<u>Code Optimization</u>

Features of Optimization:

- Transformation of Computer program to maximize the efficiency of the executable program (ie, use fewer resources)
- Transformation done by compiler as well as programmer
- It is one of the important phases of compiler
- Rarely guarantee that the resulting code is best possible.
- Transformation must preserve the meaning of programs.

Code Optimization can be accomplished in two ways

- Machine Dependent Optimization :
  - It requires the knowledge of target machine.
  - An attempt to generate object code that will utilize the target machine's registers more efficiently is an example of machine dependent code optimization. (**We will not cover Machine dependent Optimization**)
- Machine Independent Optimization
  - Data Flow Optimization
    - Common Sub-expression elimination
    - Constant folding and combining
    - Strength reduction
    - Copy propagation
    - Dead Code Elimination
  - Loop Optimization
    - Loop-invariant code motion
    - Loop unrolling
    - Loop fusion

Common Sub-Expression Elimination

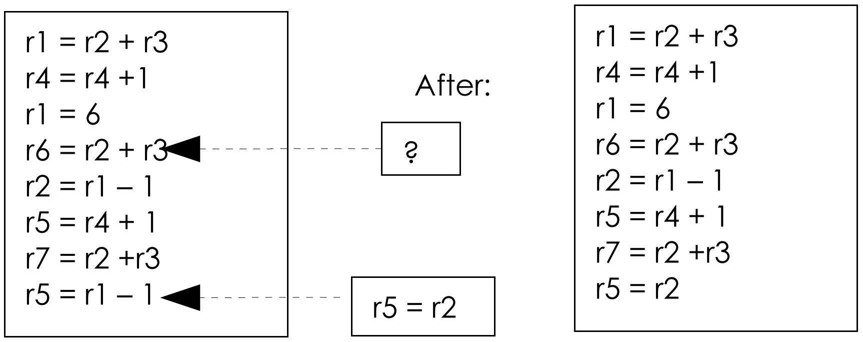
Goal: Eliminate re-computation of an expression

- More efficient code
- Resulting moves can get copy propagated

A subexpression E is called a “common” subexpression if E was previously computed and the values of E have not changed since the previous computation.

Example:

Before



```
r1 = r2 + r3
r4 = r4 +1
r1 = 6
r6 = r2 + r3
r2 = r1 – 1
r5 = r4 + 1
r7 = r2 +r3
r5 = r1 – 1
```

After:

```
r1 = r2 + r3
r4 = r4 +1
r1 = 6
r6 = r2 + r3
r2 = r1 – 1
r5 = r4 + 1
r7 = r2 +r3
r5 = r2
```

Constant Folding and Combining

Constant folding refers to the compiler pre- calculating the constant expression.

Goal: Eliminate unnecessary operation

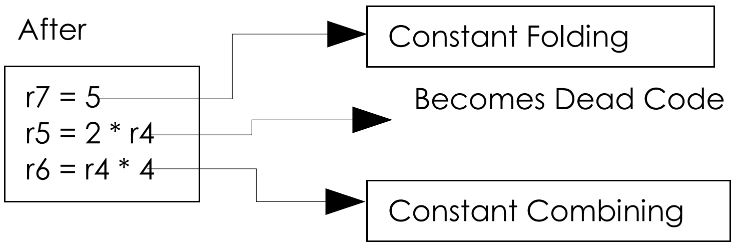
- First operation often becomes dead after constant combining.

Eg:

Before

```
r7 = 4 + 1
r5 = 2 * r4
r6 = r5 * 2
```

After



```
r7 = 5
r5 = 2 * r4
r6 = r4 * 4
```

Strength Reduction

Goal: Replace expensive operations with cheaper ones.

Eg:

Before

```
r5 = 2 * r4
r6 = r4 * 4
```

After,

```
r5 = r4 + r4
r6 = r4 << 2
```

Addition is cheaper than multiplication

Shifting is cheaper than multiplication

If x = a<<b then x = a*2^b

Eg, x = 50<<2 ➔ 200

X= 50 *2^2 = 200

Copy Propagation

Copy Propagation is the process of replacing the occurrences of targets of direct assignments with their values.

Goal: To eliminate useless moves.

Eg.

Before,

```
r2 = r1 + 1
r3 = r2
r4 = r3 * 4
```

After,

```
r2 = r1 +1
r4 = r2 * 4
```

You can use shift operator here

<u>Dead Code Elimination</u>

We try to eliminate those codes that are never used in a program.

We might have seen that some assignment statements or variables are given in program but they are never used .

Eg.

```
Begin
        Integer : I, J, A, Sum
        J = 5
        Sum = 0
        For I = 1 to 100 do
        Begin
                Sum = sum + i
        End
End
```

This program calculates, the sum of numbers from 1 to 100, but the statement J = 5 is not being used in any part of the program is called **Dead Code**

<u>Loop- Invariant Code Motion</u>

If an expression inside a loop doesn't change after the loop is entered (i,e it's invariant), calculate the value of the expression outside the loop and assign it to a temporary variable. Then use the temporary variable in the loop.

Eg,

```
for(i=0; i<max*10; i++)

        b[i] =b[i] + c * d;
```

Now code motion would yield,

```
int limit = max * 10;

float tmp = c * d;

for(i=0; i<limit; i++)

b[i] +=temp;
```

Loop Unrolling

Goal: We try to decrease the number of times the loop condition is tested.

Here we replicate the body of loop to reduce the required number of tests if the number of iterations is constant.

Eg,

```
i = 1

while(i<=100)

{

        X[i] = 0;

        i++;

}
```

In this code, the testing of **i<=100** will be performed 100 times.

In this code, if we replicate the body of loop, then the number of tests will be 50.

After replication,

```
i = 1
while(i<=100)
{

        X[i] = i;

        i++;

        X[i] = i;

        i++;

}
```

Loop Fusion

This technique merges the boidies of two loops if the two loops have the same number of iterations and they use the same indices.

Eg,

Before:

```
{
for(i = 0; i<10; i++)
        for(i = 0; i<10; i++)

                x[i,j] = 0;

        for(i = 0; i<10; i++)

                x[i,i] = 1;

}
```

After:

```
{
for(i = 0; i<10; i++)
        {

        for(j = 0; j<10; j++)

                x[i,j] = 0;

        x[i,i] = 1;

        }

}
```

Condition for making Loop Fusion Legal:

1. No quantity is computed by the second loop at the iteration **i** if it is computed by the first loop at iteration **j>=1**
2. If a value is computed by the first loop at iteration **j>=i**, then this value should not be used by second loop at iteration **i**.

"THE END"