Module 06: Instruction Set Architecture, RISC-V Assembly Programming, and Assembly Program of a C Program

Unit 3: Conditional control instructions for making decisions (if-else) and loops

ITSC 2181 - Introduction to Computer Systems

College of Computing and Informatics

COLLEGE OF COMPUTING AND INFORMATICS

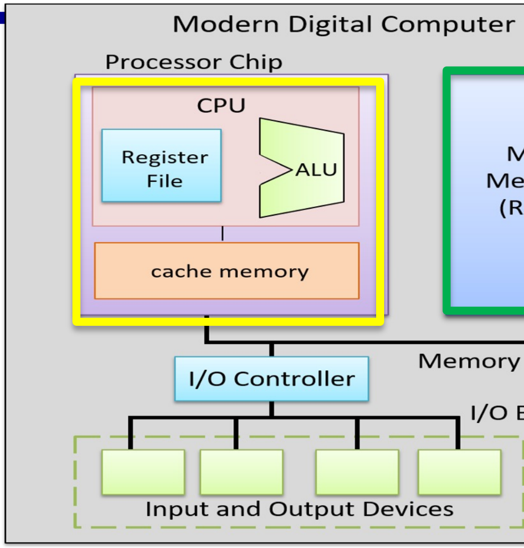
Module 06: Instruction Set Architecture, RISC-V Assembly Programming, and Assembly Program of a C Program

- **Unit 1: Module overview, Instruction Set Architecture (ISA) and assembly programs, registers, instruction operations and operands, register and immediate operands, arithmetic and logic instructions**
- **Unit 2: Memory Operands and Memory Access Instructions**
- ☛ **Unit 3: Conditional control instructions for making decisions (if-else) and loops**
- **Unit 4: Supporting Functions and procedures**
- **Unit 5: Sort examples and comparison with other ISAs**

- **Materials are developed based on textbook:**
  - Computer Organization and Design RISC-V Edition: The Hardware/Software Interface, Amazon
  - RISC-V Specification: https://riscv.org/technical/specifications/
  - ITSC 3181: https://passlab.github.io/ITSC3181/

Instructions Used So Far: add, addi, sub and slli

```
add  x10, x5, x6          // [x10] = [x5] + [x6]
addi x10, x5, 100         // [x10] = [x5] + 100
sub  x11, x5, x6          // [x11] = [x5] – [x6]
slli x12, x5, 5           // [x12] = x5 * 2^^5
ld x12, 32(x5)       // [x12] = Mem[32 + [x5]]
sd x12, 32(x5)       // Mem[32 + [x5]] = [x12]
```



- They can do computation and access memory
  - How to create more capable program? → decision making and repetive

- Another two fundamental programming constructs
  - If-else (→ nested if-else, switch-case, break, continue, etc)
  - for loop (→while loop, do-until, etc)

Flow of Control of Program Execution

- *Flow of control* is the order in which a program performs actions.
  - Up to this point, the order has been sequential.

```
void main () {
       int A[2];
       int a;
       A[0] = 1;
       A[1] = 2;

       a = A[0] + A[1];
       A[0] = a;
       A[1] = a;
}
```

Two major control-flow statements:

- A *branching statement* chooses between two or more possible actions.
  - `if-else, switch-case, etc`
- A *loop statement* repeats an action until a stopping condition occurs.
  - `for, while, do-while`

Three Kinds of Operands and Three Classes of Instructions

- **General form:**
  - **<op word> <dest operand> <src operand 1> <src operand 2>**
  - **E.g.: add x5, x3, x4, which performs [x5] = [x3] + [x4]**

