Bianca Cristina da Silva - 11711BCC023
Eliabe Vinicius Costa E Silva 11721BCC032
Felipe Alves Belisário - 11721BCC030
Guilherme H Scarpel - 11711BCC001
Gustavo Nunes de Oliveira - 11621BCC032

Componentes de Atribuição

Exemplo 1

C

```c
int main() {
  int a = 10;
}
```

LL

```
; ModuleID = 'comando-atribuicao-1.c'
source_filename = "comando-atribuicao-1.c"
target datalayout =
"e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:
32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
%a = alloca i32, align 4
store i32 10, i32* %a, align 4
ret i32 0
}
attributes #0 = { noinline nounwind optnone uwtable
"disable-tail-calls"="false" "frame-pointer"="all"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
```

```
"tune-cpu"="generic" "unsafe-fp-math"="false"
"use-soft-float"="false" }
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.0
(https://github.com/llvm/llvm-project.git
9a85643cd357e412cff69067bb5c4840e228c2ab)"}
```

Observações

- A instrução alloca é utilizada para alocar um determinado tipo de variável na pilha em um ponteiro nomeado. No exemplo “`%a = alloca i32, align 4`”, um inteiro de 32 bits é alocado no ponteiro denominado %a.
- Já a instrução store é utilizada para armazenar um determinado valor na memória. No exemplo “`store i32 10, i32* %a, align 4`”, um inteiro de 32 bits é armazenado com o valor 10.
- Finalmente, a instrução ret é usada para retornar um determinado valor. No exemplo “`ret i32 0`”, um inteiro de 32 bits com o valor 0 é retornado.

Exemplo 2

C

```
int main() {
  int a = 10;
  long int b = 20.0;
}
```

LL

```
; ModuleID = 'comando-atribuicao-2.c'
source_filename = "comando-atribuicao-2.c"
target datalayout =
"e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:
32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
```

```
%a = alloca i32, align 4
%b = alloca i64, align 8
store i32 10, i32* %a, align 4
store i64 20, i64* %b, align 8
ret i32 0
}
attributes #0 = { noinline nounwind optnone uwtable
"disable-tail-calls"="false" "frame-pointer"="all"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"tune-cpu"="generic" "unsafe-fp-math"="false"
"use-soft-float"="false" }
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.0
(https://github.com/llvm/llvm-project.git
9a85643cd357e412cff69067bb5c4840e228c2ab)"}
```

Observações

- No exemplo “`%b = alloca i64, align 8`”, a instrução alloca foi utilizada para alocar um inteiro de 64 bits no ponteiro denominado %b.
  Já no exemplo “`store i64 20, i64* %b, align 8`”, a instrução store foi utilizada para armazenar na memória um inteiro de 64 bits com valor 20.

Exemplo 3

C

```c
int main() {
  int a = 10;
  long int b = 20.0;
  float c = 30.0;
```

```
  double d = 40.0;
}
```

LL

```
; ModuleID = 'comando-atribuicao-3.c'
source_filename = "comando-atribuicao-3.c"
target datalayout =
"e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:
32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
%a = alloca i32, align 4
%b = alloca i64, align 8
%c = alloca float, align 4
%d = alloca double, align 8
store i32 10, i32* %a, align 4
store i64 20, i64* %b, align 8
store float 3.000000e+01, float* %c, align 4
store double 4.000000e+01, double* %d, align 8
ret i32 0
}
attributes #0 = { noinline nounwind optnone uwtable
"disable-tail-calls"="false" "frame-pointer"="all"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"tune-cpu"="generic" "unsafe-fp-math"="false"
"use-soft-float"="false" }
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.0
(https://github.com/llvm/llvm-project.git
9a85643cd357e412cff69067bb5c4840e228c2ab)"}
```

Observações

- No exemplo “`%a = alloca i32, align 4`” , o allign é utilizado para garantir que a instrução realizada ocupe os 4 bytes necessários para representar um inteiro de 32 bits. Já no exemplo “`%b = alloca i64, align 8`”, o allign é utilizado para assegurar que a instrução executada ocupe os 8 bytes necessários para representar um inteiro de 64 bits.

Componentes de Condição

Exemplo 1

C

```c
int main() {
  int a = 10;
  if(a == 10) {

  } else {

  }
}
```

LL

```
; ModuleID = 'comando-condicional-1.c'
source_filename = "comando-condicional-1.c"
target datalayout =
"e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:
32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
%retval = alloca i32, align 4
%a = alloca i32, align 4
store i32 0, i32* %retval, align 4
store i32 10, i32* %a, align 4
```

```
%0 = load i32, i32* %a, align 4
%cmp = icmp eq i32 %0, 10
br i1 %cmp, label %if.then, label %if.else
if.then:                                          ; preds =
%entry
br label %if.end
if.else:                                          ; preds =
%entry
br label %if.end
if.end:                                           ; preds =
%if.else, %if.then
%1 = load i32, i32* %retval, align 4
ret i32 %1
}
attributes #0 = { noinline nounwind optnone uwtable
"disable-tail-calls"="false" "frame-pointer"="all"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"tune-cpu"="generic" "unsafe-fp-math"="false"
"use-soft-float"="false" }
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.0
(https://github.com/llvm/llvm-project.git
9a85643cd357e412cff69067bb5c4840e228c2ab)"}
```

Observações

- A instrução load é utilizada para ler um valor da memória. No exemplo “`%0 = load i32, i32* %a, align 4`”, é utilizada para ler o valor de %a para %0.
- A instrução icmp retorna um valor booleano ou um vetor de booleanos baseado em uma dada comparação inteira. A sintaxe dessa instrução é icmp <cond> <ty> <op1>, <op2>. No exemplo “`%cmp = icmp eq i32 %0, 10`”, é utilizada para verificar se o valor em %0 é igual a 10.

