CS 4031
Compiler Construction
Lecture 12

Mahzaib Younas

Lecturer, Department of Computer Science

FAST NUCES CFD

Logical Expression

- An expression that contains operators like +, –, *, / are simple arithmetic expressions, whereas the expression that contains relational operators like , ≥, ≤, or , and, not, etc., are logical expressions.
- The use of logical expression always results in either true or false, which is considered 0/1. 0 indicates false and 1 or a positive number indicates true.

Rules for Logical Expression

$E \rightarrow E1 \text{ or } E2$

$E \rightarrow E1 \text{ and } E2$

$E \rightarrow \text{not } E1$

$E \rightarrow id1 \text{ relop } id2$

$E \rightarrow (E1\ )$

$E \rightarrow \text{true}$

$E \rightarrow \text{false}$

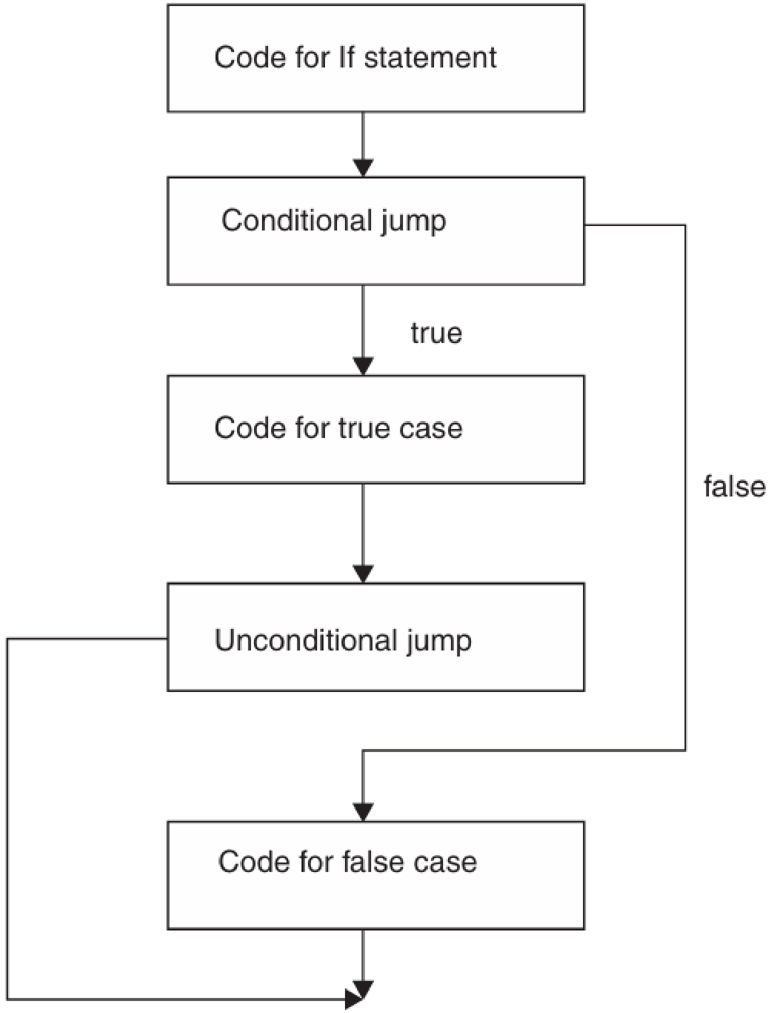
The translation rules to convert to three address code are as follows

$E \rightarrow E_1$ or E_2	{E.value = newtemp(); gen(E.value "=" E_1.value "or" E_2.value)}
$E \rightarrow E_1$ and E_2	{E.value = newtemp(); gen(E.value "=" E_1.value "and" E_2.value)}
$E \rightarrow$ not E_1	{E.value = newtemp(); gen(E.value "=" "not" E_1.value)}
$E \rightarrow (E_1)$	{E.value = E_1.value}
$E \rightarrow id_1$ **relop** id_2	{E.value = newtemp(); gen("if" id_1.value relop.op id_2.value "goto" nextstat + 3) gen(E.value "=" "0") gen("goto" nextstat + 2) gen(E.value "=" "1")}
$E \rightarrow$ true	{E.value = newtemp(); gen(E.value "=" "1")}
$E \rightarrow$ false	{E.value = newtemp(); gen(E.value "=" "0")}

Example:

- Write three address statement and SDT for the x or y and not z.

