Document 3

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 le 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) $^  > $@
```

Créé 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 $@ $^
```

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

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

Désassemblage du fichier binaire (.hex) pour re-créer 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 (.vpp).

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

Lancement de la simulation verilog (***vpp***). 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.***