Université de Franche Comté
Architecture des Ordinateurs - Fiche TP 1

Les TP sont effectués en utilisant Linux.
Vous utiliserez le simulateur `logisim-evolution-3.8.0-all.jar` disponible ici
https ://github.com/logisim-evolution/logisim-evolution#download
Pour lancer le simulateur, depuis un terminal ouvert dans le dossier contenant le fichier ci-dessus, utilisez la commande suivante (java 16 ou supérieur)
`/opt/jdk-17.0.3.1/bin/java -jar logisim-evolution-3.8.0-all.jar`

Exercice 1 — Débuter avec `Logisim`

1. **Tutoriel**
   Effectuez en détail les tutoriels suivants disponible dans le menu `aide`
   — Guide du débutant (Etapes 0 à 4)
   — Design hiérarchique (Créez des circuits et utilisez des sous-circuits)
   — Caractéristiques supplémentaires (Créez des faisceaux de câbles, répartiteurs et couleurs des câbles)
2. **Additionneur**
   Sur le modèle du cours, créez un nouveau projet contenant un demi-additionneur(2 bits), un additionneur complet (3 bits) et un additionneur à propagation ayant deux entrées 4 bits.
3. **Comparateur non signé**
   Avec la table de vérité vue en TD, utilisez l'outil d'analyse pour implémenter le circuit de calcul de `ai` $<$ `bi` (2 bits).

Exercice 2 — Composants du processeur

Vous compléterez les circuits présents dans le fichier `MIPS-ENONCE.circ`

1. `DECODEUR_CO`. Complétez le circuit qui décode les signaux suivant en fonction de l'entrée nommé `Co`. Vous utiliserez des décodeurs 8 bits en cascade comme le suggère le schéma. L'ordre des sorties doit être identique à ceux du tableau suivant (pas de fils croisés). Vous testerez votre circuit en plaçant une valeur sur `Co` et vérifierez que la sortie correspondante est à 1.

`Co`	Sortie Activée
0	zero
3	ijal
4	ibeq
5	ibne
8 ou 9	iaddiu
10	islti

`Co`	Sortie Activée
11	isltiu
12	iandi
13	iori
14	ixori
15	ilui
32 ou 37	ilbu
40 ou 43	isb

   Vous vérifierez dans le circuit de commande `CMD` que l'ordre des sorties est correcte correspondant aux noms des tunnels connectés à ce décodeur.
2. `DECODEUR_NF`. Complétez le circuit qui décode les signaux suivant en fonction de l'entrée nommé `Nf`. Une sortie est à 1 seulement si le signal `en`=1. Pour cela utiliser l'entrée d'activation du décodeur. Vous testerez votre circuit en plaçant une valeur sur `Nf` et vérifierez que la sortie correspondante est à 1.

Nf	Sortie Activée
0	isll
2	isrl
3	isra
4	isllv
6	isrlv
7	israv
8	ijr
32 ou 33	iaddu

Nf	Sortie Activée
34 ou 35	isubu
36	iand
37	ior
38	ixor
39	inor
42	islt
43	isltu

Vous vérifierez dans le circuit de commande `CMD` que l'ordre des sorties est correcte correspondant aux noms des tunnels connectés à ce décodeur.

3. `ENCODEUR`. en utilisant le résultat du TD, complétez le circuit qui place dans un nombre 4 bits les valeurs des indices des entrées `e0` à `e11`. Il y a juste des fils verticaux à placer.
   Attention : Ne pas changer l'ordre des entrées et vérifiez dans le circuit de commande `CMD` la numérotation.
4. **Comparateur à zéro**
   Complétez le circuit `ZERO` qui renvoie 1 si l'entrée 32 bits est à zéro. Vous utiliserez une porte logique 32 bits et un `Répartiteur` ayant un faisceau d'entrée de 32 bits et 32 sorties 1 bit.
5. **UAL**. Complétez le circuit qui réalise les opérations suivantes sur `a` et `b` en fonction de du signal `Op`. Vous utiliserez le circuit `Extension de bit` qui permet de réduire ou d'augmenter la taille d'un faisceau de câble et un `comparateur` (attention à la propriété Type Numérique). Vous utiliserez aussi le circuit ZERO de la question précédente pour déterminer si le résultat de l'UAL est à 0.