If Else Structure



The rules for writing different constructs are as follows:

- $S \rightarrow$. if E then S1
- $S \rightarrow$ if E then S1 else S2
- $S \rightarrow$ while E do S1

The translation rules are as follows:

$S \rightarrow$ if E then S_1	{E.true = newlabel(); E.false = S.next; S_1.next = S.next; S.code = E.code || gen(E.true , “ :”) || S_1.code}
$S \rightarrow$ if E then S_1 else S_2	{E.true = newlabel(); E.false = newlabel(); S_1.next = S.next; S_2.next = S.next; S.code = E.code || gen(E.true , “ :”) || S_1.code || gen(“ GOTO ”, S.next) || gen(E.false , “ :”) || S_2.code}
$S \rightarrow$ while E do S_1	{S.begin = newlabel(); E.true = newlabel(); E.false = S.next; S_1.next = S.next; S.code = gen(S.begin “:”) || E.code || gen(E.true , “ :”) || S_1.code || gen(“GOTO” , S.begin)}

Example:

- Give three address code and Syntax Directed Translation for the following:

While (a < 5) do a: = b + 2

Solution:

```
L1:     If a < 5 goto L2
        goto last
L2:     t1 = b + 2
        a = t1
        goto L1
last:
```

Code Generation Process in LLVM

- The frontend of LLVM outputs target independent LLVM IR code
- Compiler Backend has to transform this into machine code for a specific platform
- In this process it can apply optimizations for the targeted platform