Exemplo 2

C

```c
int main() {
  float a = 10.0;
  if (a < 5.00) {
  } else {

  }
}
```

LL

```
; ModuleID = 'comando-condicional-2.c'
source_filename = "comando-condicional-2.c"
target datalayout =
"e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:
32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
%retval = alloca i32, align 4
%a = alloca float, align 4
store i32 0, i32* %retval, align 4
store float 1.000000e+01, float* %a, align 4
%0 = load float, float* %a, align 4
%conv = fpext float %0 to double
%cmp = fcmp olt double %conv, 5.000000e+00
br i1 %cmp, label %if.then, label %if.else
if.then:                                          ; preds =
%entry
br label %if.end
if.else:                                          ; preds =
%entry
br label %if.end
```

```
if.end:                                           ; preds =
%if.else, %if.then
%1 = load i32, i32* %retval, align 4
ret i32 %1
}
attributes #0 = { noinline nounwind optnone uwtable
"disable-tail-calls"="false" "frame-pointer"="all"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"tune-cpu"="generic" "unsafe-fp-math"="false"
"use-soft-float"="false" }
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.0
(https://github.com/llvm/llvm-project.git
9a85643cd357e412cff69067bb5c4840e228c2ab)"}
```

Observações

- A instrução fpext estende um valor em ponto flutuante para um outro valor em ponto flutuante ainda maior. No exemplo “`%conv = fpext float %0 to double`”, é usado para converter um valor float para double.
- A instrução fcmp comporta-se de forma semelhante a icmp diferenciando-se no fato de realizar comparação de pontos flutuantes. A sintaxe dessa instrução é fcmp [fast-math flags]* <cond> <ty> <op1>, <op2>. No exemplo “`%cmp = fcmp olt double %conv, 5.000000e+00`”, é realizada uma comparação de double para verificar se o valor presente em %conv equivale a 5.00.

Exemplo 3

C

```
int main() {
```

```
 double a = 15.10;
 if(a >= 2.0) {
     return 1;
 } else {
     return 0;
 }
}
```

LL

```
; ModuleID = 'comando-condicional-3.c'
source_filename = "comando-condicional-3.c"
target datalayout =
"e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:
32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
%retval = alloca i32, align 4
%a = alloca double, align 8
store i32 0, i32* %retval, align 4
store double 1.510000e+01, double* %a, align 8
%0 = load double, double* %a, align 8
%cmp = fcmp oge double %0, 2.000000e+00
br i1 %cmp, label %if.then, label %if.else
if.then:                                          ; preds =
%entry
store i32 1, i32* %retval, align 4
br label %return
if.else:                                          ; preds =
%entry
store i32 0, i32* %retval, align 4
br label %return
return:                                           ; preds =
%if.else, %if.then
%1 = load i32, i32* %retval, align 4
ret i32 %1
}
attributes #0 = { noinline nounwind optnone uwtable
"disable-tail-calls"="false" "frame-pointer"="all"
```

```
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"tune-cpu"="generic" "unsafe-fp-math"="false"
"use-soft-float"="false" }
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.0
(https://github.com/llvm/llvm-project.git
9a85643cd357e412cff69067bb5c4840e228c2ab)"}
```

Observações

- O argumento oge é utilizado na instrução fcmp para realizar uma comparação, na qual o retorno é true se op1 ordenado e maior ou igual que op2 e false caso contrário. Nesse caso, a classificação ordenado significa que nenhum dos operandos é QNAN, ou seja, não são “quites not-a-number”. No exemplo “`%cmp = fcmp oge double %0, 2.000000e+00`”, foi utilizado para comparar se o valor contido em %0 é ordenado e maior ou igual a 2.0.

Exemplo 4

C

```c
int main() {
  int a = 1;
  if (a == 1) {}
  if (a != 1) {}
  if (a < 1) {}
  if (a > 1) {}
  if (a <= 1) {}
  if (a >= 1) {}
}
```

LL

```
; ModuleID = 'comando-condicional-4.c'
source_filename = "comando-condicional-4.c"
target datalayout =
"e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:
32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
%retval = alloca i32, align 4
%a = alloca i32, align 4
store i32 0, i32* %retval, align 4
store i32 1, i32* %a, align 4
%0 = load i32, i32* %a, align 4
%cmp = icmp eq i32 %0, 1
br i1 %cmp, label %if.then, label %if.end
if.then:                                          ; preds =
%entry
br label %if.end
if.end:                                           ; preds =
%if.then, %entry
%1 = load i32, i32* %a, align 4
%cmp1 = icmp ne i32 %1, 1
br i1 %cmp1, label %if.then2, label %if.end3
if.then2:                                         ; preds =
%if.end
br label %if.end3
if.end3:                                          ; preds =
%if.then2, %if.end
%2 = load i32, i32* %a, align 4
%cmp4 = icmp slt i32 %2, 1
br i1 %cmp4, label %if.then5, label %if.end6
if.then5:                                         ; preds =
%if.end3
br label %if.end6
if.end6:                                          ; preds =
%if.then5, %if.end3
%3 = load i32, i32* %a, align 4
%cmp7 = icmp sgt i32 %3, 1
```

```
br i1 %cmp7, label %if.then8, label %if.end9
if.then8:                                         ; preds =
%if.end6
br label %if.end9
if.end9:                                          ; preds =
%if.then8, %if.end6
%4 = load i32, i32* %a, align 4
%cmp10 = icmp sle i32 %4, 1
br i1 %cmp10, label %if.then11, label %if.end12
if.then11:                                        ; preds =
%if.end9
br label %if.end12
if.end12:                                         ; preds =
%if.then11, %if.end9
%5 = load i32, i32* %a, align 4
%cmp13 = icmp sge i32 %5, 1
br i1 %cmp13, label %if.then14, label %if.end15
if.then14:                                        ; preds =
%if.end12
br label %if.end15
if.end15:                                         ; preds =
%if.then14, %if.end12
%6 = load i32, i32* %retval, align 4
ret i32 %6
}
attributes #0 = { noinline nounwind optnone uwtable
"disable-tail-calls"="false" "frame-pointer"="all"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"tune-cpu"="generic" "unsafe-fp-math"="false"
"use-soft-float"="false" }
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.0
(https://github.com/llvm/llvm-project.git
9a85643cd357e412cff69067bb5c4840e228c2ab)"}
```

Observações

- O argumento eq é utilizado na instrução icmp para realizar uma comparação, na qual o retorno é true se os operandos forem iguais e false caso contrário. No exemplo “`%cmp = icmp eq i32 %0, 1`”, o argumento foi utilizado para comparar se o valor presente em %0 é igual a 1.
- Já o argumento ne é utilizado na instrução icmp para realizar uma comparação, na qual o retorno é true se os operandos forem diferentes e false caso contrário. No exemplo “`%cmp1 = icmp ne i32 %1, 1`”, o argumento foi utilizado para comparar se o valor presente em %1 é diferente de 1.
- Por outro lado, o argumento slt é utilizado na instrução icmp para realizar uma comparação, na qual o retorno é true caso op1 < op2 e false caso contrário. No exemplo “`%cmp4 = icmp slt i32 %2, 1`”, o argumento foi utilizado para comparar se o valor presente em %2 é menor que 1.
- Já o argumento sgt é utilizado na instrução icmp para realizar uma comparação, na qual o retorno é true se op1 > op2 e false caso contrário. No exemplo “`%cmp7 = icmp sgt i32 %3, 1`”, foi utilizado para comparar se o valor presente no ponteiro %3 é maior que 1.
- Já o argumento sle é utilizado na instrução icmp para realizar uma comparação, na qual o retorno é true se op1 <= op2 e false caso contrário. No exemplo “`%cmp10 = icmp sle i32 %4, 1`”, foi utilizado para comparar se o valor presente no ponteiro %4 é menor ou igual a 1.
- Finalmente, o argumento sge é utilizado na instrução icmp para realizar uma comparação, na qual o retorno é true se op1 >= op2 e false caso contrário. No exemplo “`%cmp13 = icmp sge i32 %5, 1`”, foi utilizado para comparar se o valor presente no ponteiro %5 é maior ou igual a 1.

