Chapter 4 – Control Flow

1 Introduction

With the instructions we have seen so far, we can only write programs as purely linear sequences of instructions. However most algorithms need to break this sequential pattern in order to make choices during execution. For example the programmer might want to do one thing *or* another, depending on some condition. To express such choices, general-purpose programming languages offer various syntax constructs known as **control structures**, like if-then-else statements, “for” loops, or “while” loops.

Our goal for today is to understand how control structures are implemented at the assembly level.

2 Branch instructions

Idea To implement control structures, the SCAT processor provides **branch** (aka **jump**) instructions. Instead of incrementing the PC to advance to the next instruction, a *branch* instruction overwrites the PC with a well-chosen value. As a result, the CPU will appear to *jump* to that location and continue executing from there.

Syntax The simplest of these jumps is called “branch always”. Its syntax is the mnemonic `bra` followed by an **offset** e.g. “`bra +8`” or “`bra -12`”. Its effect is to *add* the offset to the PC, instead of adding 4 like ordinary instructions. For instance, “`bra +8`” will *skip* the following instruction, and jump to the one after that. Similarly, “`bra -12`” will rewind execution by three instructions.

Note 1: “`bra +0`” is effectively an infinite loop that will jump to itself forever. When this happens, our simulator stops executing.

Note 2: in practice, the CPU itself does not know about `bra`, it is an example of a **pseudo-instruction**. What happens is the assembler translates every `bra` into an `addi` with PC as destination register.

Note 3: speaking of pseudo-instructions, “`bra +0`” can also be spelled as “`halt`”.

Conditional jump instructions are useful to implement choices: the jump is either taken or ignored, depending on some condition on the values of registers. For instance “`beq ri, rj, offset`” jumps if and only if the values of Ri and Rj are equal. The other possible conditions are listed below.

`asm`	Name	comp code	Description
`beq`	Branch if equal	0000	`if (rd == rs1) PC += sxt(imm)`
`bne`	Branch if not equal	0001	`if (rd != rs1) PC += sxt(imm)`
`blt`	Branch if lower than	0010	`if (rd <  rs1) PC += sxt(imm)`
`bge`	Branch if greater or equal	0011	`if (rd >= rs1) PC += sxt(imm)`

In machine code, conditional jumps are encoded as follows:

- bits 31–28 are always 0b0011 to indicate a type 3 instruction.
- bits 27–24 indicate the desired comparison.
- bits 23–20 and 19–16 indicate which registers to compare.
- the remaining 16 bits encode the offset i.e. the desired jump distance, should the jump be taken.

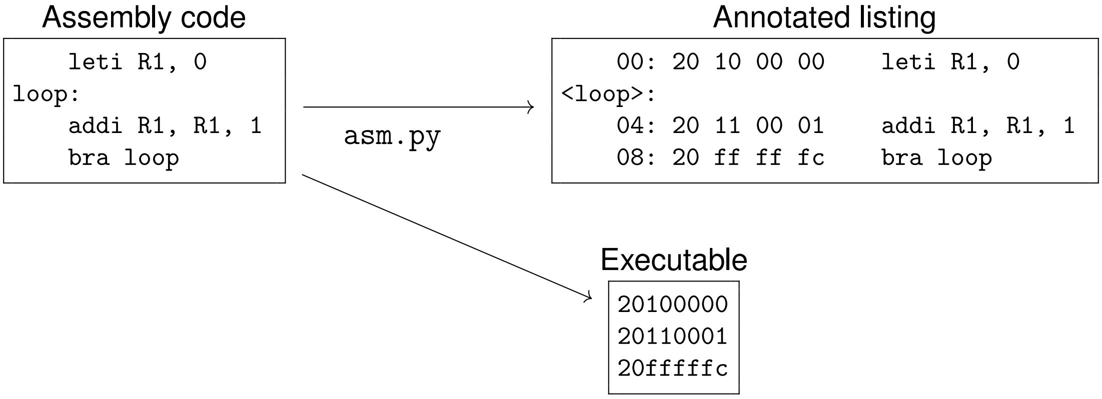
31 28	27 24	23 20	19 16	15 0
0 0 1 1	comp	rd	rs1	imm

Exercise 1 Choose two numbers *A* and *B* between 0 and 1000. Write a program that uses the `leti` pseudo-instruction to assign these numbers to R3 and R4 (syntax: `leti rd, imm`). Then, using conditional and/or mandatory jump(s), set register R5 to the maximum of *A* and *B*. Use `bra +0` to halt execution afterwards.

Reminder: in the simulator, the `step` command executes just one instruction.

3 Symbolic labels

As you may have noticed, calculating jump offsets by hand is not practical. Fortunately with most assemblers (ours included) the programmer can freely associate a **label name** to any program location, and use that name as a jump destination. When generating machine code, the assembler keeps track of where each label is located in memory, and computes all offsets accordingly.
The syntax is illustrated below. Observe how both `leti` and `bra` pseudo-instructions get translated into `addi` (opcode 0x20). To help with debugging, the annotated listing shows both label names and numeric addresses.



Assembly code

```
    leti R1, 0
loop:
    addi R1, R1, 1
    bra loop
```

Annotated listing

```
    00: 20 10 00 00    leti R1, 0
<loop>:
    04: 20 11 00 01    addi R1, R1, 1
    08: 20 ff ff fc    bra loop
```

Executable

```
20100000
20110001
20fffffc
```

Exercise 2 Given a positive integer N, its factorial $N!$ is defined as the product of all the positive integers up to N. In other words, $N! = N \times (N-1) \times ... \times 2 \times 1$. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.
Write a program that uses a loop to compute the factorial of an integer initially stored in `R1`.
You may find these simulator commands useful:

- `step` to execute one instruction
- `continue` to execute the program until it stops by itself
- `breakpoint addr_or_label` to place a new **break point** at some location
- `help cmd_name` to get help on another command

4 Putting it all together: Euclid's GCD Algorithm

In this part you're going to write a program involving both an if-then-else construct and a while loop. Our excuse is compute the *greatest common divisor* of two integers i.e. to find out the largest integer that divides both numbers. But instead of performing repeated divisions, we will follow Euclid's algorithm. To quote Wikipedia[1]:

> *"The method introduced by Euclid for computing greatest common divisors is based on the fact that, given two positive integers a and b such that a > b, the common divisors of a and b are the same as the common divisors of a − b and b.*
> *So, Euclid's method for computing the greatest common divisor of two positive integers consists of replacing the larger number by the difference of the numbers, and repeating this until the two numbers are equal: that is their greatest common divisor. "*

Example Let's unroll the algorithm on a=12 and b=8.

- As stated above, $gcd(12, 8)$ is the same as $gcd(12 - 8, 8)$ i.e. $gcd(4, 8)$.
- We then repeat this step: $gcd(4, 8)$ is the same as $gcd(4, 8 - 4)$ i.e. $gcd(4, 4)$.
- In conclusion, we find that $gcd(12, 8) = gcd(4, 4) = 4$

Exercise 3 **(pen & paper)** With the same technique, compute $gcd(48, 18)$ and $gcd(1071, 462)$.

[1] `https://en.wikipedia.org/wiki/Greatest_common_divisor`. Go look it up !

Exercise 4 (assembly) Write a program to compute $gcd(a, b)$ for any values of a and b. Your code will start by initializing two registers, as illustrated below:

```
leti R1, 48 ; argument a
leti R2, 18 ; argument b
...
```

Don’t hesitate to reuse some of the code you wrote for the previous exercises (maximum, factorial).