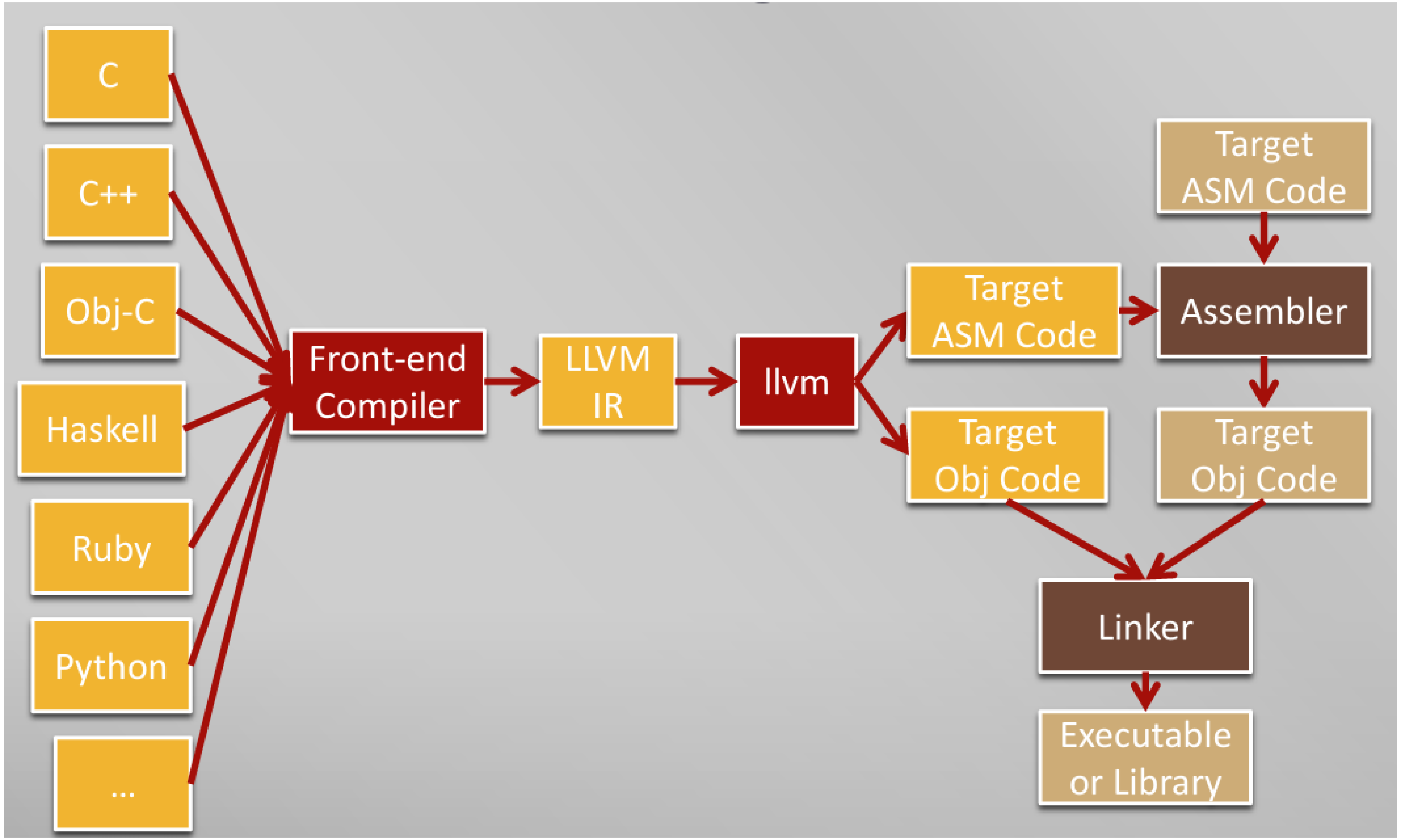
Why we use LLVM

- Modern Compiler (with an arguably modular design).
- Language Agnostic.
- Better documentation (compared to alternatives).
- Less restrictive license.
- Easier to extend, add optimizations, add new targets, etc.

Auto-Vectorization in LLVM

- Converts code using only scalar data types and operations into code using vector-types and operations.
- Vectorizing can lead to significant performance gains.
- The compiler can perform vectorizing for the programmer (sometimes).

LLVM Tool chain for High Level



Main Task of LLVM

- The main purpose of LLVM is to

1. Code Generation
    1. Selection Phase
    2. Scheduling and formation
    3. Register allocation phase
    4. Final phase
2. Auto Vectorization
    1. Loop Vectorization

Code Generator

- Code generation is one of the most complex tasks in the LLVM
- We need to "lower" the LLVM IR into the corresponding machine code
- Optimize code and replace unsupported data types and operations for the target platform
- LLVM supports this process through it's target-independent code generator framework

Processing Steps

1. Instruction Selection
2. Scheduling and Formation
3. SSA-based Machine Code Optimizations
4. Register Allocation
5. Prolog/Epilog Code Insertion
6. Final Optimizations
7. Code Emission

Key Features of TAC in LLVM

- **Each instruction has at most three operands** (hence "three-address").
  - Example: a + b
  - Three address code : %t1 = add i32 %a, %b   ; t1 = a + b
- **Uses SSA form**, meaning each variable is assigned exactly once.
- **Operations are explicit**, including arithmetic, memory access, and control flow.
  - Example: If(a<b) then s1
  - %cmp = icmp slt i32 %a, %b ; Compare: a < b
  - br i1 %cmp, label %if_true, label %if_false ; Branch based on comparison

LLVM Basics

- A Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly.
- Contains many instructions normally found in target assemblies:
- Binary operations:
  - ret, br, add, sub, mul, udiv, sdiv, urem, srem, fadd, fsub, fmul, fdiv.
- Bitwise operations:
  - shl, lshr(logical), ashr (arithmetic), and, or, xor
- Comparisons
  - icmp, fcmp (perhaps, ASMs don’t normally have this form).
- Memory operations
  - load, store, cmpxchg

Binary Operation

Operation	Purpose
ret	Return from a function.
br	Branch to another instruction based on a condition.
add	Perform addition of two operands.
sub	Perform subtraction of two operands.
mul	Perform multiplication of two operands.
udiv	Perform unsigned division of two operands.
sdiv	Perform signed division of two operands.
urem	Compute the remainder of unsigned division.
srem	Compute the remainder of signed division.
fadd	Perform floating-point addition.
fsub	Perform floating-point subtraction.
fmul	Perform floating-point multiplication.
fdiv	Perform floating-point division.

Example:

