Architecture matérielle : Simulation d'un processeur dans Diglog

Novembre 2022

Table des matières

1 Introduction **2**

2 Exercice1 **3**
2.1 . . . 3
2.2 . . . 3
2.3 . . . 3
2.4 . . . 3
2.5 . . . 3
2.6 . . . 3

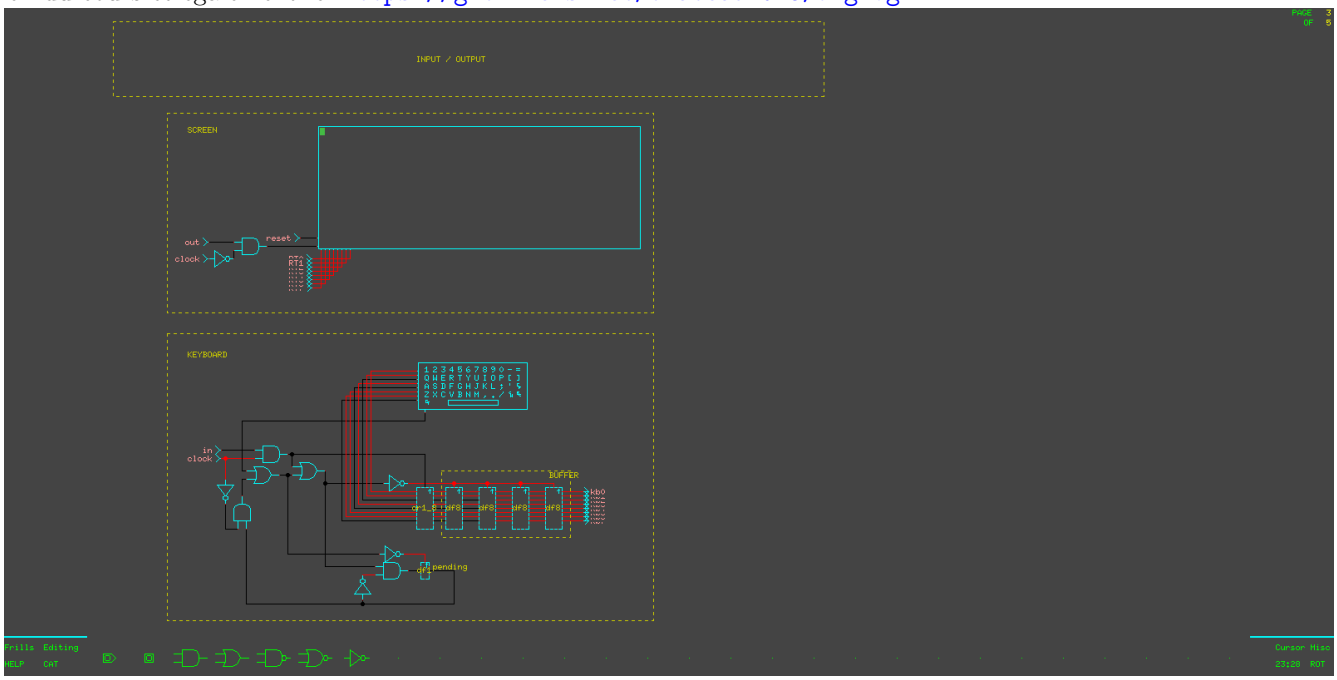
3 Exercice2 **4**
3.1 . . . 4
3.2 . . . 4

4 Exercice3 **5**
4.1 . . . 5
4.2 . . . 5
4.3 . . . 5
4.4 . . . 5
4.5 . . . 5
4.6 . . . 5
4.7 . . . 6

5 Exercice4 **7**
5.1 . . . 7

1 Introduction

Le but de ce projet est de construire un mini processeur digproc, avec l'outil diglog disponible sur le lien du cours et également ici https://git.iiens.net/brateau2015/diglog.



On lancera le processeur, après compilation avec la commande :

```
./diglog digproc/*lgf&
```

Je m'étais arrêté à la question 2.2 après la fin du TP5, et n'ai pas pris beaucoup de notes.

2 Exercice1

2.1

```
r0=59,r1=42
```

2.2

```
ldi r1, 42
addi r0, r1, 17
end: jmp end
```

```
cat exo1.asm.hi
0000:09 #9 en hexa = 1001 en binaire
0001:40 #64 = 1100100
0002:e0 #224 = 11100000
```

```
cat exo1.asm.lo
0000:2a #42 en hexa = 101010
0001:31 #49 = 110001
0002:02 #2 = 10
```

2.3

```
./digcomp exo1.asm
end = 2
--------
   0: r1 <- 42
   1: r0 <- r1 + 17
   2: goto end
```

2.4

```
./diglog digproc/*.lgf (puis -) -> reset 1 cycle , mettre en rouge

lo: 101010 -> 2a

Manipulations:
clic droit dans la boite en haut -> save .lo et .hi respectivement
reset 1 fois + clic bouton bas clock (2 * = 1 instruction)
r sultat addition -> fils alu 8

11011100 // 59=00111011
```

2.5

inst	do_jmp_abs	write_reg	arg2_imm	res_imm
ldi	0	1	x	1
addi	0	1	1	0
jmp	1	0	x	x
nop	0	0	x	x

