Chapter 2: Instructions: Language of the Computer

2.6 – 2.7 Logical Operations, and Branch Instructions

ITSC 3181 Introduction to Computer Architecture

https://passlab.github.io/ITSC3181/

Department of Computer Science

Yonghong Yan

yyan7@uncc.edu

https://passlab.github.io/yanyh/

Chapter 2: Instructions: Language of the Computer

- Lecture
  - 2.1 Introduction
  - 2.2 Operations of the Computer Hardware
  - 2.3 Operands of the Computer Hardware
- Lecture
  - 2.4 Signed and Unsigned Numbers
  - 2.5 Representing Instructions in the Computer
- ☛ Lecture
  - 2.6 Logical Operations
  - 2.7 Instructions for Making Decisions
- Lecture
  - 2.8 Supporting Procedures in Computer Hardware
  - 2.9 Communicating with People
  - 2.10 RISC-V Addressing for Wide Immediate and Addresses
- Lecture
  - ~~2.11 Parallelism and Instructions: Synchronization~~
  - ~~2.12 Translating and Starting a Program~~
    - We covered before along with C Basics
  - 2.13 A C Sort Example to Put It All Together
  - 2.14 Arrays versus Pointers
    - We covered most before along with C Basics
  - ~~2.15 Advanced Material: Compiling C and Interpreting Java~~
  - 2.16 Real Stuff: MIPS Instructions
  - 2.17 Real Stuff: x86 Instructions
  - ~~2.18 Real Stuff: The rest of RISC-V~~
  - ~~2.19 Fallacies and Pitfalls~~
  - 2.20 Concluding Remarks
  - ~~2.21 Historical Perspective and Further Reading~~

Three Classes of Instructions We Will Focus On:

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

- **Control transfer instructions (changing sequence of instruction execution)**
  - Conditional branch: bne, beq
  - Unconditional jump: j
  - Procedure call and return: jal and jr

Logical Operations

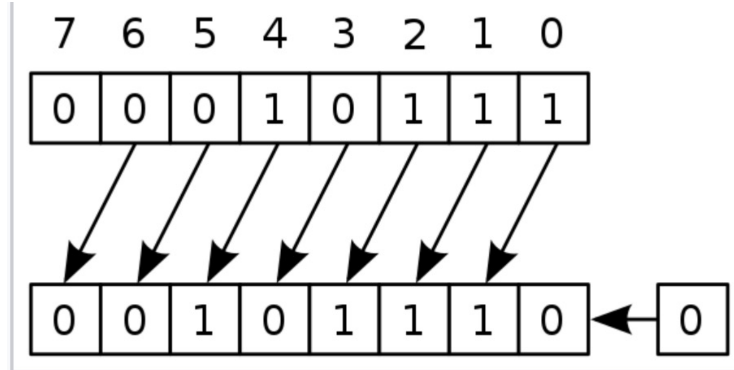
- Instructions for **bitwise** manipulation

<table>
<tr><th>Operation</th><th>C</th><th>Java</th><th>RISC-V</th></tr>
<tr><td>Shift left</td><td><<</td><td><<</td><td><code>sll, slli</code></td></tr>
<tr><td>Shift right</td><td>>></td><td>>>></td><td><code>srl, srli</code></td></tr>
<tr><td>Bit-by-bit AND</td><td>&</td><td>&</td><td><code>and, andi</code></td></tr>
<tr><td>Bit-by-bit OR</td><td>|</td><td>|</td><td><code>or, ori</code></td></tr>
<tr><td>Bit-by-bit XOR</td><td>^</td><td>^</td><td rowspan="2"><code>xor, xori</code></td></tr>
<tr><td>Bit-by-bit NOT</td><td>~</td><td>~</td></tr>
</table>

