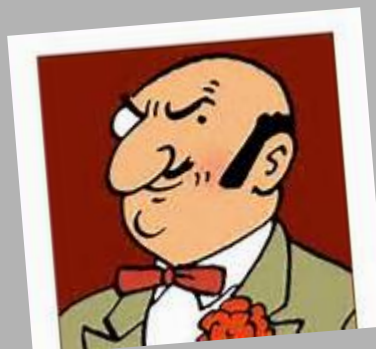




NOTE CONFIDENTIELLE

MESSAGE AGENT 008 :

Détection sur chambre de compensation
Clearflow mouvement 50 Milliards USD par
Dimitri Rastapopouline - STOP - Objectif :
OPA hostile sur société AMR-corp. - STOP-
Probable perte d'accès France / Europe sur
technologie CPU Curtex -FIN-



D.Rastapopouline

RELEVÉ DE DÉCISION COMITÉ STRATÉGIQUE DNSN`

(Présents Professeur Nestor Halambique, docteur Jonathan Septimus,
Professeur Tryphon Tournesol, Professeur Hippolyte Bergamotte.)


Contexte : Risque maximum de perte de souveraineté sur technologie
numérique clé CPU 32 bit AMR-Curtex. - source agent 008.

Recommandation : Mettre en place et sécuriser technologie alternative base
RISC-V avant OPA sur société AMR.

Phase 1 : Identifier / former des formateurs filières technologiques sur
écosystème RISC-V en environnement libre matériel / logiciel : linux/GCC/
Make/iVerilog/RISC-V/Picorv32.

Opérateur phase 1 : IUT, département Geii. Proposition en attente.

- Jalon 1 : Compilation, simulation, synthèse projet risc-v sans modification
- Jalon 2 : Démonstration chaine de compilation logicielle GCC/Make sur code original
- Jalon 3 : intégration d'un périphérique fourni sur bus risc-v et développement test logiciel

 université PARIS-SACLAY <hr/> IUT DE CACHAN	<p>Université Paris-Saclay — IUT de CACHAN</p> <p>BUT-GEii</p> <p>Année 2024/2025 — semestre S3</p> <p>Développement en environnement libre pour CPU RISC-V</p> <p>Cahier de Charge de la formation</p> <p>1er septembre 2024</p>	<p>CdC</p> <p>BUT-GEii</p> <p>RISC-V</p> <p>J.O.Klein</p>
---	---	--

En réponse à la demande de la *Direction Nationale de la Souveraineté Numérique (DNSN)*, le département GEii-1 de l'IUT de Cachan propose une formation sur 4 demi-journées, caractérisée par les objectifs pédagogiques suivants :

Acquis d'Apprentissage Visés (AAV) :

A l'issue de la formation, tous les étudiants seront capables de :

- **AAV1** : Développer du code en C dans un environnement libre (sous linux en mode terminal, avec l'utilitaire make/makefile, et un compilateur croisé GCC pour risc-v).
- **AAV2** : Interfacer un périphérique sur le bus d'un softcore RISC-V (par ex. picorv32) décrit en verilog et le programmer en C.
- **AAV3** : Simuler un système modélisé en verilog en utilisant un logiciel libre (iverilog), sous linux.
- **AAV4** : Réaliser la synthèse d'un softcore RISC-V pour un FPGA, avec sa mémoire (RAM et /ou ROM) et son contenu (code binaire).

Calendrier

Jour 1 :

Lundi 16 septembre 2024 de 8h à 11h45

- En équipe : analyser les documents fournis, lister (sur un paperboard) les mots-clés connus et inconnus, les questions en suspens et les ressources pour y répondre.
- En binôme ou trinôme : démontrer la simulation verilog d'un système à base de RISC-V.
- En binôme ou trinôme : Démontrer la synthèse d'un système à base de RISC-V sur un FPGA.

Inter-séance 1-2 (livrable) :

- Individuellement : rédiger un tutoriel sur le langage verilog (notamment les différences avec le langage VHDL) et sur la simulation avec l'outil iverilog (et les différences avec l'outil ModelSim).

Jour 2 :

Lundi 23 septembre 2024 de 8h à 11h45

- En équipe: Produire son propre code C montrant l'accès à des périphériques, en modifiant leurs adresses. Adapter le code verilog aux nouvelles adresses choisies.
- En binôme ou trinôme : Tester le code C de l'équipe et montrer par simulation son bon fonctionnement.
- En équipe : Se répartir dans les 5 groupes thématiques pour l'interséance 2-3.

