Three address code using quadruples

General representation:

x = y op z

Where x, y, and z are IDs, constants, or temps (generated by the compiler)

Assignment

x = *y binop z*	(binop, y, z, x)
x = *uop y*	(uop, y, x)

Copy Statements

x = *y*	*(COPY, y, x)*
x = &*y*	*(COPY_FROM_REF, y, x)*
x = *y	*(COPY_FROM_DEREF, y, x)*
*x = *y*	*(COPY_TO_DEREF, y, x)*

Conditional Jumps

if x relop y goto L1 else goto L2	comparison and jump accordingly CMP(x, y) *(jop, L)*

Unconditional Jumps

goto L	*(JUMP, L)*

Register Operations

load R, A	(LOAD, R, A)
store A, *R*	(STORE, A, R)

Return Statements

return y, y is optional	(RETURN, y)

Procedure Calls

call function_name(arglist)

arglist is optional

*

binop: +, -, *, /, %, &&, ||

uop: +, -, !, ~

relop: ==, !=, <, <=, >, >=

L is a label

R is a register

A is an address

the result will always be the last parameter in the table, preceded by one or two arguments and an instruction

Example:

```
int add(int a, int b)
{
        return a + b;
}

int main()
{
        int x = 4;
        int y = 5;
        int z;

        z = x + y
```

```
    return 0;
}
```

IR:

.main()	FUNC main
x = 4	COPY 4 x
y = 5	COPY 5 y
R1 = x + y	ADD x y R1
z = R1	COPY R1 z

One of the main benefits of using this approach is that the IR is similar to x86 assembly but it still maintains the meaning of the original program. One of the major drawbacks of this approach is that the table is very explicit and can potentially take up a lot of space.