  R2, z      // R2 = z
SUB R1, R1, R2 // R1 = R1 - R2
ST  x, R1      // x = R1
```

2, Suppose 'a' is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of 'a' are indexed starting at 0. We may execute the three-address code $b = a[i]$ by the machine instruction.

```
LD  R1, i      //R1 = i
MUL R1, R1, 8  // R1 = R1 * 8
LD  R2, a(R1)  // R2 = contents (a + contents (R1))
ST  b, R2      // b = R2.
```

3, To implement a simple pointer indirection, such as the three-address code $\boxed{x = *p}$, we can use machine instruction

```
LD  R1, p      // R1 = p
LD  R2, 0(R1)  // R2 = contents (0 + contents (R1))
ST  x, R2      // x = R2
```

The Assignment through a pointer `*p = y` is similarly implemented in machine code by.

```
LD    R1, P          // R1 = p.
LD    R2, Y          // R2 = y
ST    0(R1), R2      // contents(0 + contents(R1)) = R2
```

4) Consider a conditional-Jump Three-address instruction like

`if x < y goto L`

the machine-code equivalent would be something like

```
LD    R1, X          // R1 = x
LD    R2, Y          // R2 = y
SUB   R1, R1, R2     // R1 = R1 - R2
BLTZ  R1, M          // if R1 < 0 Jump to M.
```

Here, M is the label that Represents the first machine instruction generated from the 3-address instruction that label L.

Note: BLTZ, it is a conditional Jump struction.

Bcond r, L, where r is a register, L is a label.

Exercises:

1, Generate code for the following 3-address statements assuming all variables are stored in memory locations.

a) $X = 1$ b) $x = a$ c) $x = a + 1$ d) $x = a + b$ e) $x = b * c$, $y = a + x$ } Two Statements

2,
```
y = *2
2 = 2+4
*p = y
p = p+4
```

3, 'a' and 'b' are arrays whose elements are 4-byte values.

1,
```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

2,
```
x = a[i]
y = b[i]
z = x*y.
```

3,
```
x = a[i]
y = b[x]
a[i] = y
```

Determine the costs of the following instruction sequences.

a)
```
LD  R0, y
LD  R1, z
ADD R0, R0, R1
ST  x, R0
```

b)
```
LD  R0, i
MUL R0, R0, 8
LD  R1, a(R0)
ST  b, R1
```

c)
```
LD  R0, c
LD  R1, i
MUL R1, R1, 8
ST  a(R1), R0
```

d)
```
LD  R0, p
LD  R1, 0(R0)
ST  x, R1
```

e)
```
LD  R0, p
LD  R1, x
ST  0(R0), R1
```

f)
```
LD  R0, x
LD  R1, y
SUB R0, R0, R1
BLTZ *R3, R0.
```