Inter-séance 2-3 :

- Produire et présenter un résumé de cours sur un des 5 thèmes : 1. linux en mode terminal, 2. utilitaire make, 3. verilog, 4. bus système du picorv32, 5. l'accès aux périphériques mappés en mémoire en langage C.

Jour 3 :

Lundi 30 septembre 2024 de 8h à 11h45

- En groupe thématique : préparer un cours (sur un paperboard) sur le thème.
- En groupe thématique : Présenter le cours préparé à l'ensemble du groupe.
- Evaluation formative (blanche) sur feuille de l'**AAV2** : « Interfacer un périphérique sur le bus d'un softcore RISC-V (par ex. picorv32) décrit en verilog et le programmer en C . »

Interséance 3-4 :

- Finaliser la réponse détaillée à l'évaluation formative.

Jour 4 :

Lundi 7 octobre 2024 de 8h à 11h45

- Correction de l'évaluation formative.
- **Evaluation pratique individuelle** : Produire son propre code C montrant l'accès à des périphériques, en utilisant leurs adresses. Adapter le code verilog aux adresses choisies. Tester le code C et montrer par simulation son bon fonctionnement. Réaliser la synthèse sur FPGA.
- Bilan de la formation.

Documents ressources :

Document 1 : L'interface bus du softcore RISC-V picorv32.

Document 2 : Les cycles bus du softcore RISC-V picorv32.

Document 3 : L'accès aux registres en mémoire en langage C.

Document 4 : Le fichier makefile

Document 5: Cours d'introduction au langage Verilog,

Document 6: Le système complet et son décodage d'adresse, en verilog

Document 7: Le schéma du système complet

L'ensemble de ces documents sur github :

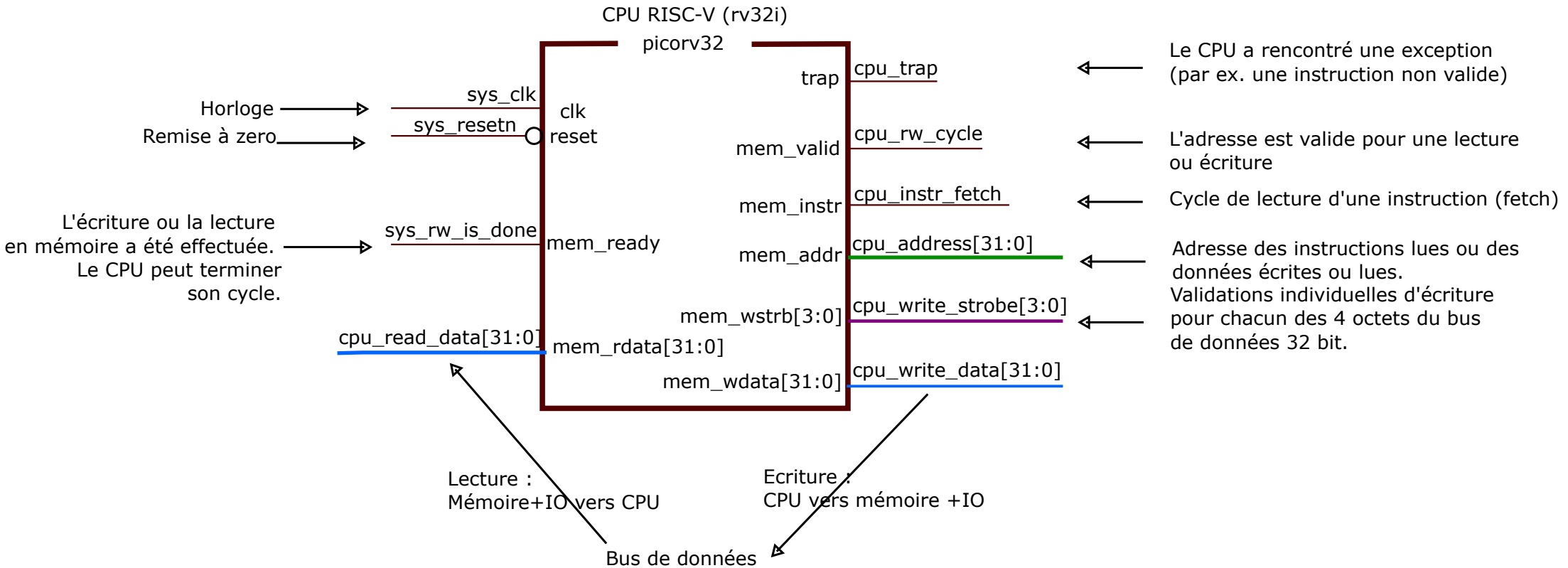
https://github.com/JOKleinGe1/Module_Initiation_Riscv.git