Comentários

C

```c
int main() {
   //comentario 1
   //comentario 2
   return 0;
}
```

LL

```
; ModuleID = 'comentarios.c'
source_filename = "comentarios.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:
32:64-S128"
target triple = "x86_64-pc-windows-msvc19.28.29333"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 ret i32 0
}
attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }
!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}
!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Observações

- Os comentários são ignorados, logo apenas o resto do código é compilado.

Comando de seleção

C

```c
int main() {
  int a = 2;
  switch (a)
  {
     case 1:
        a= a+10;
        break;
     case 2:
        a= a+20;
        break;
     default:
        a= a+30;
        break;
  }
  return 0;
}
```

LL

```
; ModuleID = 'switch.c'
source_filename = "switch.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:
32:64-S128"
target triple = "x86_64-pc-windows-msvc19.28.29333"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 2, i32* %2, align 4
 %3 = load i32, i32* %2, align 4
 switch i32 %3, label %10 [
```

```
    i32 1, label %4
    i32 2, label %7
  ]
4:                                                ; preds =
%0
  %5 = load i32, i32* %2, align 4
  %6 = add nsw i32 %5, 10
  store i32 %6, i32* %2, align 4
  br label %13
7:                                                ; preds =
%0
  %8 = load i32, i32* %2, align 4
  %9 = add nsw i32 %8, 20
  store i32 %9, i32* %2, align 4
  br label %13
10:                                               ; preds =
%0
  %11 = load i32, i32* %2, align 4
  %12 = add nsw i32 %11, 30
  store i32 %12, i32* %2, align 4
  br label %13
13:                                               ; preds =
%10, %7, %4
  ret i32 0
}
attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="true" "stack-protector-buffer-size"="8"
"target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }
!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}
!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Observações

- São criados criados 4 labels, uma para cada possibilidade de switch e uma para a saída da função switch, a linha `switch i32 %3, label %10` é a entrada da função switch definindo a label 10 como seleção default.
- As linhas `i32 1, label %4` e `i32 2, label %7` são os dois casos possíveis e seus valores de seleção.
- Após a execução dos casos, é chamado o label 13 pela linha `br label %13` que é o equivalente ao break do código em c, que volta a execução principal.

Estruturas de Repetição

Exemplo 1

C

```c
int main() {
    int i;

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

    }
}
```

LL

```
 ; ModuleID = 'estruturas-repeticao-1.1.c'
source_filename = "estruturas-repeticao-1.1.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
target triple = "x86_64-pc-windows-msvc19.16.27034"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
```

```
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  br label %3

3:                                                ; preds
= %7, %0
  %4 = load i32, i32* %2, align 4
  %5 = icmp slt i32 %4, 10
  br i1 %5, label %6, label %10

6:                                                ; preds
= %3
  br label %7

7:                                                ; preds
= %6
  %8 = load i32, i32* %2, align 4
  %9 = add nsw i32 %8, 1
  store i32 %9, i32* %2, align 4
  br label %3

10:                                               ; preds
= %3
  %11 = load i32, i32* %1, align 4
  ret i32 %11
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
```

```
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Observações

- A estrutura de repetições do código se transforma em uma estrutura de laços que são chamados de acordo com a comparação inicial dada que no caso do código em C acima seria a condição de "`i < 10`".
- Após as instruções de declaração da variável `i` o código prossegue para o laço inicial denominado "3", onde é feita a comparação inicial na instrução "`%5 = icmp slt i32 %4, 10`", caso o valor de i não seja o valor de parada ele vai para o laço "6" e caso seja vai para o laço ¨10" diretamente (onde o código é finalizado) através da decisão da instrução "`br i1 %5, label %6, label %10`".
- Como o for está vazio após cair no laço "6" é redirecionado para o laço "7" onde é feito o incremento da variável em 1 através da instrução "`%9 = add nsw i32 %8, 1`" e dela voltando para o laço "3" até que a condição de parada seja cumprida.

Exemplo 2

C

```
int main(){
    int i = 0;

    while(i < 10){
       i++;

        int j;
```

```
            for(j = 0; j < 20; j++){

            }
        }
    }
```

LL

```
; ModuleID = 'estruturas-repeticao-1.2.c'
source_filename = "estruturas-repeticao-1.2.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
target triple = "x86_64-pc-windows-msvc19.16.27034"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  br label %4

4:                                                ; preds
= %17, %0
  %5 = load i32, i32* %2, align 4
  %6 = icmp slt i32 %5, 10
  br i1 %6, label %7, label %18

7:                                                ; preds
= %4
  %8 = load i32, i32* %2, align 4
  %9 = add nsw i32 %8, 1
  store i32 %9, i32* %2, align 4
  store i32 0, i32* %3, align 4
```

```
  br label %10

10:                                               ; preds
= %14, %7
  %11 = load i32, i32* %3, align 4
  %12 = icmp slt i32 %11, 20
  br i1 %12, label %13, label %17

13:                                               ; preds
= %10
  br label %14

14:                                               ; preds
= %13
  %15 = load i32, i32* %3, align 4
  %16 = add nsw i32 %15, 1
  store i32 %16, i32* %3, align 4
  br label %10

17:                                               ; preds
= %10
  br label %4

18:                                               ; preds
= %4
  %19 = load i32, i32* %1, align 4
  ret i32 %19
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
```

```
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Observações

- Encapsulando o for do exemplo anterior dentro de um while a ideia do código IR não muda, apenas são adicionados mais laços ao redor do que já se tinha. O laço “4” é o início do primeiro while em que já se é feita a verificação inicial da variável e redireciona ou para o fim do código (laço “16”) ou para dentro da estrutura de repetição correspondente para começar o for (laço “7”), a partir disso o for faz a mesma coisa descrita do exemplo anterior.

