2.8 Instructions for making decisions

> The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation …. This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.
> *Burks, Goldstine, and von Neumann, 1947*

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. LEGv8 assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*. The first instruction is

```
CBZ register, L1
```

This instruction means go to the statement labeled `L1` if the value in `register` equals zero. The mnemonic `CBZ` stands for *compare and branch if zero*. The second instruction is

```
CBNZ register, L1
```

It means go to the statement labeled `L1` if the value in `register` does *not* equal zero. The mnemonic `CBNZ` stands for *compare and branch if not zero*. These two instructions are traditionally called *conditional branches*.

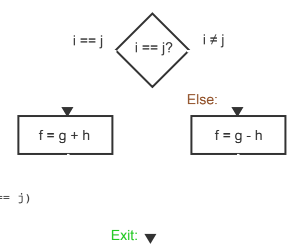
Conditional branch: An instruction that tests a value and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the test.

PARTICIPATION ACTIVITY 2.8.1: Example of compiling if-then-else into conditional branches (COD Figure 2.9).

Start 2x speed

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

```
SUB X9, X22, X23       # X9 = i - j
CBNZ X9, Else          # go to Else if i ≠ j (X9 ≠ 0)
ADD X19, X20, X21      # f = g + h  (skipped if i ≠ j)
B  Exit                # go to Exit
Else: SUB X19, X20, X21    # f = g - h  (skipped if i == j)
Exit:
```



Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores.

Hardware/Software Interface

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

PARTICIPATION ACTIVITY 2.8.2: Branches.

Assume X1 has 50 and X2 has 30. Given this code:

```
      CBNZ X3, Else
      ADD X0, X1, X2
      B Exit
Else: SUB X0, X1, X2
Exit:
```

1) What is "CBNZ" short for?
   - compare and branch if zero
   - compare and branch if not zero

2) If X3 is 0, which instruction executes after CBNZ?
   - ADD
   - SUB

3) `B Exit` is executed when X3 ______ to 0.
   -

is equal

- is not equal

4) If the first instruction were CBZ rather than CBNZ, what instruction should then appear immediately after CBZ?

- `ADD X0, X1, X2`
- `SUB X0, X1, X2`

5) Given the C statement "if (i == 0) f = g + h", what instruction is needed (assuming variables are mapped to registers properly)?

```
        _____  X3, Exit
        ADD    X0, X1, X2
Exit:
```

- CBNZ
- CBZ

CHALLENGE ACTIVITY 2.8.1: Write branch conditions.

Note: Some instructions are deliberately unchangeable.

Start

Convert pseudocode to ARM:

```
if (X2 != X3)
   X4 = X4 + X4;
```

```
ADD X4, X4, X4
ADD X4, X4, X4
ADD X4, X4, X4
L1:
```

1 2 3 4 5

Check Next

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

PARTICIPATION ACTIVITY 2.8.3: Compiling a C while loop.

Start 2x speed

```
while (x == 0) {
   // Loop body
}
```

```
Loop:  CBNZ X0, Exit
  # Loop body
  B Loop
Exit:
```

Example 2.8.1: Compiling a while loop in C.

Here is a traditional loop in C:

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

Assume that `i` and `k` correspond to registers `X22` and `X24` and the base of the array `save` is in `X25`. What is the LEGv8 assembly code corresponding to this C code?

Answer

The first step is to load `save[i]` into a temporary register. Before we can load `save[i]` into a temporary register, we need to have its address. Before we can add `i` to the base of array `save` to form the address, we must multiply the index `i` by 8 due to the byte addressing issue. Fortunately, we can use shift left, since shifting left by 3 bits multiplies by 2^3 or 8. We need to add the label `Loop` to it so that we can branch back to that instruction at the end of the loop:

```
Loop: LSL X10, X22, #3      // Temp reg X10 = i * 8
```

