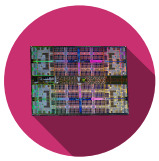


Instruction Set Architecture

Exercises Computer Architecture

1 Instruction-Set Architecture

1.1 Simple C-Code to RISC-V Assembler

Compile the following C-Code into RISC-V assembler.

a)
```c
a = b + c;
```

b)
```c
a = b + c - d;
```

c)
```c
a = b + 6;
```

d)
```c
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

e)
```c
int a = 0xFEDC8765;
```

f)
```c
int a = 0xFEDC8EAB;
```

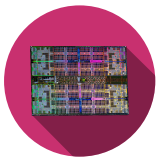
isa/c-to-riscv-01

1.2 Algorithmic C-Code to RISC-V Assembler

Compile the following C-Code into RISC-V assembler.

a)
```c
if (i == j){
  f = g + h;
}
f = f - i;
```

b)
```c
if (i == j){
  f = g + h;
```



```
}
else {
  f = f - i;
}
```

c)

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i=i+1){
  sum = sum + i;
}
```

d)

```
// add the powers of 2 from 1 to 100
int sum = 0;
int i;

for (i=1;i<101; i=i*2){
  sum = sum + i;
}
```

e)

```
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

f)

```
int array[1000];
int i;

for (i=0; i<100; i=i+1){
  array[i] = array[i] * 8;
}
```

g)

```
char str[80] = "CAT";
int len = 0;

// compute length of string
while (str[len]) len++;
```

isa/c-to-riscv-02

1.3 Machine code to RISC-V Assembler

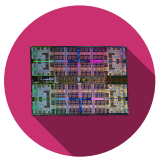
Decode the following Machine Code into RISC-V assembler.

a) **0x41FE 83B3**
b) **0xFDA4 8393**

isa/machinecode-to-riscv-01

1.4 Logic operations on registers

Execute the Assember code and specify the contents of the destination register **rd** if the source registers **rs** contain the following data:



```
s1 = 0x46A1 F1B7
s2 = 0xFFFF 0000
```

a)
```
and s3, s1, s2
```

b)
```
or s4, s1, s2
```

c)
```
xor s5, s1, s2
```

isa/riscv-execution-01

1.5 Logic operations on values

Execute the Assember code and specify the contents of the destination register **rd**, if the source registers **rs** contain the following data:

```
t3 = 0x3A75 0D6F
```

a)
```
and s5, t3, -1484
```

b)
```
or s6, t3, -1484
```

c)
```
xor s7, t3, -1484
```

isa/riscv-execution-02

1.6 RISC-V multiplication

Execute the Assember code and specify the contents of the destination register **rd**, if the source registers **rs** contain the following data:

```
s1 = 0x4000 0000
s2 = 0x8000 0000
```

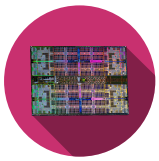
```
mulh s4, s1, s2
mul  s3, s1, s2
```

isa/riscv-execution-03

1.7 Division and modulo

Execute the Assember code and specify the contents of the destination register **rd**, if the source registers **rs** contain the following data:

```
```



```
s1 = 0x0000 0011
s2 = 0x0000 0003
```

```
div s3, s1, s2
rem s4, s1, s2
```

isa/riscv-execution-04

1.8 R-Type to Machine code

Encode the following RISC-V assembler into Machine Code.

a)
```
add s2, s3, s4
```

b)
```
sub t0, t1, t2
```

c)
```
sll s7, t0, s1
```

d)
```
xor s8, s9, s10
```

e)
```
srai t1, t2, 29
```

isa/riscv-to-machinecode-01

1.9 I-Type to Machine code

Encode the following RISC-V assembler into Machine Code.

a)
```
addi s0, s1, 12
```

b)
```
addi s2, t1, -14
```

c)
```
lw t2, -6(s3)
```

d)
```
lh s1, 27(zero)
```

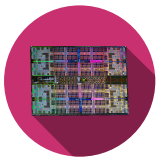
e)
```
lb s4, 0x1F(s4)
```

isa/riscv-to-machinecode-02

1.10 S-Type to Machine code

Encode the following RISC-V assembler into Machine Code.

a)
```
sw t2, -6(s3)
```



b)
```
sh s4, 23(t0)
```

c)
```
sb t5, 0x2D(zero)
```

isa/riscv-to-machinecode-03

1.11 Real-time system

What is the main difference between a “hard” real-time system and a “soft” real-time system?

- ☐ In a hard-real-time system all deadlines must be met while in a soft-real time system occasionally some deadlines may be missed.
- ☐ In a soft-real-time system all deadlines must be met while in a hard-real time system occasionally some deadlines may be missed.
- ☐ A game console must run under a “hard” real-time system, otherwise it’s impossible to run a game.
- ☐ A printer’s injection head must run under a “hard” real-time system, otherwise the printed page may be erroneous.
- ☐ An aircraft’s speed sensor must run under a “hard” real-time system, otherwise the autopilot may be disabled.
- ☐ A “soft” real-time system is faster than a “hard” real-time system.
- ☐ A “hard” real-time system is faster than a “soft” real-time system.
- ☐ Windows 10 is a hard real-time OS.
- ☐ Ubuntu Desktop is a hard real-time OS.
- ☐ RTLinux est un OS capable de travailler en temps réel “hard”.

isa/riscv-to-machinecode-04

1.12 U-Type to Machine code

Encode the following RISC-V assembler into Machine Code.