- Useful for extracting and inserting groups of bits in a word

Shift Logic Operation Examples

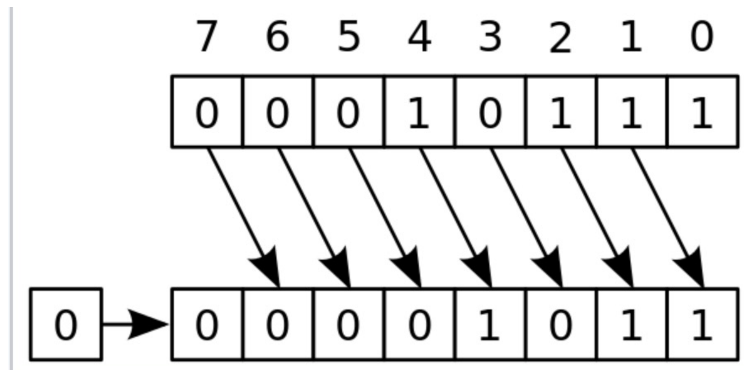
- **Shift Left Logic: `slli` by *i* bits: multiplies by 2^i**
  - C/java: int i = 23; int j = i<<1; //46
  - RISC-V: If i is in x5, and j is stored in x6:
    - slliw x6, x5, 1
    - slliw: shift left logic immediate word



Logical left shift one bit

- **Instruction name**
  - Carries the operand type it operates
    - B: byte, H: half-word, W: word, D: double word

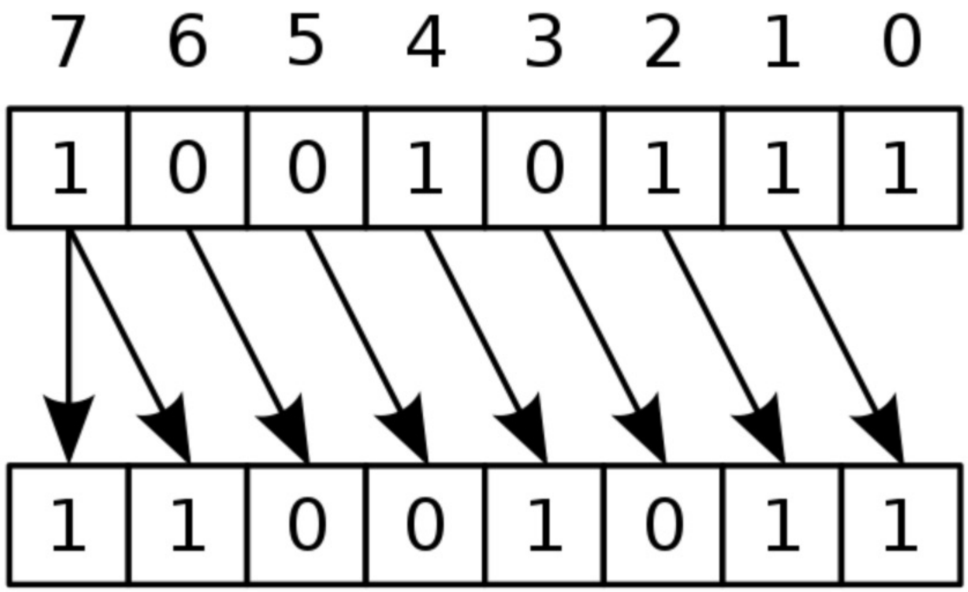
- **Shift Right Logic**
  - Java: int i = 23; int j = i >>> 1; //j=11
  - C: int i = 23; int j = i >> 1; //j=11
  - RISC-V: if i is in x5, j will be in x6:
    - srliw x6, x5, 1
  - Fill in 0, not much used for signed



Logical right shift one bit

Shift Right Arithmetic

- Shift right arithmetic (srai): **Format: srai(w) rd, rs, #immediate**
  - Shift right and fill with sign bit
  - `srai` by *i* bits: divides by 2^i
  - Java: i=-105; int j=i>>1; //-53
  - RISC-V: if i is in x5, j will be in x6:
    - sraiw x6, x5, -1;



-105

-53

Summary of Shift Operations

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- immed: how many positions to shift

- Shift left logical (sll): **Format: ssli(w) rd, rs, #immediate**
  - Shift left and fill with 0 bits
  - `slli` by *i* bits: multiplies by 2^i
  - **E.g. int a = b<<2; //a = b * 4 (2^2)**

- Shift right logical (srl): **Format: srli(w) rd, rs, #immediate**
  - Shift right and fill **with 0 bits**
  - `srli` by *i* bits: divides by 2^i (unsigned only)
  - **E.g. int a = b>>2; //a = b / 4 (2^2)**