Three Kinds of Operands

1. Register operands, e.g., x0 – x31

2. Immediate operands, e.g., 0, -10, etc

3. Memory operands, e.g. 16(x4)

Module 06: Unit 1

Module 06: Unit 2

Module 06: Unit 3

Three Classes of Instructions

1. **Arithmetic-logic instructions**
   - **add, sub, addi, and, or, shift left|right, etc**
2. **Memory load and store instructions**
   - **lw and sw: Load/store word**
   - **ld and sd: Load/store doubleword**
3. **Control transfer instructions (changing sequence of instruction execution)**
   - **Conditional branch: bne, beq**
   - **Unconditional jump: (`j offset` OR `jal x0 offset`)**
   - **Procedure call and return: jal and jr**

We will study how to use branch instructions to implement if-else and for loop

```
add  x10, x5, x6          // [x10] = [x5] + [x6]
addi x10, x5, 100         // [x10] = [x5] + 100
sub  x11, x5, x6          // [x11] = [x5] – [x6]
slli x12, x5, 5           // [x12] = x5 * 2^^5
ld x12, 32(x5)     // [x12] = Mem[32 + [x5]]
sd x12, 32(x5)     // Mem[32 + [x5]] = [x12]
beq x5, x6, <label1> // if ([x5] == [x6]) …
```

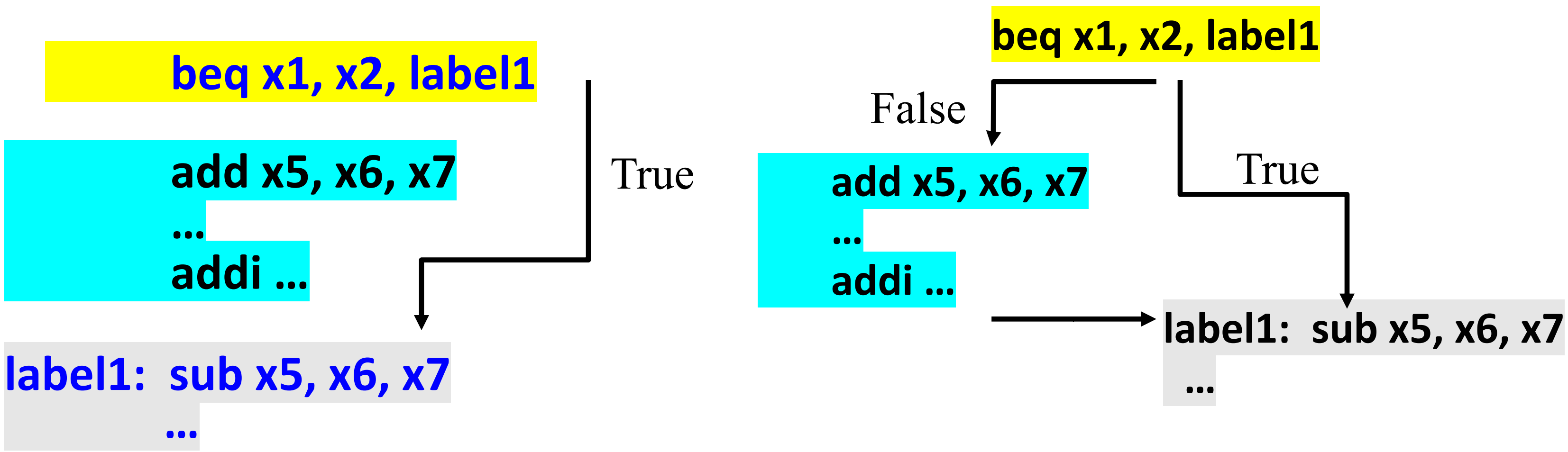
If-else and loop change the sequence of execution, the same for beq

E.g. if (a == b) diff = 0
    else diff = a-b;

Conditional Branch Instruction

Branch to the labeled instruction if a condition is true, otherwise continue

- `beq rs1, rs2, L1`
  - if ([rs1] == [rs2], i.e., true) branch to instruction labeled L1 (branch target);
  - else continue the following instruction