Exemplo 3

C

```
int main(){
   int i = 0;
   do{
       i++;
   } while(i < 10);
}
```

LL

```
; ModuleID = 'estruturas-repeticao-1.3.c'
source_filename = "estruturas-repeticao-1.3.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
target triple = "x86_64-pc-windows-msvc19.16.27034"
; Function Attrs: noinline nounwind optnone uwtable
```

```
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 store i32 0, i32* %2, align 4
 br label %3
3:                                                ; preds
= %6, %0
 %4 = load i32, i32* %2, align 4
 %5 = add nsw i32 %4, 1
 store i32 %5, i32* %2, align 4
 br label %6
6:                                                ; preds
= %3
 %7 = load i32, i32* %2, align 4
 %8 = icmp slt i32 %7, 10
 br i1 %8, label %3, label %9
9:                                                ; preds
= %6
 %10 = load i32, i32* %1, align 4
 ret i32 %10
}
attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }
!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}
!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Observações

- Novamente a estrutura de call de laços até se cumprir a condição de parada ainda continua, porém a única diferença para os exemplos anteriores é que a própria característica específica do do{} while é de que logo no primeiro laço chamado pela main() no início da estrutura o valor da variável já é incrementado (laço “3”) e só no próximo chamado (laço “6”) que é feita a comparação para descobrir se chegou ou não na condição de parada.

Tipos de Dados

Um novo tipo de dado é criado com a seguinte sintaxe:

%X = alloca tipo, align n

Onde X representa o ID da variável na tabela de símbolos, o tipo corresponde ao tipo de dado da variável e o n representa a quantidade do de bytes que serão alocados. Os tipos de dados da linguagem C e seus correspondentes em IR são apresentados abaixo:

Char

O tipo char pode ser representado da seguinte forma:

	alloca	align
char	i8	1
unsigned char	i8	1

Int

O tipo int pode ser representado da seguinte forma:

	alloca	align
int	i32	4

unsigned int	i32	4

Short

O tipo short pode ser representado da seguinte forma:

	alloca	align
short	i16	2
unsigned short	i16	2

Long

O tipo long pode ser representado da seguinte forma:

	alloca	align
long	i32	4
unsigned long	i32	4

Float

O tipo float pode ser representado da seguinte forma:

	alloca	align
float	float	4

Double

O tipo double pode ser representado da seguinte forma:

	alloca	align
double	double	8

long double	double	8

Ponteiros

Os ponteiros são representados da seguinte forma:

%X = alloca tipo*, align 8

Onde o X é o ID da variável na tabela de símbolos, o tipo é qualquer um dos informados acima (char, int, short, long, float e double) seguido de *. E, por fim, são alocados 8 bytes independente do tipo do dado.

Vetores

Os vetores, assim como os ponteiros, são representados de modo diferente dos tipos normais:

%X = alloca [Y x tipo], align n

Onde o X é o ID da variável na tabela de símbolos, o Y representa o número de elementos contidos no vetor (ex. [3 x i32] corresponde a int[3]), o tipo é qualquer um dos informados acima (char, int, short, long, float e double). E, por fim, n representa a quantidade de bytes referentes ao tipo do dado.

Códigos

C

```c
int main(){
    char a;
    unsigned char b;

    int c;
    unsigned int d;

    short e;
    unsigned short f;

    long g;
    unsigned h;
```

```
    float i;
    double j;
    long double k;

    char *l;
    int *m;
    float *n;

    int v[3];
    float x[3];
    double y[3];
    char z[3];
}
```

LL

```
; ModuleID = 'dataTypes.c'
source_filename = "dataTypes.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
target triple = "x86_64-pc-windows-msvc19.11.0"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i8, align 1
 %2 = alloca i8, align 1
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 %5 = alloca i16, align 2
 %6 = alloca i16, align 2
 %7 = alloca i32, align 4
 %8 = alloca i32, align 4
 %9 = alloca float, align 4
 %10 = alloca double, align 8
 %11 = alloca double, align 8
 %12 = alloca i8*, align 8
 %13 = alloca i32*, align 8
```

```
  %14 = alloca float*, align 8
  %15 = alloca [3 x i32], align 4
  %16 = alloca [3 x float], align 4
  %17 = alloca [3 x double], align 16
  %18 = alloca [3 x i8], align 1
  ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Expressões

Aritméticas

Como os tipos char, short e long são variações do int, é possível utilizar os comandos abaixo relacionados ao int apenas trocando o tipo (i32) para o tipo correspondente (i8 e i16). Além disso, os valores x e y correspondem às variáveis a serem adicionadas à expressão.

Adição

Os dados podem ser adicionados utilizando os comandos abaixo:

int	add nsw i32 x, y
float	fadd float x, y
double	fadd double x, y

Subtração

Os dados podem ser subtraídos utilizando os comandos abaixo:

int	sub nsw i32 x, y
float	fsub float x, y
double	fsub double x, y

Multiplicação

Os dados podem ser multiplicados utilizando os comandos abaixo:

int	mul nsw type x, y
float	fmul float x, y
double	fmul double x, y

Divisão

Os dados podem ser divididos utilizando os comandos abaixo:

int	sdiv i32 x, y
float	fdiv float x, y
double	fdiv double x, y

Módulo

É possível obter o módulo de um int utilizando o comando abaixo:

int	srem i32 x, y

Relacionais

Assim como anteriormente, os valores x e y correspondem às variáveis a serem adicionadas à expressão.

	int	float	double
>	icmp sgt i32 x, y	fcmp ogt float x, y	fcmp ogt double x, y
<	icmp slt i32 x, y	fcmp olt float x, y	fcmp olt double x, y
>=	icmp sge i32 x, y	fcmp oge float x, y	fcmp oge double x, y
<=	icmp sle i32 x, y	fcmp ole float x, y	fcmp ole double x, y
==	icmp eq i32 x, y	fcmp oeq float x, y	fcmp oeq double x, y
!=	icmp ne i32 x, y	fcmp une float x, y	fcmp une double x, y

Lógicas

Ambos os comandos lógicos do c (and e or) utilizam o comando de diferença (!=). A partir dele, é criada uma estrutura que representa os comandos lógicos como pode ser visto a seguir.