- Shift right arithmetic (sra): **Format: srai(w) rd, rs, #immediate**
  - Shift right and fill with sign bit
  - `srai` by *i* bits: divides by 2^i

Shift Operation Encoding

- Use immediate operands, I-Format
  - Immediate: slli, sri, srai, etc

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

- Can use registers for all operands, R-Format
  - Sll, sri, sra

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

AND Operations

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

- Useful to mask bits in a word
  - Select only some bits, clear others to 0

```
and x9,x10,x11
```

  - To only select 4 bits of x10 in the specific positions: Set the bits of x11 in the same positions 1, and the bits in other positions 0, and then perform AND and store the result in a new register x9

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or x9,x10,x11
```

  - To only set 4 bits of x10 in the specific positions 1: Set the bits of x11 in the same positions 1, and the bits in other positions 0, and then perform OR and store the result in a new register x9

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

- Differencing operation
  - E.g. NOT operation

```
xor x9,x10,x12  // NOT operation, invert bits
```

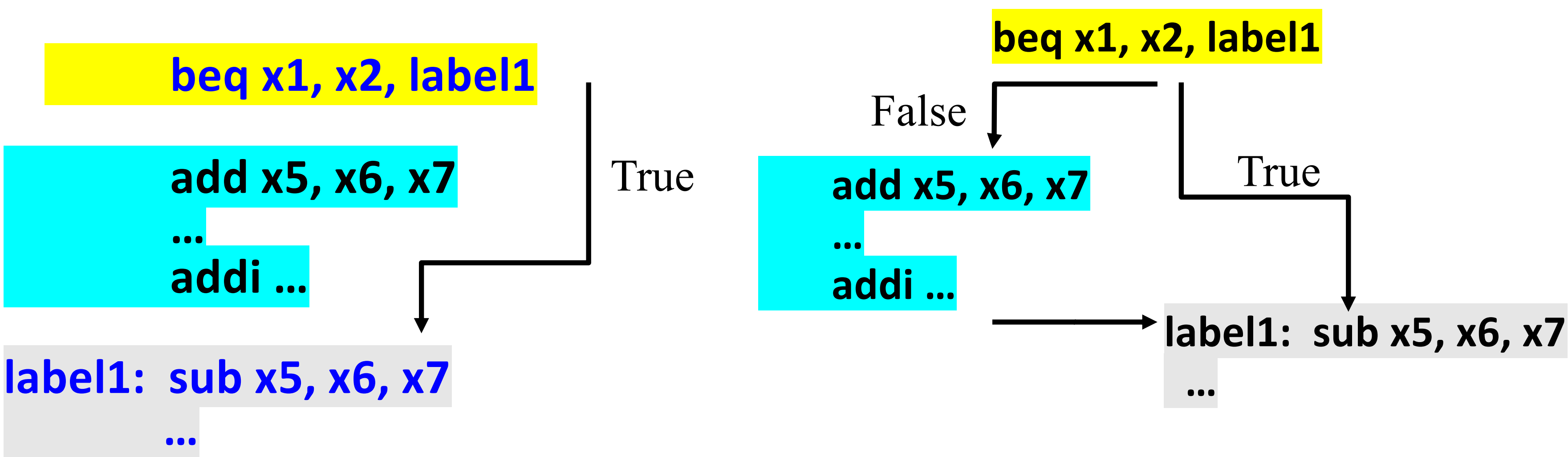
  - To invert bit (logical NOT) of x10: set all bits of x12 as 1, do xor of x10 and x11, and store the result in x9

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x12	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
x9	11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

Conditional Branch

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

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



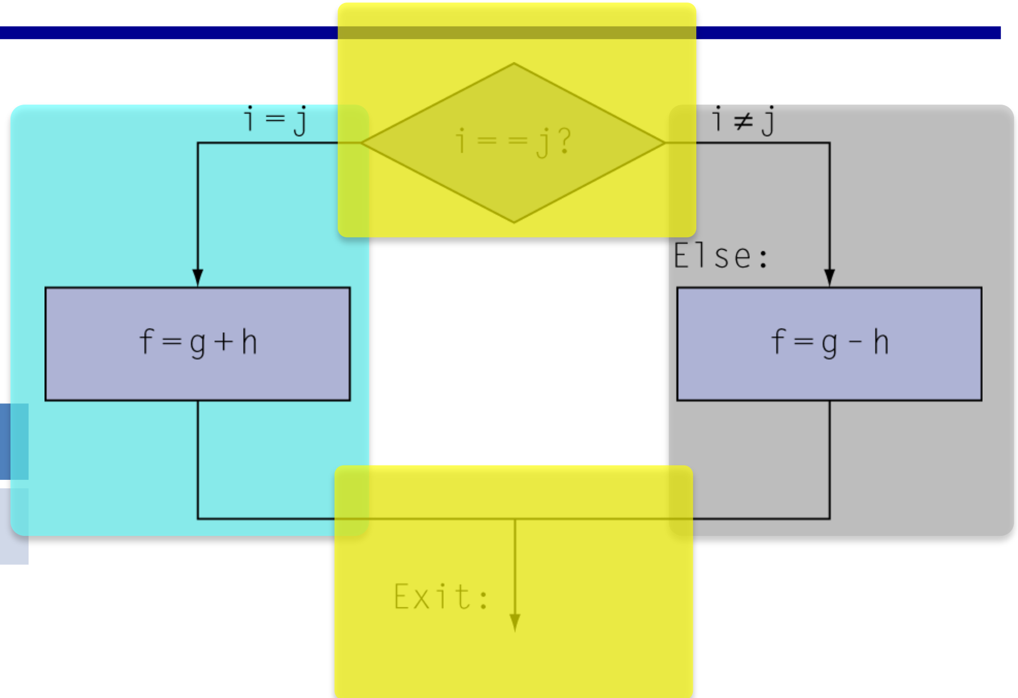
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1 (branch target);
  - else continue the following instruction
- J: unconditional jump (not an instruction)
  - beq x0, x0, L1

Translating If Statements 1/2

- C code:

```
if (i==j) f = g+h;
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: …
```

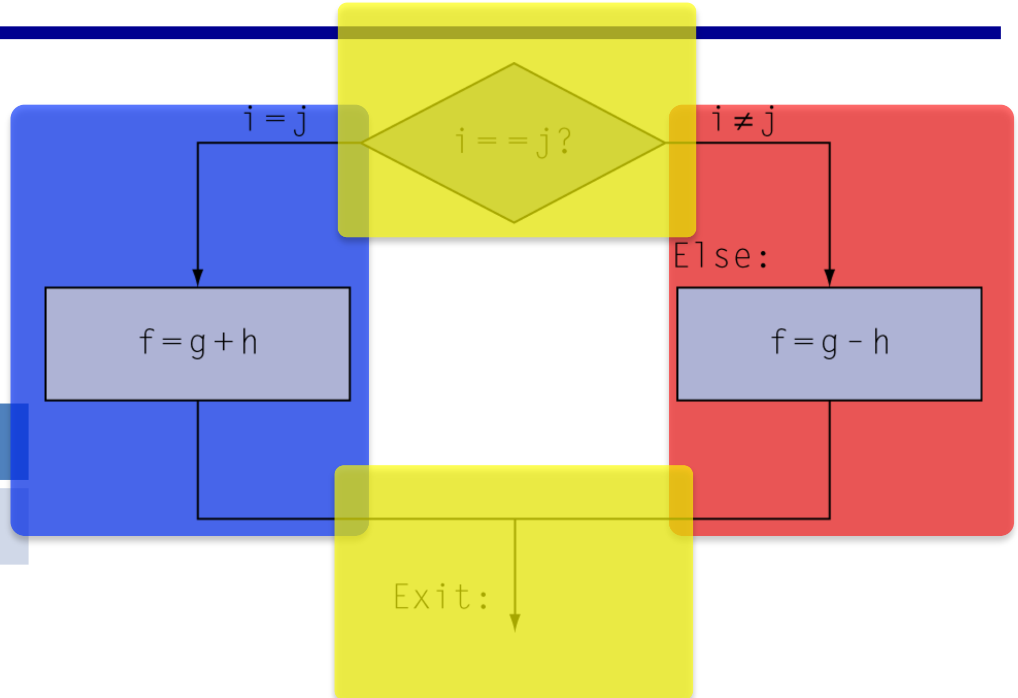
1. **Using bne (reverse of if (==)) to branch to the Else path b.c. we want the code following the bne to be the code of the Then path**
2. **We need “beq x0 x0 Exit”, an unconditional jump, to let Then path terminate since CPU executes instruction in the sequence if not branching.**

Translating If Statements 2/2

- 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:

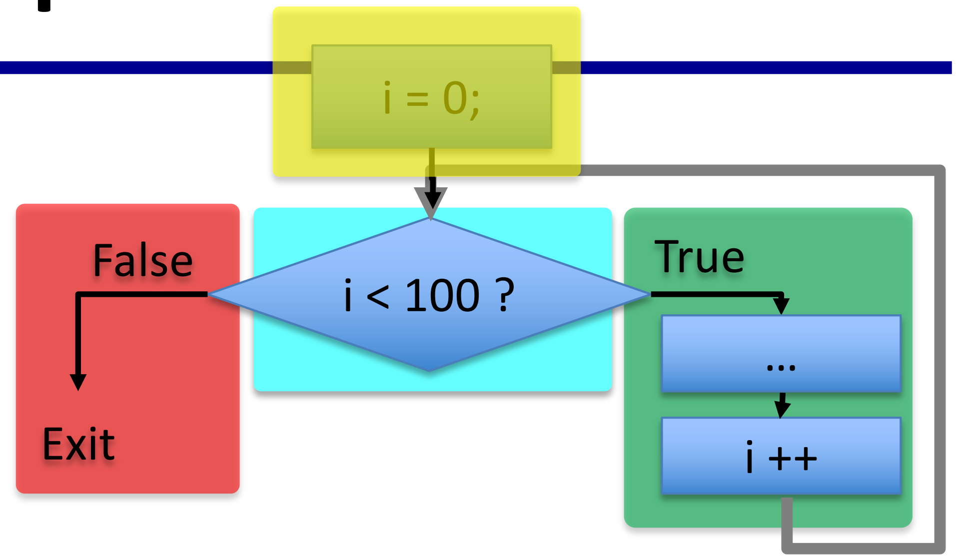
```
      beq x22, x23, Then //branch if equal
      sub x19, x20, x21 //Else path
      beq x0, x0, Exit //unconditional
Then: add x19, x20, x21  //Then path
Exit:
```

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", a unconditional jump, to let Else path terminate since CPU executes instruction in the sequence if not branching.**

Translating Loop Statement