2.6

fichiers langage.pdf jmp : 111 : 3premiers fils

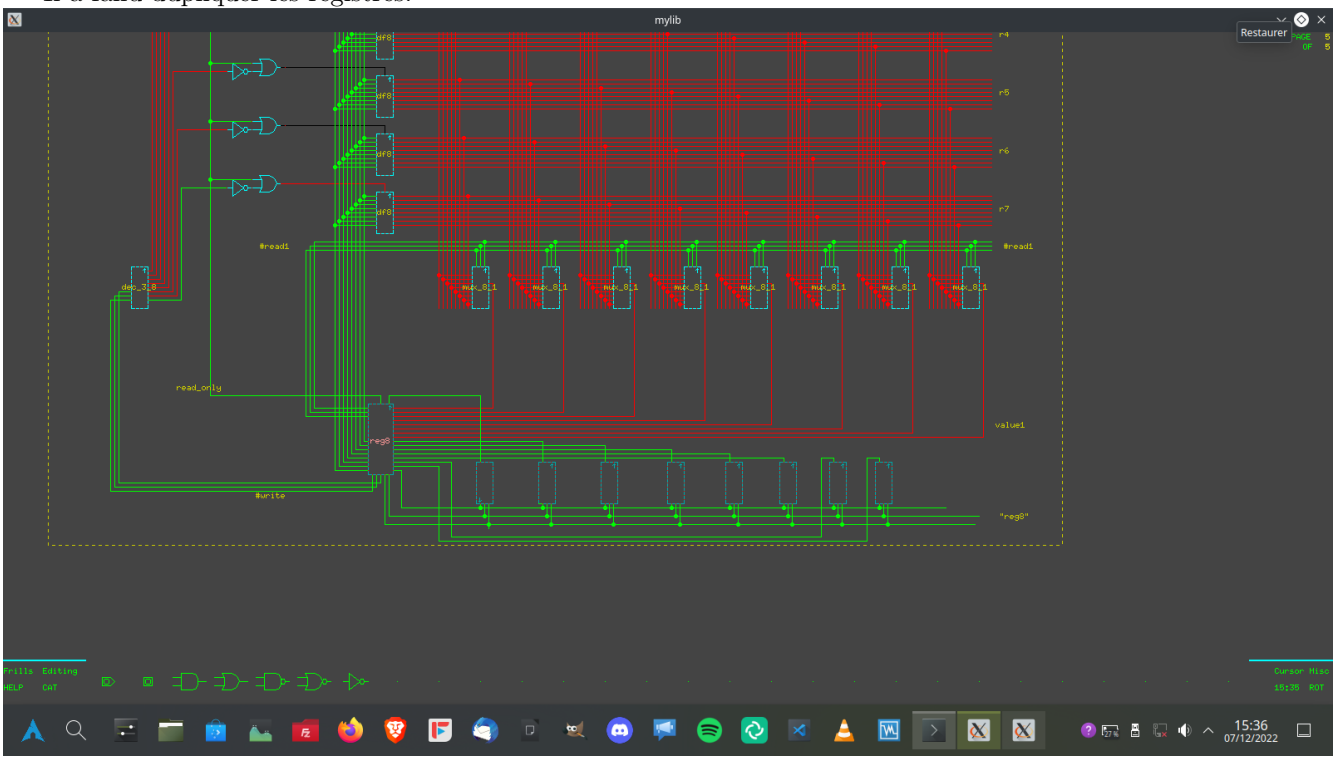
3 Exercice2

3.1

Sub : prendre en compte 5 = (2+3) cas selon rd = ou ¡= ou ¡rs etc-¿ add5 Finir de rajouter un 2nd banc de registres

3.2

Il a fallu dupliquer les registres.



4 Exercice3

On effectue les opérations sur diglog.

4.1

4.2

4.3

4.4

4.5

4.6

On peut effectuer le code suivant

```
%token <int> INT
%token NOP MOV ADD ADDI SUB SUBI JMP LD ST IN OUT JLE JLT JEQ JNE
%token COMA COLON LPAR RPAR
%token <int> REG
%token <string> LABEL
%token EOL

%start main
%type <(Asm.label * Asm.instr option)> main
%type <Asm.cond> cjump

%%

main:
    LABEL COLON                   { ($1, None) }
  | EOL                           { ("", None) }
  | code EOL                      { ("", Some $1) }
  | code                          { ("", Some $1) }
;

code:
  | NOP                            { Nop }
  | MOV REG COMA INT               { Ldi ($2,$4) }
  | MOV REG COMA REG               { Addi ($2,$4,0,false)  }
  | ADD REG COMA REG COMA REG      { Add  ($2,$4,$6,false) }
  | ADDI REG COMA REG COMA INT     { assert (0<=$6 && $6<32); Addi ($2,$4,$6,false) }
  | ADD REG COMA REG COMA INT      { assert (0<=$6 && $6<32); Addi ($2,$4,$6,false) }
  | ADD REG COMA INT COMA REG      { assert (0<=$4 && $4<32); Addi ($2,$6,$4,false) }
  | SUB REG COMA REG COMA REG      { Add  ($2,$4,$6,true)  }
  | SUB REG COMA REG COMA INT      { assert (0<=$6 && $6<32); Addi ($2,$4,$6,true)
}
  | SUBI REG COMA REG COMA INT     { assert (0<=$6 && $6<32); Addi ($2,$4,$6,true)
}
  | LD REG COMA REG                { Load ($2, $4)  }
  | MOV REG COMA LPAR REG RPAR     { Load ($2, $5)  }
  | ST REG COMA REG                { Store ($4, $2) }
  | MOV LPAR REG RPAR COMA REG     { Store ($3, $6) }
  | IN REG                         { In $2  }
  | OUT REG                        { Out $2 }
  | cjump REG COMA REG COMA LABEL { CJmp ($2,$4,$6,$1) }
  | JMP LABEL                      { Jmp $2 }
```

```
;
```

4.7

Pour tenir en compte du caractère :

```
cjump:
  | JLE  { LE }
  | JLT  { LT }
  | JEQ  { EQ }
  | JNE  { NE }
;
```

5 Exercice4

5.1

On a ici le support des instructions load et store avec le nouveau bit de contrôle write_mem