Démonstration de la simulation en vidéo : <https://youtu.be/eN36onBk7ro>

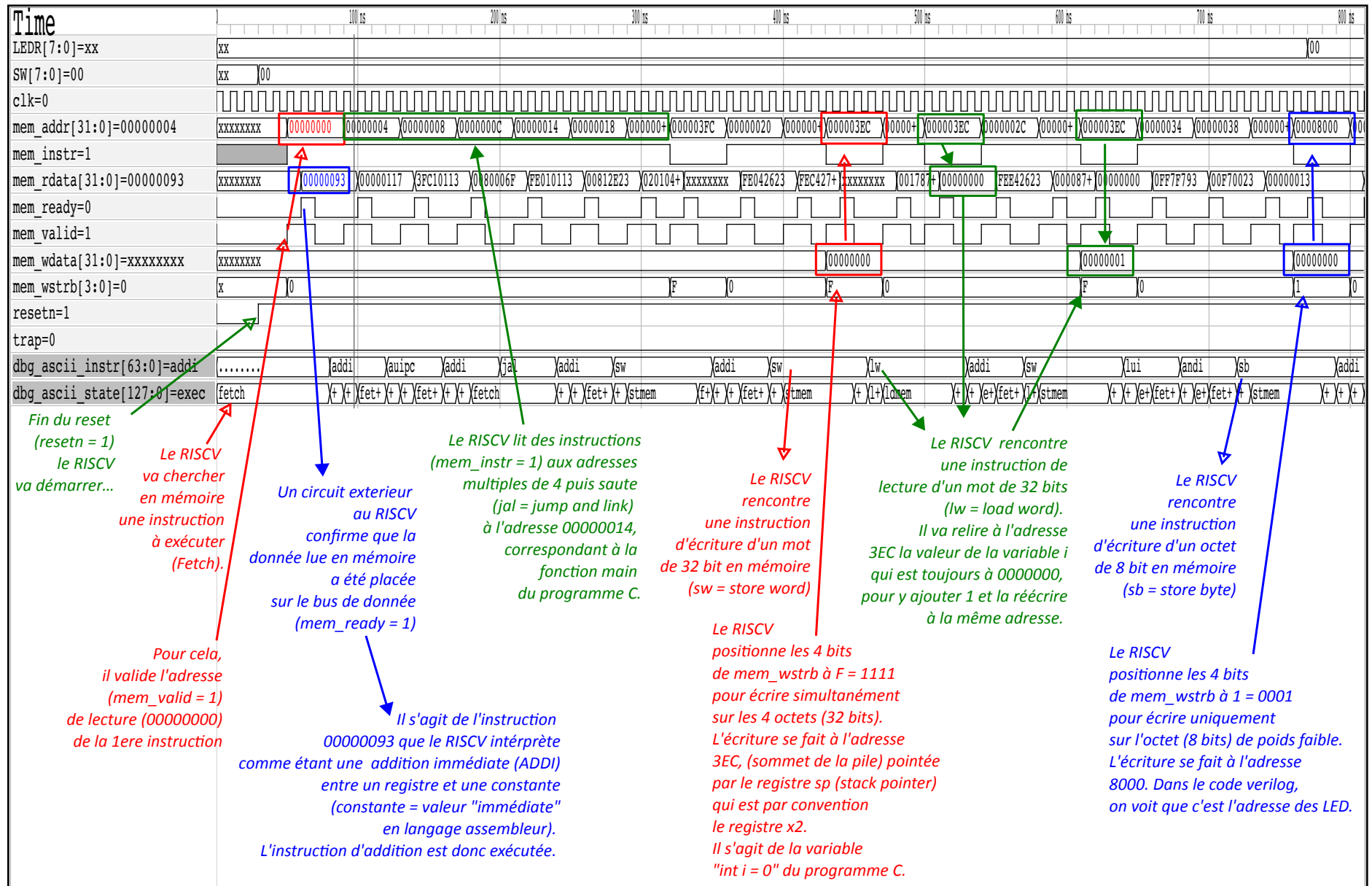
Codes sources

- Pour la simulation seulement (icarus verilog) :
https://github.com/JOKleinGe1/min_sys_riscv.git
- Pour la simulation (icarus verilog) et la synthese avec quartus (intel-fpga) :
https://github.com/JOKleinGe1/riscv_quartus.git