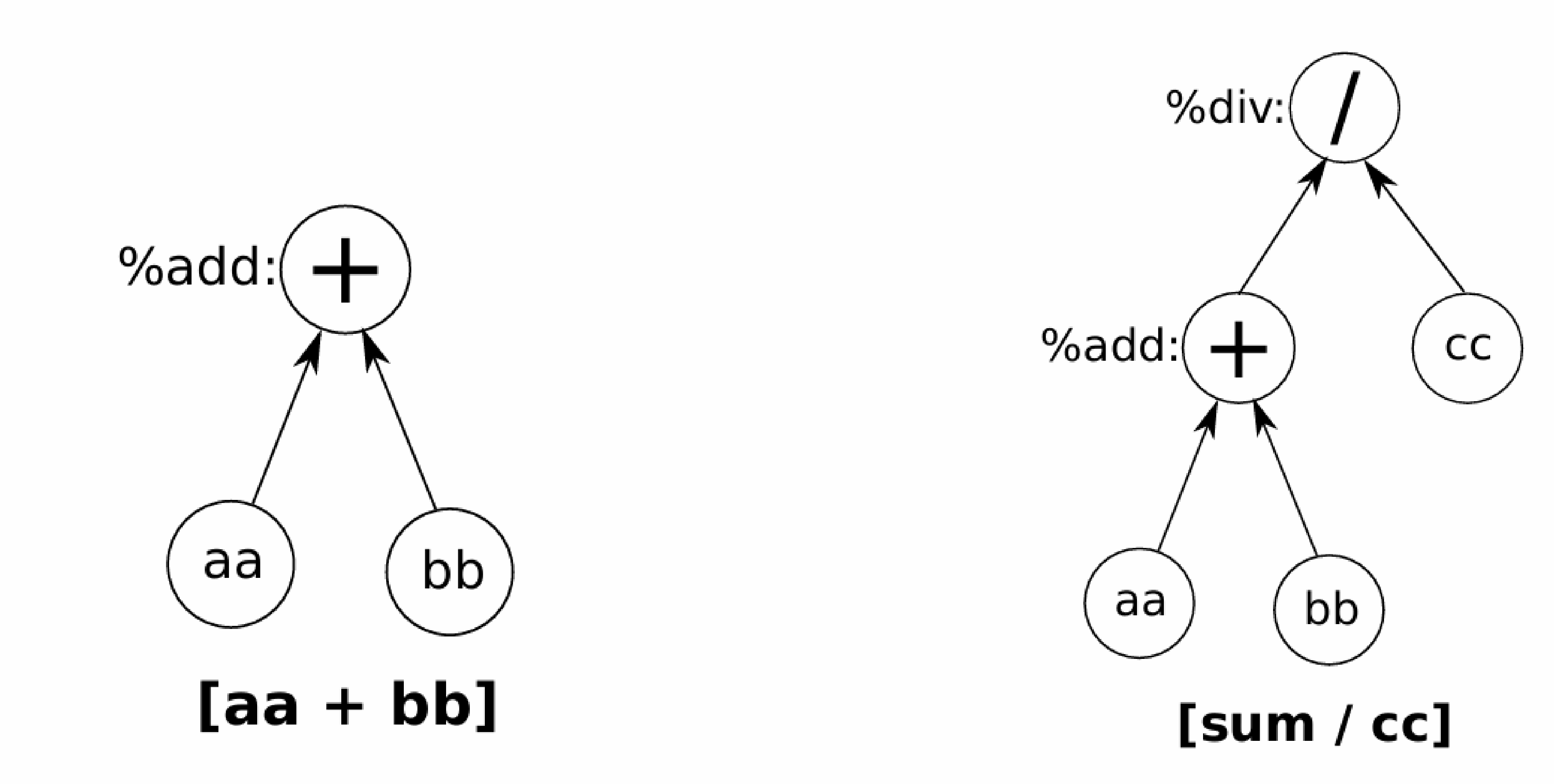
- Let’s consider a simple C function:

```
int foo(int aa , int bb, int cc)
{
int sum = aa + bb;
return sum / cc;
}
```

Convert into Three Address Code

- define i32 @foo(i32 %aa , i32 %bb, i32 %cc)
- {
- entry :
- %add = add i32 %aa, %bb
- %div = sdiv i32 %add, %cc
- ret i32 %div
- }

DAG



Example:

- Let Consider the Simple C Code

```
int main() {
  int a = 10, b = 4, result;
  result = a + b;
  result = a - b;
  result = a * b;
  result = a / b;
  result = a % b;
  return 0;
}
```

LLVM Code : Declaration

- int a = 10, b = 4, result;
- Code:

$1	**INT**
$2	**ID → the variable name like a**
$3	**ASSIGN**
$4	**expr → the evaluated integer like 10**
$5	**SEMICOLON**

```
declaration:
  INT ID ASSIGN expr SEMICOLON {
    printf("@%s = global i32 %d\n", $2, $4);
  }
```

Declaration Example

Detail of Three Address Code	
Printf	prints the declaration as **LLVM IR** code.
@%s	refers to the variable name (LLVM global variable syntax)
global i32 %d	declares it as a **global 32-bit integer** with an initial value.
$2	ID → the variable name like a
$4	expr → the evaluated integer like 10

Expression: Plus

```
expr PLUS expr {
        int reg = get_new_reg();
        printf("  %%r%d = add i32 %d, %d\n", reg, $1, $3);
        $$ = reg;
      }
```

$$ = reg;
Used to store the result of register

Function for Register

```
int reg_count = 0;
int get_new_reg() {
    return reg_count++;
}
```

Explanation:

1. int reg = get_new_reg();

gets a **new register number** for the result of the addition.