And (&&)

É criado uma estrutura semelhante a um encadeamento de if's. O primeiro argumento aplicado a função and é aplicado na função de diferença junto com um 0. Como o c não possui o tipo primitivo booleano, é utilizado o 0 para representar falso e 1 para representar verdadeiro. Assim, a função diferença tem o papel de informar se o primeiro argumento é verdadeiro ou não. Se o argumento for falso, sabemos que o resultado da função and será falso. Porém, se o primeiro argumento for verdadeiro, é preciso testar o segundo argumento. O processo se repete, agora com o segundo argumento. Se ele for verdadeiro, a função and retornará verdadeiro, caso contrário retornará falso.

Podemos representar esse processo na função abaixo:

```
função and(x, y){
        resultado = falso

        se (x é verdadeiro){
                se (y é verdadeiro){
                        resultado = verdadeiro
                }
        }

        retorna resultado
}
```

Or (||)

A função or se assemelha a função and. Ela também é representada por um encadeamento de if's. Como o resultado da função or é verdadeiro se algum dos dois argumentos forem verdadeiros, ela verifica o primeiro argumento utilizando a função de diferença com 0. Se o resultado da função é verdadeiro, o segundo argumento não precisa ser testado e o resultado da função or é verdadeiro. Porém, se o resultado da função diferença for falso, é preciso testar o segundo argumento. Se ele for verdadeiro, o resultado da função or será verdadeiro, caso contrário será falso.

Podemos representar esse processo com a função abaixo:

```
função or (x, y){
        resultado = falso

        se (x é verdadeiro) resultado = verdadeiro
        senão se (y é verdadeiro) resultado = verdadeiro

        retorna resultado
}
```

Condicionais

If Simplificado

O comando de if simplificado é representado da seguinte forma:

```
select i1 resultadoCondicao, tipoX x, tipoY y
```

Onde resultadoCondição é o resultado de uma função relacional (maior, menor, diferente, etc) ou lógica (and ou or), o tipoX é o tipo da variável x e o tipoY é o tipo da variável y. Por exemplo, supondo que A = 2, teremos o seguinte if simplicado e seu IR correspondente:

A>1? 0 : 1	select i1 1, i32 0, i32 1

Como A é maior que 1, o resultado dessa função é 1 (verdadeiro). Então o resultadoCondicao é igual a 1. Como ambos 0 e 1 são inteiros, o tipo dos dois será i32.

Códigos

Aritméticas

C

```c
int main(){
   int a=1, b=2;
   int c = a+b;
   int d = a-b;
   int e = a*b;
   int f = a/b;
   int g = a%b;

   float h=1, i=2;
   float j = h+i;
   float k = h-i;
   float l = h*i;
   float m = h/i;

   float n=1, o=2;
   float p = n+o;
   float q = n-o;
   float r = n*o;
   float s = n/o;
}
```

LL

```
; ModuleID = 'arithmetic.c'
source_filename = "arithmetic.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
```

```
target triple = "x86_64-pc-windows-msvc19.11.0"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 %5 = alloca i32, align 4
 %6 = alloca i32, align 4
 %7 = alloca i32, align 4
 %8 = alloca float, align 4
 %9 = alloca float, align 4
 %10 = alloca float, align 4
 %11 = alloca float, align 4
 %12 = alloca float, align 4
 %13 = alloca float, align 4
 %14 = alloca float, align 4
 %15 = alloca float, align 4
 %16 = alloca float, align 4
 %17 = alloca float, align 4
 %18 = alloca float, align 4
 %19 = alloca float, align 4
 store i32 1, i32* %1, align 4
 store i32 2, i32* %2, align 4
 %20 = load i32, i32* %1, align 4
 %21 = load i32, i32* %2, align 4
 %22 = add nsw i32 %20, %21
 store i32 %22, i32* %3, align 4
 %23 = load i32, i32* %1, align 4
 %24 = load i32, i32* %2, align 4
 %25 = sub nsw i32 %23, %24
 store i32 %25, i32* %4, align 4
 %26 = load i32, i32* %1, align 4
 %27 = load i32, i32* %2, align 4
 %28 = mul nsw i32 %26, %27
 store i32 %28, i32* %5, align 4
 %29 = load i32, i32* %1, align 4
```

```
%30 = load i32, i32* %2, align 4
%31 = sdiv i32 %29, %30
store i32 %31, i32* %6, align 4
%32 = load i32, i32* %1, align 4
%33 = load i32, i32* %2, align 4
%34 = srem i32 %32, %33
store i32 %34, i32* %7, align 4
store float 1.000000e+00, float* %8, align 4
store float 2.000000e+00, float* %9, align 4
%35 = load float, float* %8, align 4
%36 = load float, float* %9, align 4
%37 = fadd float %35, %36
store float %37, float* %10, align 4
%38 = load float, float* %8, align 4
%39 = load float, float* %9, align 4
%40 = fsub float %38, %39
store float %40, float* %11, align 4
%41 = load float, float* %8, align 4
%42 = load float, float* %9, align 4
%43 = fmul float %41, %42
store float %43, float* %12, align 4
%44 = load float, float* %8, align 4
%45 = load float, float* %9, align 4
%46 = fdiv float %44, %45
store float %46, float* %13, align 4
store float 1.000000e+00, float* %14, align 4
store float 2.000000e+00, float* %15, align 4
%47 = load float, float* %14, align 4
%48 = load float, float* %15, align 4
%49 = fadd float %47, %48
store float %49, float* %16, align 4
%50 = load float, float* %14, align 4
%51 = load float, float* %15, align 4
%52 = fsub float %50, %51
store float %52, float* %17, align 4
%53 = load float, float* %14, align 4
%54 = load float, float* %15, align 4
%55 = fmul float %53, %54
```

```
 store float %55, float* %18, align 4
 %56 = load float, float* %14, align 4
 %57 = load float, float* %15, align 4
 %58 = fdiv float %56, %57
 store float %58, float* %19, align 4
 ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Relacionais

C

```
int main(){
   int a = 1, b = 2;
   float c = 1, d = 2;
   double e = 1, f = 2;

   if(a>b){}
   if(a<b){}
   if(a>=b){}
```

```
    if(a<=b){}
    if(a==b){}
    if(a!=b){}

    if(c>d){}
    if(c<d){}
    if(c>=d){}
    if(c<=d){}
    if(c==d){}
    if(c!=d){}

    if(e>f){}
    if(e<f){}
    if(e>=f){}
    if(e<=f){}
    if(e==f){}
    if(e!=f){}
}
```

LL