```
for (i=0; i<100; i++) { ... }

while (i<100) { ...; i++; }
```



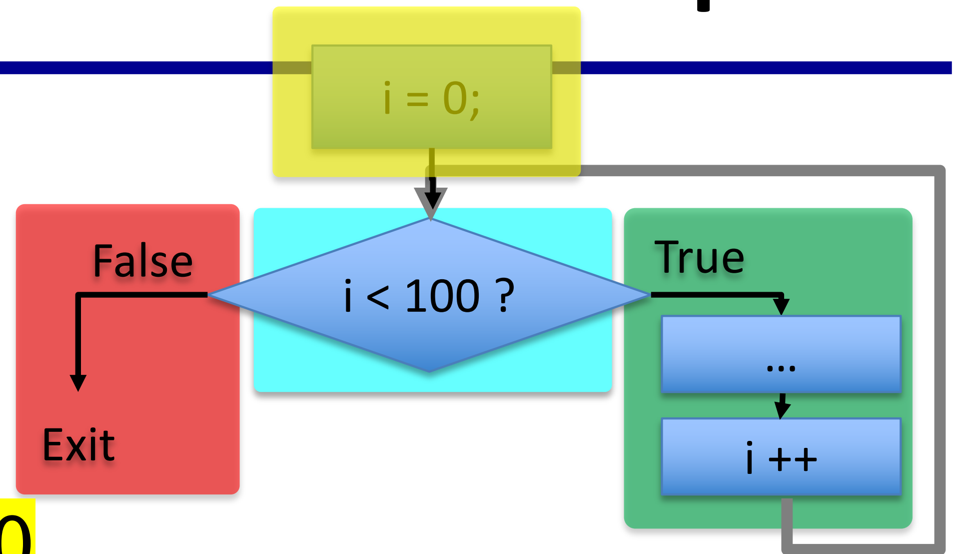

```
Loop: beq/bge x22, x23, Exit
      ...  # loop body

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

- Do the loop structure 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: …
```

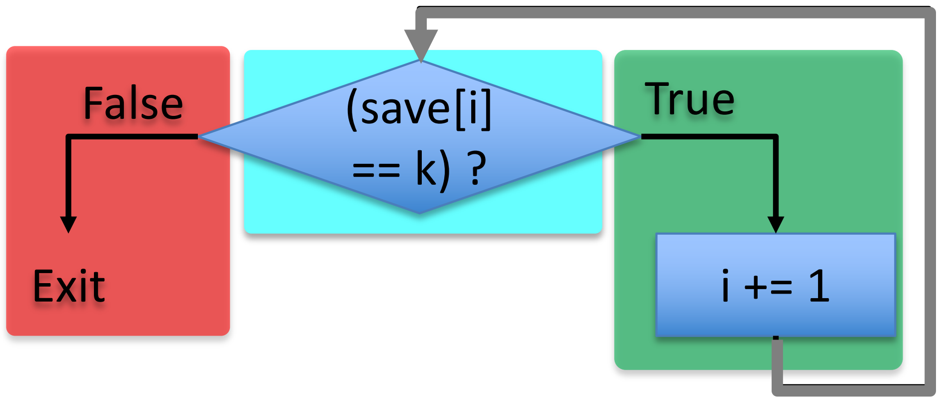
1. Using bge for (<) to branch to the false/exit path, which breaks the loop
2. The instruction(s) following bge are for the true path, which are for the loop body.
3. beq to jumping back to the beginning of the loop

Translating Loop Statement: while loop (textbook 2.7)

- 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

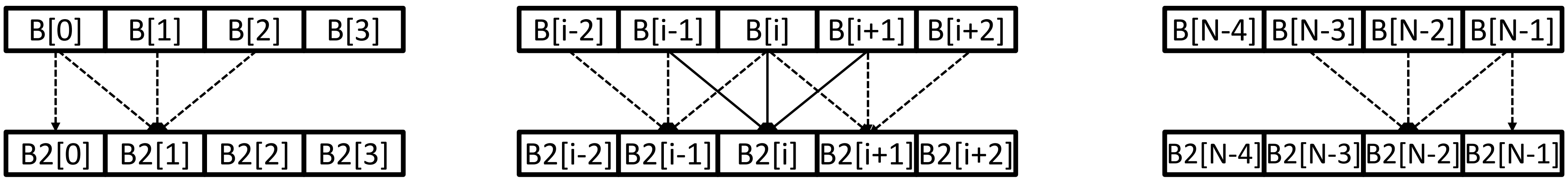
- `blt rs1, rs2, L1`
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) branch to instruction labeled L1
- Example:

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 structure 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`

Code Structure of A Program

```
.globl main    #declare main function
.data          # The .data section of the program is used to
               # reserve memory to use for the variables/arrays
.text                 #The .text section is the actual code
main:                 #definition of main function
```

Declare An Array

```
.globl main    #declare main function
.data          #The .data section, for the variables/arrays
    buffer: .space 8   #declare a symbol named "buffer" for
                       # 8 bytes of memory.
        # For a word element, this correspond to "int buffer[2]"
         #If you need to declare an array of 100 elements of int,
        # use "myArray: .space 400
.text              #The .text section of the program is the actual code
main:              #definition of main function
    la t0, buffer    # set register t0 to have the address of the buffe[0]
    li t1, 8             # Set register t1 to have immediate number 8
```

Random Number Generator

```
li a0, 0  # for random number seed
li a1, 100 # range of random number
li a7, 42 # rand code
ecall # call random number generator to generate a random number stored in a0
```

- Check: https://github.com/TheThirdOne/rars/wiki/Environment-Calls

Memory.s file