```
lui s5, 0x8CDEF
```

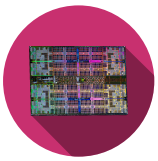
isa/riscv-to-machinecode-05

1.13 J-Type to Machine code

Encode the **first** jump instruction RISC-V assembler into Machine Code. The Program is as follows:

```
0x0000540C        jal ra,func1 # <--
0x00005410        add s1, s2, s3
...
0x001ABC04 func1: add s4, s5, s8
...
```

isa/riscv-to-machinecode-06



2 Laboratory complement

To help you, feel free to use the RISC-V interpreter on https://course.hevs.io/car/riscv-interpreter/ as well as Ripes.

Be careful with variable types!

- **int** type is considered as a signed, 32 bits value.
- **unsigned int** type is considered as an unsigned, 32 bits value.
- If followed by a number (e.g.: **int16_t**), it means the value is on x bits (here 16). If preceded by a **u**, it is unsigned.

uint8_t is an unsigned byte, whereas **int8_t** is a signed byte.

2.1 Basic calculations

a)

```
int b = 1;
int c = 2;
a = b + c;

int b = -1;
int c = 2;
a = b + c;

int b = -12;
int c = 2023;
a = b + c;
```

b)

```
int b = 2;
int c = 3;
int e = -1;
int f = -78;
int g = 2023;
int h = -12;
a = b - c;
d = (e + f) - (g + h);
```

isa/lab-basic-calc

2.2 Memory access

```
uint16_t a = mem[3];
mem[4] = a;

int16_t a = mem[3];
mem[4] = a;
```

isa/lab-memory

2.3 Basic algorithms

1. Transmit the 8-bit memory value at address 0x0000′1000 serially, bit by bit, into the Least Significant Bit (LSB) of memory address 0x0000′1001. The remaining bits of memory address 0x0000′1001 must be **'0'**. Calculate the baud rate in $\frac{\text{Instructions}}{\text{Bit}}$ for the entire transmission.

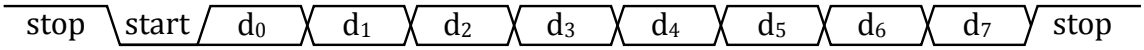


Figure 1 - UART serial transmission

2. Multiply two 4-bit numbers together using one of the commands **bne** or **bge**. The algorithm works as follows: a multiplication is the same as adding the same number x times. For example: $2 * 9 = 9 + 9 = 18$.

isa/lab-basic-algos

2.4 Branching

2.4.1 If / else

```
int a = 1, b = 2, c;

if(a == b) {
  c = 0;
} else if(a > b) {
  c = 1;
} else {
  c = 2;
}
```

2.4.2 Switch case

```
int a;

switch(mem[2]) {
  case 0:
    a = 17;
    break;
  case 3:
    a = 33;
    break;
  case 8:
  case 12:
    a = 10;
    break;
  default:
    a = 99;
}
```

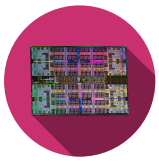
2.4.3 While / Do While

```
// A
int a = 10;

do{a = a - 1;}
while(a != 0);

// B
int a = 10;

do{a = a - 1;}
while(a >= 0);

// C
unsigned int a = 10;

do{a = a - 1;}
while(a >= 0);
```

2.4.4 For

```
int a = 0, i;

for(i = 4; i > mem[0]; i = i - 1) {
  a = a + i;
}
```

isa/lab-branch



2.5 Functions

a)

```
int a = 1, b;
b = doubleIt(a);
b = doubleItOpti(a);

...