Op	Opération
0	A+B
1	A.B
2	A OR B
3	A NOR B
4	A XOR B
5	B SLL (5 bits LSB) A

Op	Opération
6	B SRL (5 bits LSB) A
7	B SRA (5 bits LSB)A
8	A-B
9	A SLL 16
10	1 si A <(signé) B ,0 sinon
11	1 si A <(non signé) B, 0 sinon

Université de Franche Comté
Architecture des Ordinateurs - Fiche TP 2

— Vous utiliserez le compilateur gcc sans option particulière `gcc -Wall ns.c` puis `./a.out`
— Lors de l'accès aux chaînes de caractères, vous utiliserez la notation pointeur à l'exclusion de la notation tableau [].
— Si la chaîne est nommée `s`, vous utilisez `*s` et `s++` pour passer à la position suivante.
— Dans un premier temps vous corrigerez et implémenterez la version du TD avant de passer aux modifications proposées.
— Nous supposerons que les valeurs des paramètres sont correctes (valeurs, adresses chaîne, etc).

Exercice 1 — Fonctions de base

1. **bitCount** En utilisant l'algorithme développé dans les exercices théoriques, implémentez et testez la fonction qui compte le nombre de bits à 1 dans un nombre entier non signé.

```
#include <stdio.h>
unsigned bitCount(unsigned n){
    // A écrire
}
int main(){
   printf("%u \n",bitCount(0xF0F0F0F0)); // affiche  16
   printf("%u \n",bitCount(0x0)); // affiche  0
   printf("%u \n",bitCount(0xFFFFFFFF)); // affiche  32
   return 0;
}
```

   En utilisant l'algorithme développé dans les exercices théoriques, implémentez et testez la fonction qui transforme un **digit hexadécimal** en un caractère le représentant sans utiliser la clause `else`. Nous supposerons que `digit` est compris entre 0 et 0xF. Utilisez le le modèle suivant :

```
char forDigit(unsigned digit){
   char res = (char)(digit& 0xF) + '0';
   if ( digit > ...) // A compléter
      res =  res .... ; // A compléter
   return res;
}
int main(){
   printf("%c \n",forDigit(0)); // affiche '0'
   printf("%c \n",forDigit(9)); // affiche '9'
   printf("%c \n",forDigit(10)); // affiche 'A'
   return 0;
}
```

2. **digit** Modifiez la fonction du TD afin de traiter un caractère (supposé valide lettre en majuscule) en utilisant le principe précédent.

```
unsigned digit(char c){
   unsigned res = (unsigned)c .... ;
   if ( c > ...) // A compléter
```

```
      res =  res .... ; // A compléter
   return res;
}
int main(){
   printf("%u \n",digit(0)); // affiche 0
   printf("%u \n",digit(9)); // affiche 9
   printf("0x%x \n",digit(10)); // affiche 0xa
   return 0;
}
```

Exercice 2 — Chaine

1. **toHexString** Modifiez la fonction du TD afin de parcourir une seule fois les digits du nombre **sans afficher les zéros de gauche**. Ensuite, vous **ajouterez un espace** entre les 4 digits de poids forts et faibles.

```
void toHexString(unsigned n, char *s);
toHexString(0x123,s); // => *s="123\0"
toHexString(0,s); // => *s="0\0"
toHexString(0xABCDE123,s); // => *s="ABCD E123\0"
// Dans le main
int main(){
   char s[33];
   toHexString(0x123,s);
   printf(%s \n",s );
   return 0;
}
```

2. **toBinString** Modifiez la fonction du TD afin parcourir une seule fois les bits du nombre **sans afficher les zéros de gauche** . Ensuite, vous **ajouterez des espaces** entre paquets de 4 bits.

```
void toBinString(unsigned n, char *s);
toBinString(0x123,s); // => *s="100100011\0"
toBinString(0,s); // => $s="0\0"
toBinString(0x123,s); // => *s="1 0010 0011\0"
```