- `bne rs1, rs2, L1`
  - if ([rs1] != [rs2]) branch to instruction labeled L1 (branch target);
  - else continue the following instruction
- J: unconditional jump (not an instruction)
  - beq x0, x0, L1

Translating If Statements 1/2

- C code:

```
if (i==j) f = g+h;  //No else
```

Variable	f	g	h	i	j
Register	x19	x20	x21	x22	x23

- Compiled RISC-V code:

```
        bne x22, x23, NEXT //branch if not equal
        add x19, x20, x21  //Then path
NEXT:
```

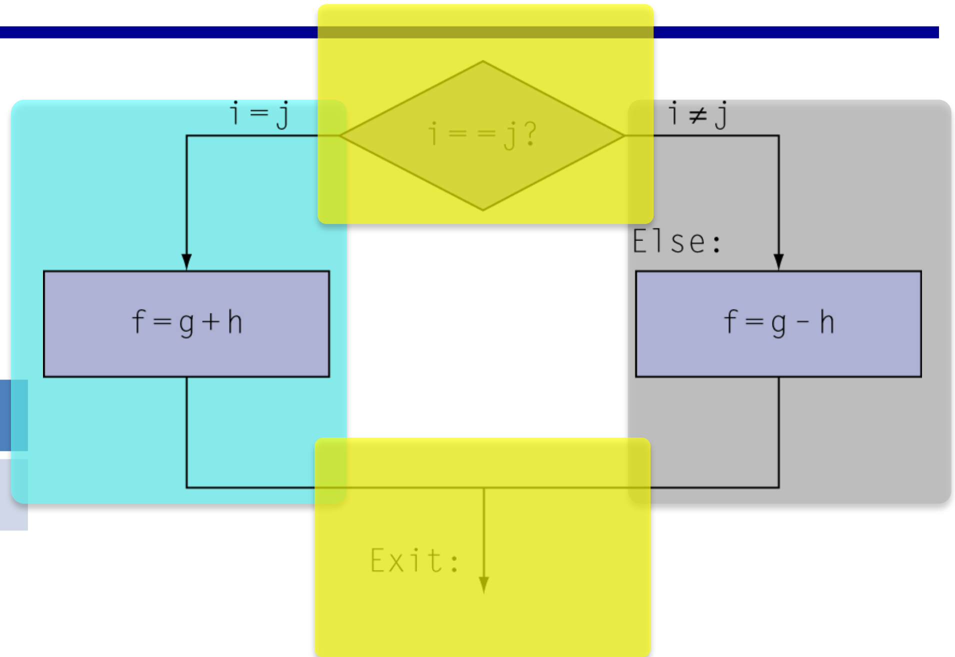
1. **Using bne (reverse of if (==)) to branch to the code after if. In this way, the code following the bne is the code for the truth path of if**

Translating If-else Statements 1/2

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

Variable	f	g	h	i	j
Register	x19	x20	x21	x22	x23



- Compiled RISC-V code:

```
            bne x22, x23, Else //branch if not equal
            add x19, x20, x21  //Then path
            beq x0, x0, Exit //unconditional
Else: sub x19, x20, x21 //Else path
Exit: …
```