Document 1



Document 2



Document 3

En langage C, l'accès à un registre **mappé** en mémoire (qui a une adresse dans l'espace mémoire), se fait en utilisant un pointeur. On part de l'adresse donnée explicitement par une constante entière, généralement en codage hexadécimal, ici en vert **0x8000** ou **0x8004**.

La taille du registre (8, 16 ou 32 bits) définit le type de base du langage C qui doit être utilisé (unsigned char, short, int, ou mieux **uint8_t**, **uint16_t**, **uint32_t** définis dans **<stdint.h>**). A partir de ce type de base, il faut construire un pointeur, par ex. ici (**uint8_t ***), c'est l'objet de l'opération de **cast** (changement de type) en bleu dans le code.

Le mot clé « **volatile** » précise au compilateur qu'il ne s'agit pas réellement d'une mémoire, la donnée écrite n'est pas réellement mémorisée, il faut donc que tous les accès en lecture et en écriture soit effectivement réalisés et éviter des optimisations du compilateur qui pourraient en supprimer certains.

Pour accéder au registre, ce n'est pas l'adresse qu'on souhaite manipuler, mais la données qui s'y trouve, il faut donc **déréférencer** le pointeur (accéder à la donnée qui se trouve à cette adresse) avec l'opérateur de **déférence *** (en rouge ici).

Finalement, il reste à définir une **constante symbolique** (ici en **orange**) pour pouvoir accéder facilement au registre en écriture (dans une affectation) ou en lecture (dans une condition, un test).

Les adresses sont des multiples de 4 pour simplifier l'accès aux registres en un cycle car le CPU possède un bus de données de largeur 32 bits (4 octets).

```
//test.c
#include <stdint.h>

#define LEDADDRESS (*(volatile uint8_t *) 0x8000)
#define SWADDRESS (*(volatile uint8_t *) 0x8004)

int main (void){
    int i=1;
    while (1){
        LEDADDRESS = i++;
        while (SWADDRESS & 0x01);
    }
    return 0;
}
```

Écriture des LED à l'adresse 0x8000

Lecture des switches à l'adresse 0x8004

Makefile

Dans un environnement de développement linux (en mode terminal), on compile un programme avec la commande **make [cible]**. Le détail des règles pour la compilation d'une cible est donné dans le fichier **makefile**, suivant la syntaxe :

[CIBLE] : [DEPENDANCE1] [DEPENDANCE2] ...

[CARACTERE DE TABULATION ->] [Règle(=commande) à exécuter pour créer la CIBLE]

- Au début du fichier makefile, on peut trouver la définition de variables pour les commandes de compilation standard (comme CC pour le compilateur C). On accède à sa valeur avec le symbole \$ (par ex. **\$(CC)**).
- Le symbole **\$@** représente la cible et le symbole **\$\$** les dépendances.
- En l'absence de cible précisée dans la commande make, la cible par défaut est « **all** ».
- Lorsqu'une dépendance est absente (le fichier n'existe pas), le makefile va chercher s'il y a une règle pour le construire. Lorsqu'une dépendance existe, il ne la reconstruit que si c'est nécessaire, lorsqu'une des dépendances a été modifiée (en comparant les dates de création des fichiers).

```
CC := /opt/riscv32i/bin/riscv32-unknown-elf-gcc
NM:= /opt/riscv32i/bin/riscv32-unknown-elf-nm
OBJCOPY:= /opt/riscv32i/bin/riscv32-unknown-elf-objcopy
OBJDUMP:= /opt/riscv32i/bin/riscv32-unknown-elf-objdump
```

Chemins vers le compilateur (CC) et les outils de manipulation des fichiers objets et binaires

```
all : test.asm test.map test.mem32 tb_sys_picorv32.vcd
```

