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 intégrés suivants :
   — 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. **Décodeur CO**. Complétez le circuit qui décode les signaux suivant en fonction de l'entrée nommé `Co`.

`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.
2. **Décodeur NF**. Complétez le circuit qui décode les signaux suivant en fonction de l'entrée nommé `Nf`. Une sortie est validée seulement si le signal `en`=1. Pour cela utiliser l'entrée d'activation du décodeur.

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.

3. **ENCODEUR**. Complétez le circuit qui place dans un nombre 4 bits les valeurs des indices des entrées `e0` à `e11`.
   Attention : Ne pas changer l'ordre des entrées.
4. **Comparateur à zéro**
   Complétez le circuit ZERO qui renvoie 1 si l'entrée 32 bits est à zéro. Vous utiliserez une porte logique et un `Répartiteur`.
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** Implémenter la fonction vue en TD.

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

2. **forDigit** Modifiez la fonction du **afin ne pas utiliser le mot réservé** `else` suivant le modèle suivant :

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

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

```
unsigned digit(unsigned char n){
   // A écrire
}
// Dans le main
printf("%u \n",digit('9')); // affiche 9
printf("0x%x \n",digit('A')); // affiche 0xA
```

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
void main(){
   char s[33];
   toHexString(0x123,s);
   printf(%s \n",s );
}
```

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 chaine 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 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 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);
```