```
; ModuleID = 'relational.c'
source_filename = "relational.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
target triple = "x86_64-pc-windows-msvc19.11.0"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca float, align 4
 %5 = alloca float, align 4
 %6 = alloca double, align 8
 %7 = alloca double, align 8
 store i32 0, i32* %1, align 4
 store i32 1, i32* %2, align 4
```

```
 store i32 2, i32* %3, align 4
 store float 1.000000e+00, float* %4, align 4
 store float 2.000000e+00, float* %5, align 4
 store double 1.000000e+00, double* %6, align 8
 store double 2.000000e+00, double* %7, align 8
 %8 = load i32, i32* %2, align 4
 %9 = load i32, i32* %3, align 4
 %10 = icmp sgt i32 %8, %9
 br i1 %10, label %11, label %12

11:                                               ; preds =
%0
 br label %12

12:                                               ; preds =
%11, %0
 %13 = load i32, i32* %2, align 4
 %14 = load i32, i32* %3, align 4
 %15 = icmp slt i32 %13, %14
 br i1 %15, label %16, label %17

16:                                               ; preds =
%12
 br label %17

17:                                               ; preds =
%16, %12
 %18 = load i32, i32* %2, align 4
 %19 = load i32, i32* %3, align 4
 %20 = icmp sge i32 %18, %19
 br i1 %20, label %21, label %22

21:                                               ; preds =
%17
 br label %22

22:                                               ; preds =
%21, %17
```

```
 %23 = load i32, i32* %2, align 4
 %24 = load i32, i32* %3, align 4
 %25 = icmp sle i32 %23, %24
 br i1 %25, label %26, label %27

26:                                               ; preds =
%22
 br label %27

27:                                               ; preds =
%26, %22
 %28 = load i32, i32* %2, align 4
 %29 = load i32, i32* %3, align 4
 %30 = icmp eq i32 %28, %29
 br i1 %30, label %31, label %32

31:                                               ; preds =
%27
 br label %32

32:                                               ; preds =
%31, %27
 %33 = load i32, i32* %2, align 4
 %34 = load i32, i32* %3, align 4
 %35 = icmp ne i32 %33, %34
 br i1 %35, label %36, label %37

36:                                               ; preds =
%32
 br label %37

37:                                               ; preds =
%36, %32
 %38 = load float, float* %4, align 4
 %39 = load float, float* %5, align 4
 %40 = fcmp ogt float %38, %39
 br i1 %40, label %41, label %42
```

```
41:                                               ; preds =
%37
 br label %42

42:                                               ; preds =
%41, %37
 %43 = load float, float* %4, align 4
 %44 = load float, float* %5, align 4
 %45 = fcmp olt float %43, %44
 br i1 %45, label %46, label %47

46:                                               ; preds =
%42
 br label %47

47:                                               ; preds =
%46, %42
 %48 = load float, float* %4, align 4
 %49 = load float, float* %5, align 4
 %50 = fcmp oge float %48, %49
 br i1 %50, label %51, label %52

51:                                               ; preds =
%47
 br label %52

52:                                               ; preds =
%51, %47
 %53 = load float, float* %4, align 4
 %54 = load float, float* %5, align 4
 %55 = fcmp ole float %53, %54
 br i1 %55, label %56, label %57

56:                                               ; preds =
%52
 br label %57
```

```
57:                                               ; preds =
%56, %52
 %58 = load float, float* %4, align 4
 %59 = load float, float* %5, align 4
 %60 = fcmp oeq float %58, %59
 br i1 %60, label %61, label %62

61:                                               ; preds =
%57
 br label %62

62:                                               ; preds =
%61, %57
 %63 = load float, float* %4, align 4
 %64 = load float, float* %5, align 4
 %65 = fcmp une float %63, %64
 br i1 %65, label %66, label %67

66:                                               ; preds =
%62
 br label %67

67:                                               ; preds =
%66, %62
 %68 = load double, double* %6, align 8
 %69 = load double, double* %7, align 8
 %70 = fcmp ogt double %68, %69
 br i1 %70, label %71, label %72

71:                                               ; preds =
%67
 br label %72

72:                                               ; preds =
%71, %67
 %73 = load double, double* %6, align 8
 %74 = load double, double* %7, align 8
 %75 = fcmp olt double %73, %74
```

```
 br i1 %75, label %76, label %77

76:                                               ; preds =
%72
 br label %77

77:                                               ; preds =
%76, %72
 %78 = load double, double* %6, align 8
 %79 = load double, double* %7, align 8
 %80 = fcmp oge double %78, %79
 br i1 %80, label %81, label %82

81:                                               ; preds =
%77
 br label %82

82:                                               ; preds =
%81, %77
 %83 = load double, double* %6, align 8
 %84 = load double, double* %7, align 8
 %85 = fcmp ole double %83, %84
 br i1 %85, label %86, label %87

86:                                               ; preds =
%82
 br label %87

87:                                               ; preds =
%86, %82
 %88 = load double, double* %6, align 8
 %89 = load double, double* %7, align 8
 %90 = fcmp oeq double %88, %89
 br i1 %90, label %91, label %92

91:                                               ; preds =
%87
 br label %92
```

```
92:                                               ; preds =
%91, %87
 %93 = load double, double* %6, align 8
 %94 = load double, double* %7, align 8
 %95 = fcmp une double %93, %94
 br i1 %95, label %96, label %97

96:                                               ; preds =
%92
 br label %97

97:                                               ; preds =
%96, %92
 %98 = load i32, i32* %1, align 4
 ret i32 %98
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Lógicas

C

```c
int main(){
   int a = 1, b = 2;
   float c = 1, d = 2;
   double e = 1, f = 2;

   if(a || b){}
   if(a && b){}

   if(c || d){}
   if(c && d){}

   if(e || f){}
   if(e && f){}
}
```