La cible "**all**" définit tout ce qui doit être reconstruit quand on lance la commande **make**.

```
test.hex: jumpstart.s test.c
[TAB -> ] $(CC) -o $@ $$ -T linker.lds -nostartfiles
```

Commande principale de compilation \$(CC) avec les options pour le fichier de commande du linker (-T). Le fichier **jumpstart.s** est notre fichier de startup, on ne veut pas ceux par défaut (-nostartfiles).

```
test.map : test.hex
[TAB -> ] $(NM) $$ > $@
```

Crée un fichier (.map) contenant la cartographie de la mémoire après compilation et édition de lien (linker).

```
test.mem : test.hex
[TAB -> ] $(OBJCOPY) -O verilog $$ $@
```

Transforme le fichier binaire standard (.hex) en fichier binaire (.mem) compatible avec la commande verilog **\$readmemh**.

```
test.mem32 : test.mem VlogMem8to32
[TAB -> ] ./VlogMem8to32 <test.mem > test.mem32
```

Réorganise le fichier verilog (.mem) pour que les données soient présentées en mots de 32 bits (mem32), avec le programme "maison" **VlogMem8to32.c**

```
test.mif : test.mem VlogMem8to32
[TAB -> ] ./VlogMem_to_QuartusMIF <test.mem > test.mif
```

Réorganise le fichier verilog (.mem) pour que les données soient compatibles avec la synthèse dans Quartus (intelFpga) en utilisant le programme "maison" **VlogMem_to_QuartusMIF.c**

```
VlogMem8to32 : VlogMem8to32.c
[TAB -> ] cc -o $@ $$
```

```
VlogMem_to_QuartusMIF : VlogMem_to_QuartusMIF.c
[TAB -> ] cc -o $@ $$
```

Compilation des outils "maison" utilisés pour réorganiser le fichier binaire

```
test.asm : test.hex
[TAB -> ] $(OBJDUMP) $$ -d > $@
```

Désassemblage du fichier binaire (.hex) pour re-crée un fichier en langage assembleur (.asm).

```
tb_sys_picorv32.vvp : picorv32.v system_picorv32.v tb_sys_picorv32.v
[TAB -> ] iverilog -o $@ $$
```

Compilation avec **iverilog** des fichiers verilog, y compris le testbench. Le modèle simulable avec **vvp** est écrit dans le fichier de sortie (.vvp).

```
tb_sys_picorv32.vcd :tb_sys_picorv32.vvp
[TAB -> ] vvp $$
```

Lancement de la simulation verilog (**vvp**). Les signaux sont enregistrés dans le fichier (.vcd) que l'on peut afficher avec l'utilitaire **gtkwave**

```
clean :
[TAB -> ] /bin/rm -f test.hex *.mem *.map *.elf *.hex tb_picorv32
rm *.vcd *.asm *.mem32 VlogMem8to32 *.vvp
```

make clean permet de supprimer (**rm** = remove) tous les fichiers créés par **make all**.

Document 6

Interface du système
(I/O de la DE10-lite)

Signaux pour connecter
le coeur du CPU

Signaux pour connecter
la RAM et les I/O

Signaux pour sélectionner
la RAM et les I/O

Multiplexeur entre RAM data
et I/O data (en lecture)

Génération de la confirmation
de fin d'écriture ou de lecture
(sys_rw_is_done)

Instance du CPU RISC-V
(picorv32)

Instance de la RAM 1 K octets
(ram1port 256 x 32 bits)

Bloc procédural combinatoire
"@(*tous les signaux d'entrée*)"

Décodage d'adresse : génération
des signaux de sélection en
lecture et écriture dans
la RAM et les périphériques

Bloc procédural synchrone
"@(*posedge sys_clk*)"
pour l'écriture et lecture dans
les périphériques (LED et boutons)

le signal 8 bit **LEDR**
type "**reg**" donc défini
dans un bloc
procédural
(avec un @)

Bus sur 8 bits

//file : system_picorv32.v

```
module system_picorv32 (input sys_clk,sys_resetsn,output reg [7:0] LEDR,input [7:0] SW);
  wire  cpu_trap;
  wire  cpu_rw_cycle;
  wire  cpu_instr_fetch;
  reg   sys_rw_is_done;
  wire [31:0] cpu_address;
  wire [31:0] cpu_write_data;
  wire [3:0]  cpu_write_strobe;
  wire [31:0] cpu_read_data;
  reg   [31:0] io_read_data;
```