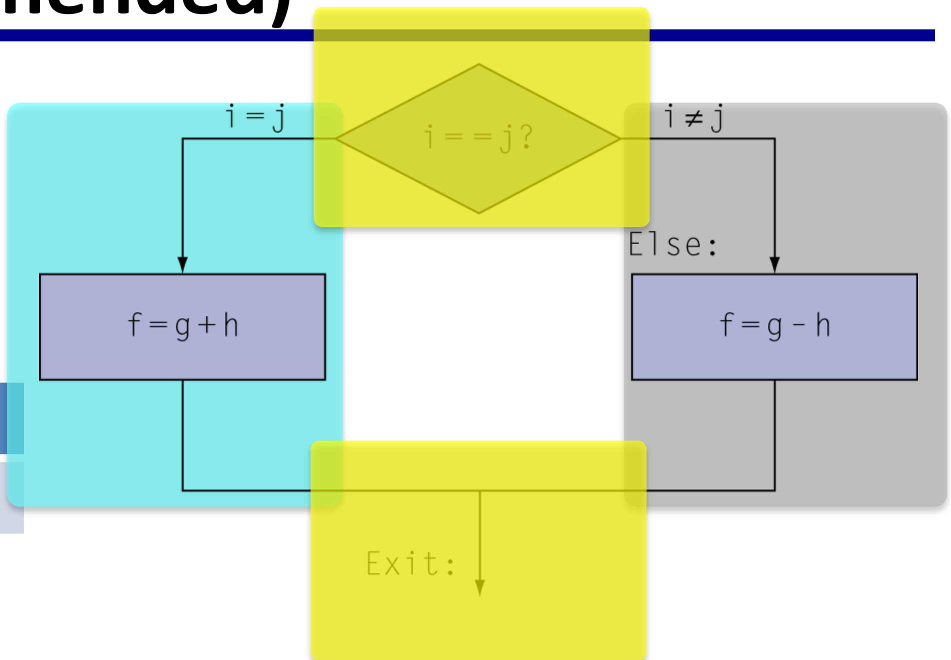
1. Using bne (reverse of if (==)) to branch to the Else path b.c. we want the code following the bne to be the code of the Then path
2. We need “beq x0 x0 Exit”, an unconditional jump, to let Then path terminate since CPU executes instruction in the sequence if not branching.

Translating If-else Statements 2/2 (Not Recommended)

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

Variable	f	g	h	i	j
Register	x19	x20	x21	x22	x23



- Compiled RISC-V code:

```
      beq x22, x23, Then //branch if equal
      sub x19, x20, x21 //Else path
      beq x0, x0, Exit //unconditional
Then: add x19, x20, x21  //Then path
Exit:
```

1. **Using beq (for if (==)) to branch to the Then path**
2. **The instruction that follows the beq is the Else path**
3. **We need “beq x0 x0 Exit”, an unconditional jump, to let Else path terminate since CPU executes instruction in the sequence if not branching.**

Pattern for translating if-else

If (cond) true-path-code

else false-path-code

```
    b(!cond) else-path    //use the branch instruction that has reverse
                          // cond of the condition in the if statement
    true-path-code
    j over
else-path:
    false-path-code
over:
```

Loop Statements and Loop Structure

- A portion of a program that repeats a statement or a group of statements is called a *loop.*
  - the for Statement
  - the while Statement
  - the do-while Statement

```
for (i=0; i<100; i++) { … }

while (i<100) { …; i++; }
```

Loop Structure:

1. Control of loop: ***ICU***
   1. ***Initialization***
   2. ***Condition for termination (continuing)***
   3. ***Updating the condition***
2. Body of loop

`break` and `continue` Statements of the Loop

- `break`: immediately stop the current loop iteration and stop the whole loop

```
while (i<100) {
        if (A[i] == key) break; //found
        i++;
 }
```

