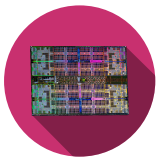


Architecture d'ensemble d'instructions

Exercices Architecture des ordinateurs

1 Instruction-Set Architecture

1.1 Code C simple vers assembleur RISC-V

Compilez le code C suivant en assembleur RISC-V.

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;
```

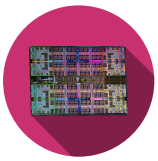
isa/c-to-riscv-01

1.2 Code C algorithmique vers assembleur RISC-V

Compilez le code C suivant en assembleur RISC-V.

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

b)
```c

```



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

c)

```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)

```c
// 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)

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

f)

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

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

g)

```c
char str[80] = "CAT\0"; // null-terminated string, i.e. [C, A, T, \0]
int len = 0;

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

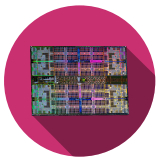
isa/c-to-riscv-02

1.3 Code machine vers assembleur RISC-V

Decoder le code machine suivant en assembleur RISC-V.

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

isa/machinecode-to-riscv-01



1.4 Opérations logiques sur registres

Exécutez le code assembleur et indiquez le contenu du registre cible **rd** si les registres sources **rs** contiennent les données suivantes :

```
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 Opérations logiques sur valeurs

Exécutez le code assembleur et indiquez le contenu du registre cible **rd** si les registres sources **rs** contiennent les données suivantes :

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

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

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

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

isa/riscv-execution-02

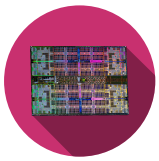
1.6 Multiplications en RISC-V

Exécutez le code assembleur et indiquez le contenu du registre cible **rd** si les registres sources **rs** contiennent les données suivantes :

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

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

isa/riscv-execution-03



1.7 Division et modulo

Exécutez le code assembleur et indiquez le contenu du registre cible **`rd`** si les registres sources **`rs`** contiennent les données suivantes :

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

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

isa/riscv-execution-04

1.8 Type R vers code machine

Encoder l'assembleur RISC-V suivant en code machine.

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 Type I vers code machine

Encoder l'assembleur RISC-V suivant en code machine.

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

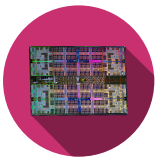
b)
```
addi s2, t1, -14
```

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

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

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

isa/riscv-to-machinecode-02



1.10 Type S vers code machine

Encoder l'assembleur RISC-V suivant en code machine.

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

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

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

isa/riscv-to-machinecode-03

1.11 Système temps réel

Quelle est la principale différence entre un système en temps réel « dur » et un système en temps réel « souple » ?

- ☐ Dans un système à temps réel « hard », tous les délais doivent être respectés, alors que dans un système à temps réel « soft », certains délais peuvent occasionnellement ne pas être respectés.
- ☐ Dans un système à temps réel « soft », tous les délais doivent être respectés, alors que dans un système à temps réel « hard », certains délais peuvent occasionnellement ne pas être respectés.
- ☐ Une console de jeu doit tourner sous système à temps réel « hard », sans quoi il est impossible de faire tourner un jeu.
- ☐ La tête d'injection d'une imprimante doit tourner sous système à temps réel « hard », sans quoi la page imprimée peut être erronée.
- ☐ La sonde de mesure de vitesse d'un avion doit tourner sous système à temps réel « hard », sans quoi le pilote automatique peut se désactiver.
- ☐ Un système à temps réel « soft » est plus rapide qu'un système à temps réel « hard ».
- ☐ Un système à temps réel « hard » est plus rapide qu'un système à temps réel « soft ».
- ☐ Windows 10 est un OS capable de travailler en temps réel « hard ».
- ☐ Ubuntu Desktop est un OS capable de travailler en temps réel « hard ».
- ☐ RTLinux est un OS capable de travailler en temps réel « hard ».

isa/riscv-to-machinecode-04

1.12 Type U vers code machine

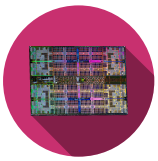
Encoder l'assembleur RISC-V suivant en code machine.

```
lui s5, 0x8CDEF
```

isa/riscv-to-machinecode-05

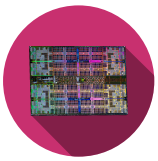
1.13 Type J vers code machine

Encodez la **première** instruction de saut de l'assembleur RISC-V en code machine. Le programme est le suivant :



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

isa/riscv-to-machinecode-06



2 Complément au laboratoire

Pour vous aider, n'hésitez pas à utiliser l'interpréteur RISC-V sur https://course.hevs.io/car/riscv-interpreter/ ainsi que Ripes.

Attention aux types des variables !

- Le type **int** est considéré de taille 32 bits signé.
- Le type **unsigned int** est considéré de taille 32 bits non-signé.
- Si il est suivi d'un nombre (ex: **int16_t**), cela signifie que la variable est sur x bits (ici 16). Si précédé d'un **u**, il est non-signé.

uint8_t est donc un byte non-signé, tandis que **int8_t** est un byte signé.

2.1 Calculs de base

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

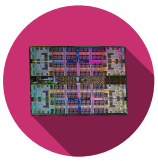
2.2 Accès mémoire

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

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

isa/lab-memory

2.3 Algorithmes basiques

1. Transmettre la valeur de 8 bits de la mémoire à l'adresse 0x0000'1000 en série, bit par bit, dans le Least Significant Bit (LSB) de la mémoire à l'adresse 0x0000'1001. Les bits restants de l'adresse mémoire 0x0000'1001 doivent être ‹ **0** ›. Calculez le débit de bauds en $\frac{\text{Instructions}}{\text{Bit}}$ pour l'ensemble de la transmission.



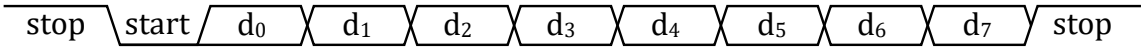


Figure 1 - Transmission sérielle UART

2. Multiplier deux nombres de 4 bits ensemble en utilisant en plus une des commandes **bne** ou **bge**. L'algorithme fonctionne de la manière suivante : une multiplication est la même chose que l'addition x fois du même nombre. Par exemple : $2 * 9 = 9 + 9 = 18$.

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;
}
```