3. **toUnsignedStringBase** Modifiez la fonction du TD afin de parcourir **une seule fois les digits du nombre puis en inversant le contenu de la chaine** résultat. La base est supposée correcte. Ensuite, vous **ajouterez des espaces** entre les paquets de 3 chiffres.

```
void toUnsignedStringBase(unsigned n,char *s,  unsigned base);
toUnsignedStringBase(0,s,10); // => *s="0\0"
toUnsignedStringBase(0x123,s,10); // => *s="291\0"
toUnsignedStringBase(0x123,s,10); // => *s="291\0"
toUnsignedStringBase(0x123,s,16); // => *s"123\0"
toUnsignedStringBase(0xFFFF,s,10); // => *s="65 535\0"
```

4. **toStringUnsigned** Implémenter cette fonction utilisant les trois fonctions précédentes selon la valeur de la base (supposée valide).

```
toStringUnsigned(unsigned n, char *s,  unsigned base);
```

Exercice 3 — Non signé

1. **parseHex** Modifiez la fonction du TD. La chaîne de caractères est supposée valide **pouvant contenir des espaces**.

```
unsigned  parseHex(char *s);
parseHex("1 ABC9"); // => retourne 109513
parseHex("0"); // => retourne 0
```

2. **parseBin** Modifiez la fonction du TD. La chaine de caractères est supposée valide **pouvant contenir des espaces**.

   ```
   unsigned  parseBin(char *s);
   parseBin("1 1001"); // => retourne 25
   parseHex("0"); // => retourne 0
   ```

3. **parseBase** Modifiez la fonction du TD. La chaine de caractères et la base sont supposés valides **pouvant contenir des espaces**.

   ```
   unsigned  parseBase(char *s,unsigned base);
   parseBase("1 291",10); // => 1291
   parseBase("123",16); // => 291
   ```

4. **parseUnsigned** Implémenter cette fonction utilisant les trois fonctions précédentes selon la valeur de la base (supposée valide).

   ```
   unsigned  parseUnsigned(char *s,unsigned base);
   ```

Université de Franche Comté
Architecture des Ordinateurs - Fiche TP 3

Le fichier sera nommé `s.c`

Exercice 1 — Fonctions de base

1. **bitCount** Implémenter la fonction modifiée. Attention au décalage à droite !

   ```
   int bitCount(int n);
   bitCount(0x80000000)=> 1
   bitCount(-1)=>32
   ```

2. **absolue** Implémenter la fonction sans utiliser l'alternative

   ```
   int absolue(int n);
   ```

Exercice 2 — Chaine de caractères / Entiers

1. **toString** Implémenter la fonction vue en TD en parcourant **une seule fois** les digits du nombre puis en inversant le contenu de la chaine résultat en utilisant la division entière. La base est supposée correcte. **Attention** : Vérifiez que cela fonctionne pour les nombres 2^{31-1} et 2^{31}, pour cela utiliser un nombre négatif dans l'itération.

   ```
   void toStringBase(int n,char *s,  int base);
   toStringBase(0,s,10); // => *s="0\0"
   toStringBase(0x123,s,10); // => *s="291\0"
   toStringBase(0xFFFFFFFF,s,10); // => *s="-1\0"
   toStringBase(0x80000000,s,16); // => *s="-2147483648\0"
   toStringBase(0x7FFFFFFF,s,16); // => *s="2147483647\0"
   ```

   Vous pouvez aussi ajouter les caractères de séparation '_' entre les paquets de 3 chiffres

2. **parseInt** Implémenter la fonction vue en TD en utilisant les opérateurs arithmétiques. La chaine et la base sont supposés valides. **Attention** : Vérifiez que cela fonctionne pour les nombres 2^{31-1} et 2^{31} en utilisant un nombre négatif dans l'itération.