// Non-optimized version
// Let's assume a is saved in s0
int doubleIt(int myvar) {
  int a = myvar; // we WANT a in s0 !
  a = a * 2;
  return a;
}

// Optimized version
// Choose your registers freely
// Try having the less possible
//   instructions
int doubleItOpti(int myvar) {
  int a = myvar; //
  a = a * 2;
  return a;
}
```

b)

```
int a=1, b=2, c=3,
 d=4, e=5, f=6, g=7,
 h=8, i=9, j=10, res;
res = sum(a,b,c,d,e,f,
  g,h,i,j);

...

int sum(int v1, int v2,
  int v3, int v4, int v5,
  int v6, int v7, int v8,
  int v9, int v10){
  int c;
  c = v1 + v2 + v3 + v4 +
    v5 + v6 + v7 + v8 +
    + v9 + v10;
  return c;
}
```

isa/lab-fcts

2.6 Advanced Algorithmus

2.6.1 Modulo

The modulo % is an operation performed on two positive integers and is nothing other than the remainder of the division. For example, 5 divided by 3 gives 1 (you can pass 3 once in 5), **remains 2**.

The modulo of a number by 0 is not defined.

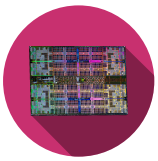
The definition for signed numbers differs from language to language. We only deal with the unsigned version.

Modulo has many uses, allowing you to cap values, extract information, calculate an X and Y position from an X*Y value in an array of known size ...

- Give a code to perform this operation for any positive integer using the RV32IM set.
- How can the same thing be implemented in RV32I? Describe the concept(s).

The role of the compiler is to optimize the code as much as possible. If the operation detected is a modulo with a constant being a power of 2 (e.g. x % 2, y % 8 ...), a variant including no division is possible.

- Give this variant.



The notion of modulo for real numbers arrived with the evolution of computing power, and the result also diverges depending on the language. In C, a specific function from the std libraries is required, **fmod()**. In Python, this operation is native. In either case, they are more resource-intensive and require the handling of specific cases (NaN, infinity).

2.6.2 °F -> °C

We want to create a function capable of converting degrees Fahrenheit to Celsius. Since Fahrenheit values range from 32 to 1000, accuracy to the nearest degree is sufficient.

The formula is simple: $C = (F - 32) * \frac{5}{9}$

This would be handy if an Floating Point Unit (FPU) were available, but only the basic RV32I instruction set is supported.

To get around this problem, you can use a few tricks whose algorithm is as follows:

A. Calculate $C = F - 32$.
B. Multiply by 5
C. Divide by 9
   - $\frac{1}{9}$ can be stored in a special binary representation

```
+-----+-----+-----+-----+-----+-----+------------+------------+
| b31 | b30 | b29 | b28 | b27 | ... |         b1 |         b0 |
+-----+-----+-----+-----+-----+-----+------------+------------+
| 2^0 | 2^-1| 2^-2| 2^-3| 2^-4| ... |      2^-30 |      2^-31 |
+-----+-----+-----+-----+-----+-----+------------+------------+
| 1   | 1/2 | 1/4 | 1/8 | 1/16| ... |1/1737418240|1/2147483648|
+-----+-----+-----+-----+-----+-----+------------+------------+
```

     in this case, it's $0000\_1110\_0011\_1000\_1110\_0011\_1000\_1110_2$.
   - The constant can be pre-calculated and is $\frac{2^n}{9} + 1$. The greater the n, the greater the precision. The number of bits defines the maximum size of n. The $+1$ is a rounding-off for lost precision.
   - Let's take $n = 16$. Our magic number is $\text{magic} = \frac{2^n}{9} + 1 = \frac{65536}{9} + 1 = 7282$.
   - Multiply the value by this magic number
D. Then divide by 2^n. In the case $n = 16 \rightarrow \frac{1}{65536}$.

To simplify the work, several assumptions are made:

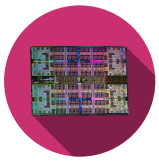
- The magic number and the temperature in Fahrenheit are always positive.
- The size of the largest multiplication is :

$$\begin{aligned} \text{nbBits}_{\text{max}_{\text{fahrenheit}}} + \text{nbBits}_{\text{mult5}} + \text{nbBits}_{\text{magicNumber}} = \\ 10(\text{max. } 1000\text{-}32) + 3 + (n - \text{nbBits}_{\text{div9}} + 1) = \\ 10 + 3 + (16 - 4 + 1) = \\ = 26 \text{ bits} \end{aligned} \tag{1}$$

  - It never exceeds 32 bits for $n < 23$.

Test and optimize the function:

- Write the corresponding code.



- Test with different Fahrenheit values.
- Test with several values for n (16, 18, 20). *Don't forget to recalculate the magic number.*
- Test the function with $n = [22, 23]$, for $°F = [100, 400, 1000]$.

isa/lab-adv-algos