LL

```
; ModuleID = 'logicas.c'
source_filename = "logicas.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
target triple = "x86_64-pc-windows-msvc19.11.0"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca float, align 4
 %5 = alloca float, align 4
 %6 = alloca double, align 8
 %7 = alloca double, align 8
 store i32 0, i32* %1, align 4
 store i32 1, i32* %2, align 4
 store i32 2, i32* %3, align 4
```

```
 store float 1.000000e+00, float* %4, align 4
 store float 2.000000e+00, float* %5, align 4
 store double 1.000000e+00, double* %6, align 8
 store double 2.000000e+00, double* %7, align 8
 %8 = load i32, i32* %2, align 4
 %9 = icmp ne i32 %8, 0
 br i1 %9, label %13, label %10

10:                                               ; preds =
%0
 %11 = load i32, i32* %3, align 4
 %12 = icmp ne i32 %11, 0
 br i1 %12, label %13, label %14

13:                                               ; preds =
%10, %0
 br label %14

14:                                               ; preds =
%13, %10
 %15 = load i32, i32* %2, align 4
 %16 = icmp ne i32 %15, 0
 br i1 %16, label %17, label %21

17:                                               ; preds =
%14
 %18 = load i32, i32* %3, align 4
 %19 = icmp ne i32 %18, 0
 br i1 %19, label %20, label %21

20:                                               ; preds =
%17
 br label %21

21:                                               ; preds =
%20, %17, %14
 %22 = load float, float* %4, align 4
 %23 = fcmp une float %22, 0.000000e+00
```

```
 br i1 %23, label %27, label %24

24:                                               ; preds =
%21
 %25 = load float, float* %5, align 4
 %26 = fcmp une float %25, 0.000000e+00
 br i1 %26, label %27, label %28

27:                                               ; preds =
%24, %21
 br label %28

28:                                               ; preds =
%27, %24
 %29 = load float, float* %4, align 4
 %30 = fcmp une float %29, 0.000000e+00
 br i1 %30, label %31, label %35

31:                                               ; preds =
%28
 %32 = load float, float* %5, align 4
 %33 = fcmp une float %32, 0.000000e+00
 br i1 %33, label %34, label %35

34:                                               ; preds =
%31
 br label %35

35:                                               ; preds =
%34, %31, %28
 %36 = load double, double* %6, align 8
 %37 = fcmp une double %36, 0.000000e+00
 br i1 %37, label %41, label %38

38:                                               ; preds =
%35
 %39 = load double, double* %7, align 8
 %40 = fcmp une double %39, 0.000000e+00
```

```
 br i1 %40, label %41, label %42

41:                                               ; preds =
%38, %35
 br label %42

42:                                               ; preds =
%41, %38
 %43 = load double, double* %6, align 8
 %44 = fcmp une double %43, 0.000000e+00
 br i1 %44, label %45, label %49

45:                                               ; preds =
%42
 %46 = load double, double* %7, align 8
 %47 = fcmp une double %46, 0.000000e+00
 br i1 %47, label %48, label %49

48:                                               ; preds =
%45
 br label %49

49:                                               ; preds =
%48, %45, %42
 %50 = load i32, i32* %1, align 4
 ret i32 %50
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
```

```
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Relacionais

C

```
int main(){
   int a = 1, b = 2;
   float c = 1, d = 2;
   double e = 1, f = 2;

   if(a>b){}
   if(a<b){}
   if(a>=b){}
   if(a<=b){}
   if(a==b){}
   if(a!=b){}

   if(c>d){}
   if(c<d){}
   if(c>=d){}
   if(c<=d){}
   if(c==d){}
   if(c!=d){}

   if(e>f){}
   if(e<f){}
   if(e>=f){}
   if(e<=f){}
   if(e==f){}
   if(e!=f){}
```

```
}
```

LL

```
; ModuleID = 'relational.c'
source_filename = "relational.c"
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
target triple = "x86_64-pc-windows-msvc19.11.0"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 %4 = alloca float, align 4
 %5 = alloca float, align 4
 %6 = alloca double, align 8
 %7 = alloca double, align 8
 store i32 0, i32* %1, align 4
 store i32 1, i32* %2, align 4
 store i32 2, i32* %3, align 4
 store float 1.000000e+00, float* %4, align 4
 store float 2.000000e+00, float* %5, align 4
 store double 1.000000e+00, double* %6, align 8
 store double 2.000000e+00, double* %7, align 8
 %8 = load i32, i32* %2, align 4
 %9 = load i32, i32* %3, align 4
 %10 = icmp sgt i32 %8, %9
 br i1 %10, label %11, label %12

11:                                               ; preds =
%0
 br label %12

12:                                               ; preds =
%11, %0
 %13 = load i32, i32* %2, align 4
```

```
 %14 = load i32, i32* %3, align 4
 %15 = icmp slt i32 %13, %14
 br i1 %15, label %16, label %17

16:                                               ; preds =
%12
 br label %17

17:                                               ; preds =
%16, %12
 %18 = load i32, i32* %2, align 4
 %19 = load i32, i32* %3, align 4
 %20 = icmp sge i32 %18, %19
 br i1 %20, label %21, label %22

21:                                               ; preds =
%17
 br label %22

22:                                               ; preds =
%21, %17
 %23 = load i32, i32* %2, align 4
 %24 = load i32, i32* %3, align 4
 %25 = icmp sle i32 %23, %24
 br i1 %25, label %26, label %27

26:                                               ; preds =
%22
 br label %27

27:                                               ; preds =
%26, %22
 %28 = load i32, i32* %2, align 4
 %29 = load i32, i32* %3, align 4
 %30 = icmp eq i32 %28, %29
 br i1 %30, label %31, label %32
```

```
31:                                               ; preds =
%27
 br label %32

32:                                               ; preds =
%31, %27
 %33 = load i32, i32* %2, align 4
 %34 = load i32, i32* %3, align 4
 %35 = icmp ne i32 %33, %34
 br i1 %35, label %36, label %37

36:                                               ; preds =
%32
 br label %37

37:                                               ; preds =
%36, %32
 %38 = load float, float* %4, align 4
 %39 = load float, float* %5, align 4
 %40 = fcmp ogt float %38, %39
 br i1 %40, label %41, label %42

41:                                               ; preds =
%37
 br label %42

42:                                               ; preds =
%41, %37
 %43 = load float, float* %4, align 4
 %44 = load float, float* %5, align 4
 %45 = fcmp olt float %43, %44
 br i1 %45, label %46, label %47

46:                                               ; preds =
%42
 br label %47
```

```
47:                                               ; preds =
%46, %42
 %48 = load float, float* %4, align 4
 %49 = load float, float* %5, align 4
 %50 = fcmp oge float %48, %49
 br i1 %50, label %51, label %52

51:                                               ; preds =
%47
 br label %52

52:                                               ; preds =
%51, %47
 %53 = load float, float* %4, align 4
 %54 = load float, float* %5, align 4
 %55 = fcmp ole float %53, %54
 br i1 %55, label %56, label %57

56:                                               ; preds =
%52
 br label %57

57:                                               ; preds =
%56, %52
 %58 = load float, float* %4, align 4
 %59 = load float, float* %5, align 4
 %60 = fcmp oeq float %58, %59
 br i1 %60, label %61, label %62