To get the address of `save[i]`, we need to add `X10` and the base of save in `X25`:

```
      ADD X10, X10, X25     // X10 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
      LDUR X9, [X10,#0]     // Temp reg X9 = save[i]
```

The next instruction subtracts k from `save[i]` and puts the difference into `X11` to set up the loop test. If `X11` is not 0, then they are unequal (`save[i] ≠ k`):

```
      SUB X11, X9, X24      // X11 = save[i] - k
```

The next instruction performs the loop test, exiting if `save[i] ≠ k`:

```
      CBNZ X11, Exit        // go to Exit if save[i] ≠ k (X11 ≠ 0)
```

The following instruction adds 1 to `i`:

```
      ADDI X22, X22, #1     // i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
      B     Loop            // go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

Hardware/Software Interface

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a *basic block* is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

Basic block: A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

PARTICIPATION ACTIVITY 2.8.4: Loops.

1) In the above example, the first three instructions (LSL, ADD, LDUR) deal with _____.

- ○ getting save[i]'s value
- ○ executing the loop body

2) In the above example, the loop condition uses ==, but the compiled code uses CBNZ (compare and branch if not zero). Use of CBNZ rather than CBZ is _____.

- ○ a mistake
- ○ correct

3) Given the C loop: while (x != 0) { ... }. != means not equal, and assume X0 is x. Complete the compiled loop:

```
Loop: _____   X0 Exit
      // Loop body
      B Loop
Exit:
```

- ○ CBZ
- ○ CBNZ

The test for equality or inequality is probably the most popular test, but there are many other relationships between two numbers. For example, a *for* loop may want to test to see if the index variable is less than 0. The full set of comparisons are less than (<), less than or equal (≤), greater than (>), greater than or equal (≥), equal (=), and not equal (≠).

Comparison of bit patterns must also deal with the dichotomy between signed and unsigned numbers. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll soon take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking.)

Architects long ago figured out how to handle all these cases by keeping just four extra bits that record what occurred during an instruction. These four added bits, called *condition codes* or *flags* are named:

- *negative (N)* - the result that set the condition code had a 1 in the most significant bit;
- *zero (Z)* - the result that set the condition code was 0;
- *overflow (V)* - the result that set the condition code overflowed; and
- *carry (C)* - the result that set the condition code had a carry out of the most significant bit or a borrow into the most significant bit.

Conditional branches then use combinations of these condition codes to perform the desired sets. In LEGv8, this conditional branch instruction is `B.cond. cond` can be used for any of the signed comparison instructions: `EQ` (= or Equal), `NE` (≠ or Not Equal), `LT` (< or Less Than), `LE` (≤ or Less than or Equal), `GT` (> or Greater Than), or `GE` (≥ or Greater than or Equal). It can also be used for the unsigned comparison instruction: `LO` (< or Lower), `LS` (≤ or Lower or Same), `HI` (> or Higher), or `HS` (≥ or Higher or Same). If the instruction that set the condition codes was a subtract (A-B), the figure below shows the LEGv8 instructions and the values of the condition codes that perform the full set of comparisons for signed and unsigned numbers.

In addition to the 10 conditional branch instructions in the figure below, LEGv8 includes these four branches to complete the testing of the individual condition code bits:

- Branch on minus (`B.MI`): N=1;
- Branch on plus (`B.PL`): N=0;
- Branch on overflow set (`B.VS`): V=1;
- Branch on overflow clear (`B.VC`): V=0.

One alternative to condition codes is to have instructions that compare two registers and then branch based on the result. A second option is to compare two registers and set a third register to a result indicating the success of the comparison, which a subsequent conditional branch instruction then tests to see if register is non-zero (condition is true) or zero (condition is false). Conditional branches in MIPS follow the latter approach (see COD Section 2.16 (Real Stuff: MIPS Instructions)).

One downside to condition codes is that if many instructions always set them, it will create dependencies that will make it difficult for pipelined execution (see COD Chapter 4 (The Processor)). Hence, LEGv8 limits condition code (flag) setting to just a few instructions- `-ADD, ADDI, AND, ANDI, SUB,` and `SUBI`--and even then condition code setting is optional. In LEGv8 assembly language, you simply append an S to the end of one of these instructions if you want to set condition codes: `ADDS, ADDIS, ANDS, ANDIS, SUBS`, and `SUBIS`. The instruction name actually uses the term flag, so the proper name of `ADDS` is "add and set flags."

Figure 2.8.1: How to do all comparisons if the instruction that set the condition codes was a subtract (COD Figure 2.10).

If it was an `ADD` or `AND`, the test is simply on the result of the operation as compared to zero. For `AND`, C and V are always set to 0.

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

CHALLENGE ACTIVITY 2.8.2: Write loops.

Start

Convert pseudocode to ARM:

```
while (X5 != X6){
   X5 = X5 << 1;
}
```

```
Loop:  SUBS X5, X5, X5
       B.EQ Exit
       LSL  X5, X5, #1
       B    Loop
Exit:
```

1 2 3 4 5

Check Next

Bounds check shortcut

Treating signed numbers as if they were unsigned gives us a low-cost way of checking if 0 ≤ x < y, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of x < y checks if x is negative as well as if x is less than y.

Example 2.8.2: Bounds check shortcut.

Use this shortcut to reduce an index-out-of-bounds check: branch to `IndexOutOfBounds` if `X20 ≥ X11` or if `X20` is negative.

Answer

The checking code just uses unsigned greater than or equal to do both checks:

```
SUBS XZR, X20, X11          // Test if X20 >= length or X20 < 0
B.HS IndexOutOfBounds       // if bad, goto Error
```

PARTICIPATION ACTIVITY 2.8.5: Condition codes.

1) How many bits are used to hold a condition code?

- four
- one
-

three

2) What does B.LT stand for?
   - branch on less than or equal to
   - branch on lowly triumph
   - branch on less than

3) How is conditional branching performed in MIPS?
   - MIPS also uses condition codes.
   - MIPS compares two registers and then branches based on the comparison result.

4) Assume the values in X19 and X20 are signed binary numbers, and X20 is positive. Complete the following code to jump to `L2` if the bounds check 0 ≤ X19 < X20 fails.

```
SUBS XZR, X19, X20
_____ L2
```

   - B.HS
   - B.GE

Case/switch statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a *branch address table* or *branch table*, and the program needs only to index into the table and then branch to the appropriate sequence. The branch table is therefore just an array of doublewords containing addresses that correspond to labels in the code. The program loads the appropriate entry from the branch table into a register. It then needs to jump using the address in the register. To support such situations, computers like LEGv8 include a *branch register* instruction (`BR`), meaning an unconditional branch to the address specified in a register. Then it branches to the proper address using this instruction. We'll see an even more popular use of `BR` in the next section.

Branch address table: Also called ***branch table***. A table of addresses of alternative instruction sequences.

Hardware/Software Interface

Although there are many statements for decisions and loops in programming languages like C and Java, the bedrock statement that implements them at the instruction set level is the conditional branch.

PARTICIPATION ACTIVITY 2.8.6: Check yourself: Branches.

C has many kinds of statements for decisions and loops, while LEGv8 has few. Which of the following explain this imbalance?

1) More kinds of decision statements make code easier to read and understand.
   - True
   - False

2) Fewer kinds of decision statements simplify the task of the underlying layer that is responsible for execution.
   - True
   - False

3) More kinds of decision statements mean fewer lines of code, which generally reduces coding time.
   - True
   - False

4) More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.
   - True
   - False

PARTICIPATION ACTIVITY 2.8.7: Check yourself: Logical operators and branches.

1) Why does C provide two sets of

operators for AND (& and &&) and two sets of operators for OR (| and ||), while LEGv8 doesn't?

- ○ Logical operations AND and ORR implement & and |, while conditional branches implement && and ||.
- ○ The above choice has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
- ○ They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.