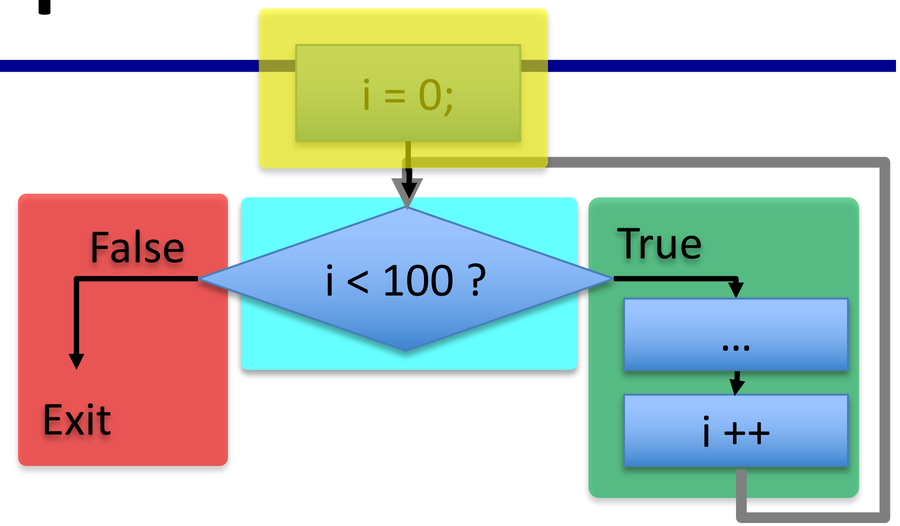
- `continue`: immediately stop the current loop iteration and continue the next iteration from the beginning of the loop

```
for (i=0; i<100; i++) {
    if (isAlreadyProcessed(A[i])) continue;
    process(&A[i]);
}
```

Translating Loop Statement

```
for (i=0; i<100; i++) { … }

while (i<100) { …; i++; }
```



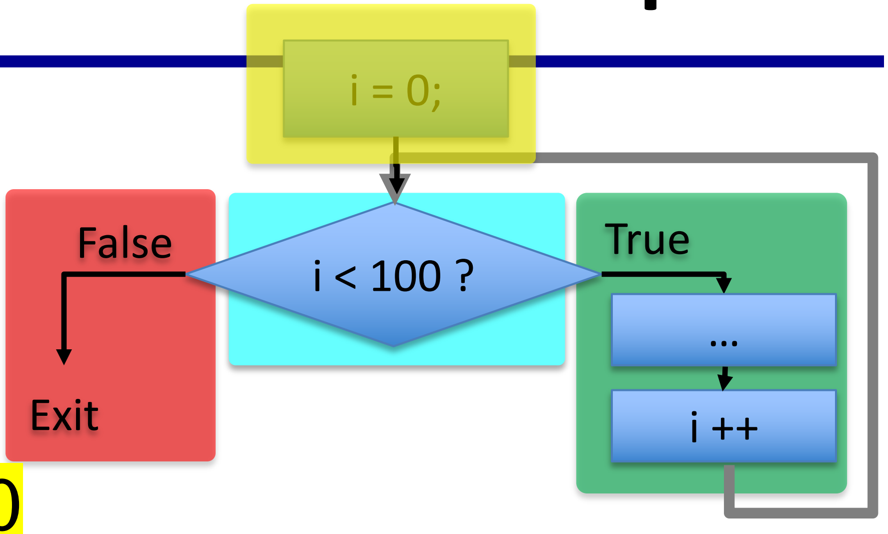

```
Loop: beq/bge x22, x23, Exit
      … # loop body

      addi x22, x22, 1
      beq x0, x0, loop
Exit:
```

- Do the loop control first
  - Init condition
  - Loop condition (using reverse relationship for branch instr)
  - True path (the loop body)
  - Loop back
  - False path (break the loop)
- Then translate the loop body

1. **Using bge for (<) to branch to the false/exit path, which breaks the loop**
2. **The instruction(s) following bge are for the true path, which are for the loop body.**
3. **beq to jumping back to the beginning of the loop**

Translating Loop Statement: for loop



- C code:

  for (i=0; i<100; i++) …

  – i in x22

- RISC-V code:

```
        addi x22, x0, 0
        li x23, 100

Loop:   bge x22, x23, Exit //beq works
        … …
        addi x22, x22, 1 //true,the loop body,i++
        beq  x0,  x0,  Loop
Exit: …
```