61:                                               ; preds =
%57
 br label %62

62:                                               ; preds =
%61, %57
 %63 = load float, float* %4, align 4
 %64 = load float, float* %5, align 4
 %65 = fcmp une float %63, %64
```

```
 br i1 %65, label %66, label %67

66:                                               ; preds =
%62
 br label %67

67:                                               ; preds =
%66, %62
 %68 = load double, double* %6, align 8
 %69 = load double, double* %7, align 8
 %70 = fcmp ogt double %68, %69
 br i1 %70, label %71, label %72

71:                                               ; preds =
%67
 br label %72

72:                                               ; preds =
%71, %67
 %73 = load double, double* %6, align 8
 %74 = load double, double* %7, align 8
 %75 = fcmp olt double %73, %74
 br i1 %75, label %76, label %77

76:                                               ; preds =
%72
 br label %77

77:                                               ; preds =
%76, %72
 %78 = load double, double* %6, align 8
 %79 = load double, double* %7, align 8
 %80 = fcmp oge double %78, %79
 br i1 %80, label %81, label %82

81:                                               ; preds =
%77
 br label %82
```

```
82:                                               ; preds =
%81, %77
 %83 = load double, double* %6, align 8
 %84 = load double, double* %7, align 8
 %85 = fcmp ole double %83, %84
 br i1 %85, label %86, label %87

86:                                               ; preds =
%82
 br label %87

87:                                               ; preds =
%86, %82
 %88 = load double, double* %6, align 8
 %89 = load double, double* %7, align 8
 %90 = fcmp oeq double %88, %89
 br i1 %90, label %91, label %92

91:                                               ; preds =
%87
 br label %92

92:                                               ; preds =
%91, %87
 %93 = load double, double* %6, align 8
 %94 = load double, double* %7, align 8
 %95 = fcmp une double %93, %94
 br i1 %95, label %96, label %97

96:                                               ; preds =
%92
 br label %97

97:                                               ; preds =
%96, %92
 %98 = load i32, i32* %1, align 4
 ret i32 %98
```

```
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Condicionais

C

```
int main(){
   int a = 1, b = 2;
   a>1? 1 : 0;
}
```

LL

```
; ModuleID =condicionais.c'
source_filename = "condicionais.c"
```

```
target datalayout =
"e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8
:16:32:64-S128"
target triple = "x86_64-pc-windows-msvc19.11.0"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 store i32 1, i32* %1, align 4
 store i32 2, i32* %2, align 4
 %3 = load i32, i32* %1, align 4
 %4 = icmp sgt i32 %3, 1
 %5 = zext i1 %4 to i64
 %6 = select i1 %4, i32 1, i32 0
 ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="none"
"less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 2}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.0.0"}
```

Estruturais

#define e typedef

C

```c
#include <stdio.h>

#define calcArea(r, pi) (pi * r * r)
typedef double real;

int main()
{
   real raio = 1.5;
   real pi = 3.14159;
   real area = calcArea(raio, pi);
}
```

LL

```
; ModuleID = 'estruturais1.c'
source_filename = "estruturais1.c"
target datalayout =
"e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca double, align 8
 %2 = alloca double, align 8
 %3 = alloca double, align 8
 store double 1.500000e+00, double* %1, align 8
 store double 3.141590e+00, double* %2, align 8
 %4 = load double, double* %2, align 8
 %5 = load double, double* %1, align 8
 %6 = fmul double %4, %5
 %7 = load double, double* %1, align 8
 %8 = fmul double %6, %7
```

```
 store double %8, double* %3, align 8
 ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "less-precise-fpmad"="false"
"min-legal-vector-width"="0"
"no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 9.0.0-2 (tags/RELEASE_900/final)"}
```

Observações

- Considerando o exemplo acima podemos ver que a interpretação dos comandos #define e typedef da linguagem C são interpretados na etapa de conversão para a IR, sendo substituídos pelos seus respectivos códigos equivalentes na IR.

Funções

C

```
#include <stdio.h>

int funcaoInt(int a, long long int b)
{
   return a + b;
```

```
}

void funcaoVoid()
{
   printf("hello world!\n");
}

int main()
{
   int x = 11;
   int a = funcaoInt(10, x);
   funcaoVoid();
}
```

LL

```
; ModuleID = 'funcoes.c'
source_filename = "funcoes.c"
target datalayout =
"e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [14 x i8] c"hello
world!\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @funcaoInt(i32, i64) #0 {
 %3 = alloca i32, align 4
 %4 = alloca i64, align 8
 store i32 %0, i32* %3, align 4
 store i64 %1, i64* %4, align 8
 %5 = load i32, i32* %3, align 4
 %6 = sext i32 %5 to i64
 %7 = load i64, i64* %4, align 8
 %8 = add nsw i64 %6, %7
 %9 = trunc i64 %8 to i32
 ret i32 %9
```

```
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @funcaoVoid() #0 {
 %1 = call i32 (i8*, ...) @printf(i8* getelementptr
inbounds ([14 x i8], [14 x i8]* @.str, i64 0, i64 0))
 ret void
}

declare dso_local i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i32, align 4
 store i32 11, i32* %1, align 4
 %3 = load i32, i32* %1, align 4
 %4 = sext i32 %3 to i64
 %5 = call i32 @funcaoInt(i32 10, i64 %4)
 store i32 %5, i32* %2, align 4
 call void @funcaoVoid()
 ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "less-precise-fpmad"="false"
"min-legal-vector-width"="0"
"no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf"
"no-infs-fp-math"="false" "no-jump-tables"="false"
"no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }
```

```
attributes #1 = {
"correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "less-precise-fpmad"="false"
"no-frame-pointer-elim"="true"
"no-frame-pointer-elim-non-leaf"
"no-infs-fp-math"="false" "no-nans-fp-math"="false"
"no-signed-zeros-fp-math"="false"
"no-trapping-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 9.0.0-2 (tags/RELEASE_900/final)"}
```

Observações

- O conceito de funções na IR é semelhante às funções na linguagem C pura. As funções continuam sendo representadas pela nomenclatura recebida pelo usuário (observe no exemplo acima as funções @funcaoInt e @funcaoVoid).
- As funções utilizada de bibliotecas da linguagem como a função "printf" por exemplo, também continuam com a mesma assinatura, mas perceba que a implementação da função ainda não é incorporada ao código, apenas a assinatura da mesma.
- Podemos reparar também que string utilizada no comando printf é considerada uma constante global na IR assinada como @.str