   ```
   int  parseInt(char *s,int base);
   parseInt("+291",10); // => 291
   parseInt("-123",16); // => -291
   parseInt("-2147483648",s,10); // => -2147483648
   parseInt("2147483647",s,10); // => 2147483647
   ```

Université de Franche Comté
Architecture des Ordinateurs - Fiche TP 4

Le fichier sera nommé `r.c`

Exercice 1 — Codage des nombres réels simple précision

1. Expliquez le fonctionnement des changement de types (*(unsigned*)(&f) et (*(float*)(&i)
2. **Vérifications des valeurs théoriques** Vérifiez les valeurs hexadécimales obtenues pour les nombres 125.25, 0.375,−0.1. Vous remarquerez que la valeur hexadécimale représentant 0.1 est arrondie à la valeur supérieure à celle de l'exercice théorique. Expliquez cette différence.
3. **Valeur maximale** Donnez la valeur maximale que nous pouvons ajouter à la valeur 1.0f pour obtenir un résultat correct. Vous procéderez par incrémentation de cette valeur.
4. **Codage** Expliquez pourquoi nous obtenons ces résultat pour les expressions suivantes.

```
printf("%d",(0.25+0.5== 0.75))=> ?
printf("%d",(0.2+0.1== 0.3))= > ?
printf("%d",(0.2+0.1-0.3< 0.0000000001)) =>
float z=0;
printf("%f",(1.0/z)) =>
printf("%f",(1.0/z)+(-1.0/z)) =>
```

Exercice 2 — Manipulation des nombres réels normalisés

1. Opérateur inférieur Implémenter la fonction **inf** qui retourne 1 si |f1| < |f2|, f1 et f2 étant des nombres réels, sans utiliser l'opérateur relationnel sur les nombres réels normalisés, mais en l'utilisant sur les nombres entiers.

```
unsigned inf(float f1,float f2)
inf(-1.5,2.75) => 1
inf(1.5,2.75) => 1
inf(-2.75,1.5) => 0
```

2. **Obtenir signe, mantisse et exposant**
   Implémentez les trois fonctions permettant de renvoyer respectivement le signe (0 ou 1), l'exposant réel (compris entre -126 et + 127) et la mantisse incluant le bit de normalisation à gauche de la virgule.

```
unsigned getSigne(float reel);
unsigned getExposant(float reel);
unsigned getMantisse(float reel);
```

3. **Modifier signe, mantisse et exposant**
   Implémentez les trois fonctions permettant de modifier le signe, l'exposant et la mantisse du paramètre. Utilisez ces fonctions pour recomposer la valeur représentant le nombre 125.25.

```
float setSigne(float reel,unsigned  signe);
float setExposant(float reel,unsigned  exposant);
float setMantisse(float reel,unsigned mantisse) ;
```

4. **Opérateur d'addition** Implémentez la fonction **add** qui retourne |f1|+|f2|, f1 et f2 étant deux nombres réels normalisés (sans utiliser l'addition des réels, ni la fonction `fabs`. Le résultat sera normalisé aussi.

```
float add(float f1,float f2)
addAbs(-1.5,2.75) => 4.25
addAbs(-1.5,-2.75) => 4.25
```

Exercice 3 — Représentation d'un nombre réel normalisé simple précision

1. **toHexString** Complétez le code de la fonction permettant de transformer un nombre réels en sa représentation hexadécimale. Vous utiliserez la fonction de la bibliothèque standard **sprintf** qui fonctionne comme **printf** mais place le résultat dans une chaîne de caractères.

```
void toHexString(char *r,float f)
toHexString(*r,125.25f) = > *r="+1.f50000p6"
toHexString(*r,0.375f) = > *r="+1.800000p-2"
toHexString(*r,-0.1f) = > *r="-1.99999ap-4"
```

```
void toHexString(char *res,float f){
   unsigned n = ...
   sprintf(res,"%c1.%xp%d", (n >> 31) ? ... : ... ,(n... ) << 1 ,
               ((n....) ... )- ...);
}
```

2. **parseFloat** Complétez le code d'une fonction permettant de transformer la représentation hexadécimale d'un réel normalisé supposée correct en un nombre réel. Vous utiliserez la fonction de la bibliothèque standard **sscanf** qui fonctionne comme **scanf** mais lit dans chaîne de caractères.

```
void toHexString(char *)
parseFloat( "+1.F50000p85") => 125,25
parseFloat(+1.800000p7D")=>0.375
parseFloat( "-1.99999Ap7B")=>-0.1
```

```
float parseFloat(char *ch){
   unsigned res=0,e,m;
   char s;
   sscanf(ch,"%c1.%xp%d",&s,&m,&e);
   res =  (s=='-') ? ... : ....,   (e+...)... , m >> ...);
   return ....;
}
```