```
.globl main    #declare main function
.data         #The .data section of the program is used to claim memory to use for the variables/arrays of the program
buffer: .space 8   #declare a symbol named "buffer" for 8 bytes of memory. For a word element, this coorespond to "int buffer[2]"
                   #This declaration claims 8 bytes of memory.
                   #If you need to declare an array of 100 elements of word, use "myArray: .space 400
.text              #The .text section of the program is the actual code
main:              #definition of main function
la t0, buffer       # set register t0 to have the address of the buffer variable
li t1, 8            # Set register t1 to have immediate number 8
sw t1, 0(t0)        # store a word (4 bytes) of what register t1 contains (8) to memory address 0(t0), which is buffer[0]
lw t2, 0(t0)        # load a word from memory address 0(t0) to register t2, i.e. buffer[0] -> t2
bne t1, t2, failure  # check whether register t1 and t2 contain the same value or not. If not, branch to failure, else continue the next instruction
li t3, 56           # set register t3 to have immediate 56
sw t3, 4(t0)        # store a word of what register t3 contains (56) to memory address 4(t0), which is buffer[1]
addi t0, t0, 4      # increment register t0 (&buffer) by 4, t0 now contains buffer+4, which is &buffer[1]
lw t4, 0(t0)        # load a word from memory 0(t0) (&buffer[1]) to register t4
bne t3, t4, failure  # check whether register t3 and t4 contain the same value or not. If not, branch to failure, else continue.
lw t5, -4(t0)       # load a word from memory -4(t0) to register t5. -4(t0) address is actually &buffer[0] since register t0 now contains the address of buffer[1]
bne t5,t1, failure   # check whether register t5 and t1 contain the same value or not. They should both contain 8
li t1, 0xFF00F007    # set register t1 to have value 0xFF00F007
sw t1, 0(t0)        # store a word of what register t1 contains to memory address 0(t0) (&buffer[1])
lb t2, 0(t0)
```

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;
}
```

Branch is "if (…) goto " of high-level code

```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);
}
```

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

L1:  sub x10, x10, x9
     add …
     …
```

Branch is ”if (…) goto ” of high-level code

- Not directly

  If (…) {

  …

  } else {

  …

  }

```
// 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);
}
```

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

- With branch (if goto), we can implement:
  - if … else
  - for loop, while loop, do loop
  - switch case

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

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;
}
```

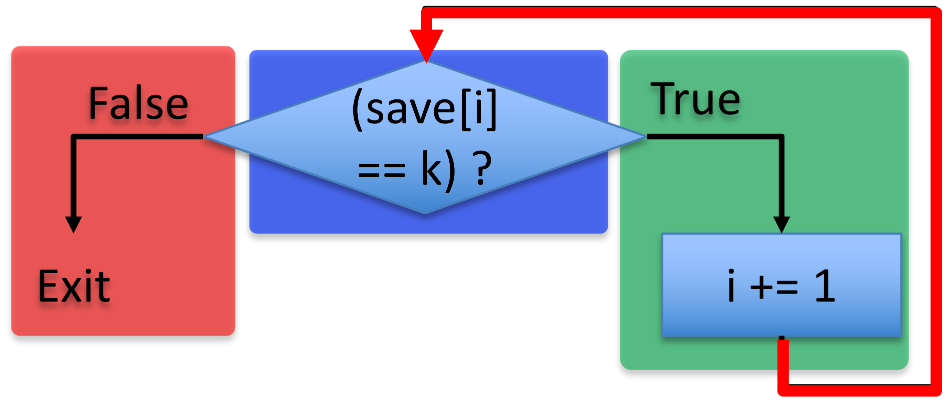
```
0000000000400640 <main>:
  400640:       55                      push   %rbp
  400641:       48 89 e5                mov    %rsp,%rbp
  400644:       48 83 ec 10             sub    $0x10,%rsp
  400648:       31 c0                   xor    %eax,%eax
  40064a:       48 b9 a0 06 40 00 00    movabs $0x4006a0,%rcx
  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
```

Compiling Loop Statements 2/3

- 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)//newbase in x10
      beq  x9,  x24, Body //True
      beq  x0,  x0,  Exit //False
Body: addi x22, x22, 1 //true,the loop body,i=i+1
      beq  x0,  x0,  Loop
Exit: …
```

1. Using beq for (==) to branch to the true path, which is the loop body
2. The instruction following beq is the false path, which breaks the loop by jumping to Exit
3. We need another beq to jumping back to the beginning of the loop, i.e. loop back
4. Not as elegant as the previous version, one more instruction in the code. But not necessary executing more instructions.