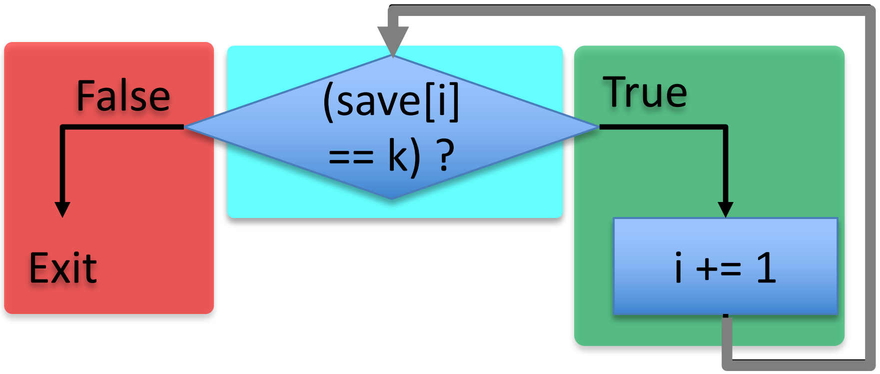
1. Using bge for (<) to branch to the false/exit path, which breaks the loop
2. The instruction(s) following bge are for the true path, which are for the loop body.
3. beq to jumping back to the beginning of the loop

Translating Loop Statement: while loop

- C code:

  while (save[i] == k) i += 1;

  – i in x22, k in x24
  – address of save in x25



- RISC-V code: (save[i] is to be read/loaded)

```
Loop: slli x10, x22, 3   //x10 has i*8
      add  x10, x10, x25 //base+offset
      ld   x9,  0(x10)//save[i] in x9
      bne  x9,  x24, Exit //false
      addi x22, x22, 1 //true,the loop body,i=i+1
      beq  x0,  x0,  Loop
Exit: …
```

1. Using bne for (==) to branch to the false path, which breaks the loop by going to the Exit
2. The instruction(s) following bne are for the true path, which are for the loop body.
3. beq to jumping back to the beginning of the loop

More Conditional Operations

- `blt rs1, rs2, L1`
  - if (rs1 < rs2) branch to instruction labeled L1

- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) branch to instruction labeled L1

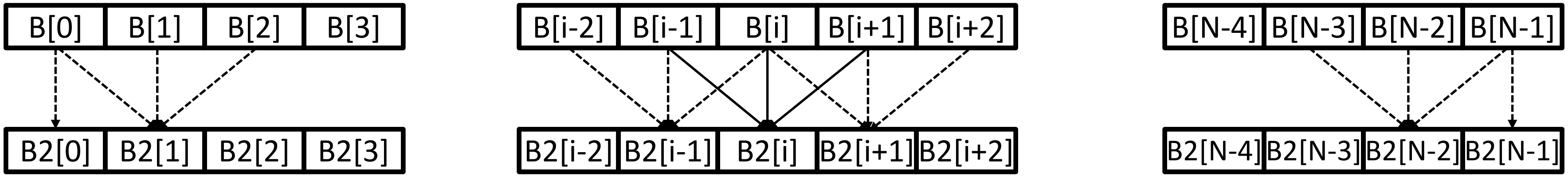
- Example:

if (a > b) a += 1; //a in x22, b in x23

```
    bge  x23, x22, Exit      // branch if b >= a
    addi x22, x22, 1
Exit:
```

for (i=1; i<M-1; i++) B2[i] = B[i-1] + B[i] + B[i+1];

- 1-D stencil: B2[i] = B[i-1] + B[i] + B[i+1]; int type
  - Representing a typical program pattern: Need to access a memory location and its surrounding area



- Converting to assembly
  - Similar to while loop
  - Do the loop control first (init, condition, loop back, etc)
  - Then do the loop body

for (i=1; i<M-1; i++) B2[i] = B[i-1] + B[i] + B[i+1];

- Base address B and B2 are in register x22 and x23. i is stored in register x5, M is stored in x4.