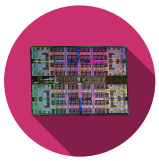
2.4.3 While / Do While

```
// 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

Le modulo % est une opération qui se pratique sur deux nombres entiers positifs et n'est autre que le reste de la division. Par exemple, 5 divisé par 3 donne 1 (on peut faire passer une fois 3 dans 5), **reste 2**.

Le modulo d'un nombre par 0 n'est pas défini.

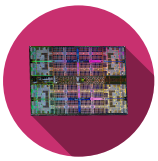
La définition pour les nombres signés diverge selon le langage. Nous ne traitons que la version avec les unsigned.

Le modulo a plusieurs utilités, permettant de plafonner des valeurs, extraire de l'information, calculer une position X et Y à partir d'une valeur X*Y dans un tableau de taille connue …

- Donner un code permettant d'effectuer cette opération pour n'importe quel entier positif en utilisant le set RV32IM.
- Comment la même chose peut-elle être implémentée dans RV32I ? Décrivez le(s) concept(s).

Le rôle du compilateur est d'optimiser au mieux le code. Si l'opération détectée est un modulo avec une constante étant une puissance de 2 (ex. x % 2, y % 8 …), une variante n'incluant aucune division est possible.

- Donner cette variante.



La notion de modulo pour les nombres réels est arrivée avec l'évolution des puissances de calcul et le résultat diverge aussi selon le langage. En C, une fonction spécifique des librairies std est nécessaire, **fmod()**. En Python, cette opération est native. Dans tous le cas, elles sont plus gourmandes en ressources et nécessitent de gérer certains cas spécifiques (NaN, infinity).

2.6.2 °F -> °C

Nous souhaitons créer une fonction capable de convertir les degrés Fahrenheit en Celsius. Comme les valeurs Fahrenheit varient entre 32 et 1000, une précision au degré près est suffisante.

La formule est simple : $C = (F - 32) * \frac{5}{9}$.

Ce serait pratique si un Floating Point Unit (FPU) était disponible, mais seul le jeu d'instructions de base RV32I est supporté.

Pour contourner ce problème, vous pouvez utiliser quelques astuces dont l'algorithme est le suivant :

A. Calculer $C = F - 32$.
B. Multiplier par 5
C. Diviser par 9
   - $\frac{1}{9}$ peut être enregistré par une représentation binaire spéciale

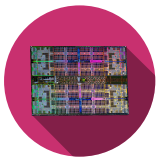
```
+-----+-----+-----+-----+-----+-----+------------+------------+
| b31 | b30 | b29 | b28 | b27 | ... |         b1 |         b0 |
+-----+-----+-----+-----+-----+-----+------------+------------+
| 2^0 | 2^-1| 2^-2| 2^-3| 2^-4| ... |      2^-30 |      2^-31 |
+-----+-----+-----+-----+-----+-----+------------+------------+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

     dans ce cas, il s'agit de $0000\_1110\_0011\_1000\_1110\_0011\_1000\_1110_2$.
   - La constante peut être précalculée et vaut $\frac{2^n}{9} + 1$. Plus n est grand, plus la précision est élevée. Le nombre de bits définit la taille maximale de n. Le $+1$ est un arrondi pour la précision perdue.
   - Prenons $n = 16$. Notre nombre magique est donc magic $= \frac{2^n}{9} + 1 = \frac{65536}{9} + 1 = 7282$.
   - Il faut multiplier la valeur par ce nombre magique
D. ensuite le diviser par 2^n. Dans le cas $n = 16 \rightarrow \frac{1}{65536}$.

Pour simplifier le travail, plusieurs hypothèses sont faites :
- Le nombre magique et la température en Fahrenheit sont toujours positifs.
- La taille de la plus grande multiplication est :

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

  - Elle ne dépasse jamais 32 bits pour $n < 23$.



Testez et optimiser la fonction :

- Ecrire le code correspondant.
- Tester avec différentes valeurs de Fahrenheit.
- Tester avec plusieurs valeurs pour n (16, 18, 20). *N'oubliez pas de recalculer le nombre magique.*
- Tester la fonction avec $n = [22, 23]$, pour $°F = [100, 400, 1000]$.

isa/lab-adv-algos