2. printf(" %%r%d = add i32 %d, %d\n", reg, $1, $3);

Used for the output statement

%r<reg> = add i32 <left operand>, <right operand>

- reg is the result register.
- $1 is the left-hand side value (expr).
- $3 is the right-hand side value (expr).

Complete code for Arithmetic Expression

```
expr:
      expr PLUS expr {
        int reg = get_new_reg();
        printf("  %%r%d = add i32 %d, %d\n", reg, $1, $3);
        $$ = reg;
      }
    | expr MINUS expr {
        int reg = get_new_reg();
        printf("  %%r%d = sub i32 %d, %d\n", reg, $1, $3);
        $$ = reg;
      }
;
```

Complete code for Arithmetic Expression

```
| expr MUL expr {
        int reg = get_new_reg();
        printf("  %%r%d = mul i32 %d, %d\n", reg, $1, $3);
        $$ = reg;
      }
   | expr DIV expr {
        int reg = get_new_reg();
        printf("  %%r%d = sdiv i32 %d, %d\n", reg, $1, $3);
        $$ = reg;
      }
   | expr MOD expr {
        int reg = get_new_reg();
        printf("  %%r%d = srem i32 %d, %d\n", reg, $1, $3);
        $$ = reg;
      }
   | NUMBER { $$ = $1; }
```

Running Procedure

```
bison -d parser.y
flex scanner.l
gcc parser.tab.c lex.yy.c -o compiler -lfl
./compiler < test.c
```

LLVM in User Code Section

```
int main() {
    ir_file = fopen("output.ll", "w");

   printf(ir_file, "define i32 @main() {\n");

    yyparse();

    fclose(ir_file);
```

Running Procedure for LLVM

- clang output.ll -o program

Sample output of LLVM

```
@a = global i32 10
@b = global i32 4
@res = global i32 0
  %r0 = add i32 10, 4
  store i32 %r0, i32* @res
  %r1 = sub i32 10, 4
  store i32 %r1, i32* @res
  %r2 = mul i32 10, 4
  store i32 %r2, i32* @res
  %r3 = sdiv i32 10, 4
  store i32 %r3, i32* @res
  %r4 = srem i32 10, 4
  store i32 %r4, i32* @res
```

Three Address code for Condition Statements

- Let consider a simple C Code for implementation

```
#include <stdio.h>

int main() {
    int a = 5, b = 3, x;
    if (a > b) {
        x = 1;
    } else {
        x = 2;
    }
    printf("%d\n", x);
    return 0;
}
```

Syntax Analyzer Code

```
%{
#include <stdio.h>
#include <stdlib.h>

int yylex();
void yyerror(const char *s) { fprintf(stderr, "%s\n", s); }
%}

%token INT IF ELSE NUMBER IDENTIFIER
%left '=' '>'
%left '(' ')'
%left '{' '}'
%%
```

Syntax Analyzer Code

```
program:
    stmt ;
stmt:
    "int" IDENTIFIER "=" NUMBER ';'  { printf("Declared: %s = %d\n", $2, $4); }
    | IF '(' expr ')' stmt ELSE stmt { printf("If-Else Statement\n"); } ;
expr:
    IDENTIFIER '>' IDENTIFIER        { printf("Comparison: %s > %s\n", $1, $3); } ;
%%
int main() {
    yyparse();  // Start parsing input
    return 0;
}
```

LLVM Code

```
declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
    ; Allocate memory for variables a, b, and x
    %a = alloca i32
    %b = alloca i32
    %x = alloca i32

    ; Store initial values (a = 5, b = 3)
    store i32 5, i32* %a
    store i32 3, i32* %b
```

LLVM Code

```
; Load values of a and b
    %a_val = load i32, i32* %a
    %b_val = load i32, i32* %b

    ; Compare a > b
    %cmp = icmp sgt i32 %a_val, %b_val

    ; Conditional branch based on comparison
    br i1 %cmp, label %then, label %else

then:
    ; If a > b, assign 1 to x
    store i32 1, i32* %x
    br label %end
```

LLVM Code

```
else:
    ; If a <= b, assign 2 to x
    store i32 2, i32* %x
    br label %end

end:
    ; Load the value of x and return it
    %x_val = load i32, i32* %x
    ret i32 %x_val
}
```