Using bge (>=) for <, i.e. reverse relationship, to exit

```
      addi x5, x0, 1      // i=1
      addi x21, x4, -1    // loop bound x21 has M-1
LOOP: bge x5, x21, Exit
      slliw x6, x5, 2     // x6 now store i*4, slliw is i<<2 (shift left logic)
      add x7, x22, x6     // x7 now stores address of B[i].
      lw x9, 0(x7)        // load B[i] from memory location (x7+0) to x9
      lw x10, -4(x7)      // load B[i-1] to x10
      add x9, x10, x9     // x9 = B[i] + B[i-1]
      lw x10, 4(x7)       //load B[i+1] to x10
      add x9, x10, x9     // x9 = B[i-1] + B[i] + B[i+1]
      add x8, x23, x6     // x8 now stores the address of B2[i]
      sw x9, 0(x8)        // store value for B2[i] from register x9 to memory (x8+0)
      addi x5, x5, 1      // i++
      beq x0, x0, LOOP
Exit:
```

Why Use Reverse Relationship between High-level Language Code and instructions

- To keep the original code sequence and structure as much as possible.

- High level language
  - If (==|>|<, …) true **do the following things**
  - while (==|>|<, …) **do the following things**
  - for (; i<M; …) **do the following things**

- b* Instructions
  - **If (true), go to branch target,**
    - i.e. **do NOT the following things of b***

```
L2:  addi x5, x5, 1
     add x10, x5, x11
     beq x5, x6, L1
     add x10, x10, x9
     sub ….
     …
L1:  sub x10, x10, x9
     add …
     …
```

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23 // signed`
    - −1 < +1
    - `“blt x22 x23” true and branch to target`
  - `x22 > x23 // unsigned`
    - +4,294,967,295 > +1
    - `“bltu x22 x23” false and not branch`

Example

- Find the minimum of an array

```
int A[N];
int min = A[0];
for (i=0; i<N; i++) {
    if (A[i] < min) min = A[i]; //loop body
}
```

A is in t0, min is in t1, i is in t2, N is in t3

```
    # Init condition: i=0
    add t2, x0, x0 # li t2, 0
    lw t1, 0(t0)
Loop: bge t2, t3, Exit # (if i >= N) break the loop, the false path

    slli t6, t2, 2 #mul t6, t2, 4
    add t7, t0, t6
    lw t4, 0(t7)
    blt t4, t1, TRUE
    J FALSE
TRUE: add t1, x0, t4  # copy A[i] to min
FALSE:
     addi t2, t2, 1
    J loop #beq x0 x0 loop
Exit:
```

Switch-case

```
int i;
switch (i) {
    case 0:
        a = 0;
        break;
    case 1:
        a = 1;
        break;
    case 2:
        a = 2;
        break;
    default:
        a = i;
}
```

Label in C

- Label (a program symbol) is the symbolic representation of the address of the memory that the instruction is stored in.

```c
// function to check even or not
void checkEvenOrNot(int num)
{
    if (num % 2 == 0)
        // jump to even
        goto even;
    else
        // jump to odd
        goto odd;

even:
    printf("%d is even", num);
    // return if even
    return;
odd:
    printf("%d is odd", num);
}
```

```c
// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    printf("%d ",n);
    n++;
    if (n <= 10)
        goto label;
}
```

```
0000000000400640 <main>:
  400640:       55                      push   %rbp
  400641:       48 89 e5                mov    %rsp,%rbp
  400644:       48 83 ec 10             sub    $0x10,%rsp
  400648:       31 c0                   xor    %eax,%eax
  40064a:       48 b9 a0 06 40 00 00    movabs $0x4006a0,%rcx
  400651:       00 00 00
  400654:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  40065b:       89 7d f8                mov    %edi,-0x8(%rbp)
  40065e:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
  400662:       48 bf d0 07 40 00 00    movabs $0x4007d0,%rdi
  400669:       00 00 00
  40066c:       89 c6                   mov    %eax,%esi
  40066e:       48 89 ca                mov    %rcx,%rdx
  400671:       b0 00                   mov    $0x0,%al
```