Dans ce contexte, "|" est un opérateur unaire
de **réduction**, C'est le **OU** de tous les bits (4)
du bus *cpu_write_strobe*.

```
  wire  sys_write_enable = (! cpu_write_strobe); // if one or more byte written
  wire  sys_read_enable  = cpu_rw_cycle & (! sys_write_enable);

  wire [31:0] ram_read_data;
```

```
  // individual read_enable and write_enable
  wire sys_RAM_read, sys_RAM_write; //RAM-1KB:0-0x3FF(byte) = 0xFF(32-bit-word)
  wire sys_LED_read, sys_LED_write;
  wire sys_SWITCH_read;
```

```
  reg [4:0] sys_rw_bus ; // grouping 5 individual read_enable and write_enable in a bus
  wire global_rw = |(sys_rw_bus) ; // if any individual r/w is enable
```

```
  // degrouping individual read_enable and write_enable
  assign (sys_SWITCH_read, sys_LED_write,sys_LED_read, sys_RAM_write, sys_RAM_read)
    = sys_rw_bus;
```

```
  // select if data bus is read from RAM or IO
  assign cpu_read_data = (sys_RAM_read) ? ram_read_data : io_read_data ;
```

```
  // Acknowledge read/write request cpu one cycle after cpu_rw_cycle using
  // sys_rw_is_done if cpu address is in ram or for any I/O registers
  always @(posedge sys_clk) sys_rw_is_done <= (global_rw && !sys_rw_is_done) ;
```

```
  // instance of RISC-V
  picorv32 uut (
    .clk      (sys_clk      ),
    .resetsn  (sys_resetsn ),
    .trap     (cpu_trap     ),
    .mem_valid (cpu_rw_cycle ),
    .mem_instr (cpu_instr_fetch ),
    .mem_ready (sys_rw_is_done ),
    .mem_addr  (cpu_address ),
    .mem_wdata (cpu_write_data ),
    .mem_wstrb (cpu_write_strobe ),
    .mem_rdata (cpu_read_data )
  );
```

Connexions des signaux internes,
précédés d'un ".", aux signaux externes
(définis au dessus) entourés
de parenthèses.

```
  // instance RAM (1KB)
  ram1port ram1port_inst (
    .address ( cpu_address[9:2] ),
    .byteena ( cpu_write_strobe ),
    .data     ( cpu_write_data ),
    .clock    ( sys_clk ),
    .rden     ( sys_RAM_read ),
    .wren     ( sys_RAM_write ),
    .q        ( ram_read_data )
  );
```

Utilisation de l'instruction
casex (correspondance avec des
0 des 1 et des **X**=don't care)
portant sur un bus dont les signaux
sont regroupés avec l'opérateur **{,}**
de **concaténation**.

```
  // r/w individual signal generation from address decoding
  always @((sys_read_enable,sys_write_enable,cpu_address))
  casex ({sys_read_enable,sys_write_enable,cpu_address})
    //ram read 0-3FF :
    {2'b10,32'b00000000_00000000_000000xx_00000000}:sys_rw_bus <= 5'b000001;
    //ram write 0-3FF :
    {2'b01,32'b00000000_00000000_000000xx_00000000}:sys_rw_bus <= 5'b00010;
    {2'b10,32'h0000_8000} : sys_rw_bus <= 5'b00100; //led read @ 8000
    {2'b01,32'h0000_8000} : sys_rw_bus <= 5'b01000; //led write @ 8000
    {2'b10,32'h0000_8004} : sys_rw_bus <= 5'b10000; //switch read @ 8004
    default                : sys_rw_bus <= 5'b00000;
  endcase

  always @(posedge sys_clk) begin
    if (sys_LED_read) io_read_data <= { 24'd0 , LEDR };
    else if (sys_LED_write & cpu_write_strobe[0]) LEDR <= cpu_write_data[ 7: 0];
    else if (sys_SWITCH_read) io_read_data <= { 24'd0 , SW };
  end
```

endmodule

Constante numérique zéro (**0**) de **24** bits,
ici écrite en décimal (**d**).
Ces constantes peuvent aussi s'écrire
en hexa (h) ou binaire (b).

Document 7

