HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Prof. Yann Thoma

Laboratoire CSF
semestre de printemps 2020 - 2021

Emetteur Morse

Composant à tester

Nous souhaitons tester un système permettant l'envoi de code Morse pour les caractères alphanumériques standards [1]. Le composant accepte des codes ASCII en entrée et est capable de les envoyer sur un lien série qui peut ensuite agir sur la génération d'un signal auditif ou lumineux.
Le composant possède un FIFO interne permettant de stocker plusieurs caractères et de les envoyer ensuite en rafale.

Paramètres génériques

Le composant possède un FIFO de taille générique, qui est configurable via le paramètre `FIFOSIZE`. Un deuxième paramètre générique, `ERRNO`, a une fonctionnalité expliquée plus loin dans ce document.

Entrées/sorties

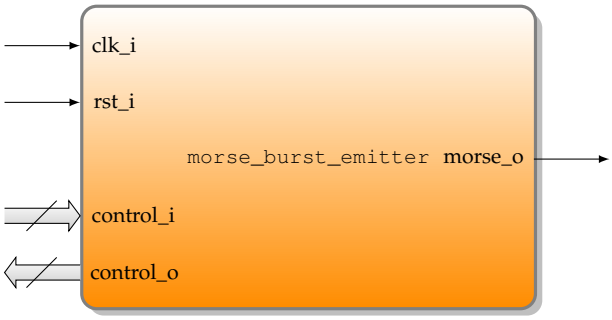
L'entité est déclarée ainsi :

```
entity morse_burst_emitter is
generic (
    FIFOSIZE : integer := 16;
    ERRNO    : integer := 0
);
port (
    clk_i     : in std_logic;
    rst_i     : in std_logic;
    control_i : in morse_burst_emitter_control_in_t;
    control_o : out morse_burst_emitter_control_out_t;
    morse_o   : out std_logic
);
end morse_burst_emitter;
```

L'usage de `record` pour le port de contrôle offre une connectique plus simple que des ports individuels.
Les entrées/sorties sont :

1. Idée tirée d'un travail de Loïc Gillioz et Eddy Mariéthoz

Nom	Taille	Dir	Description
`clk_i`	1	in	Horloge du système
`rst_i`	1	in	Reset du système
`control_i.char`	8	in	Valeur ASCII du caractère à transmettre, sur 8 bits
`control_i.load_char`	1	in	Force le chargement d'un nouveau caractère dans le FIFO d'envoi
`control_i.send`	1	in	Lance l'envoi du contenu du FIFO sur la sortie, en morse
`control_i.dot_period`	28	in	Nombre de cycles d'horloge correspondant à la durée d'un *dot*
`control_o.busy`	1	out	Indique qu'un envoi est en cours, actif haut
`control_o.full`	1	out	Indique que le FIFO interne est plein, actif haut
`control_o.unknown`	1	out	Indique que le caractère chargé n'est pas supporté
`morse_o`	1	out	Sortie série sur lequel le code Morse sera généré

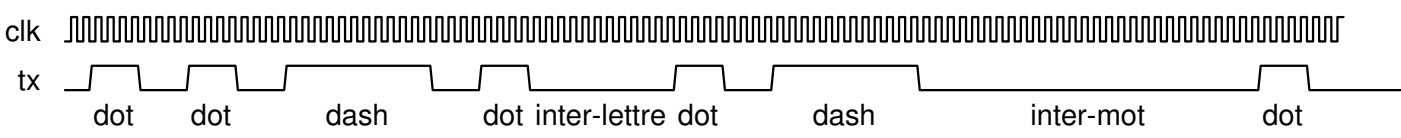


Fonctionnement

Le code morse permet d'envoi d'information en codant les lettres grâce à des activations d'un signal sur des durées longues ou courtes. Nous ne traiterons ici que des caractères standards, à savoir toutes les lettres (minuscule et majuscule), et les chiffres. Nous ajouterons également l'espace afin de pouvoir séparer des mots.
La figure 1 présente la représentation usuelle des caractères qui nous intéressent [2].
Nous utiliserons le terme *dot* pour désigner le point, et par abus de langage la durée d'un point. Le terme *dash* représentera un tiret, et sa durée associée. Les durées sont les suivantes :

Signal Morse	durée (en *dot*)	Description
'1'	1	Un *dot* codant pour une partie de lettre
'1'	3	Un *dash* codant pour une partie de lettre
'0'	1	Un espace entre deux *dot* ou *dash*, à l'intérieur d'une lettre
'0'	3	Un espace délimitant deux lettres
'0'	7	Un espace délimitant deux mots

Si le signal est à '0' pendant 7 *dot* ou plus, nous considérerons ceci comme un seul espace.
Le chronogramme suivant présente l'envoi de la lettre F suivie de A, d'un espace et de la lettre E, avec une durée de quatre cycles d'horloge par *dot*.



2. Il existe plusieurs standards

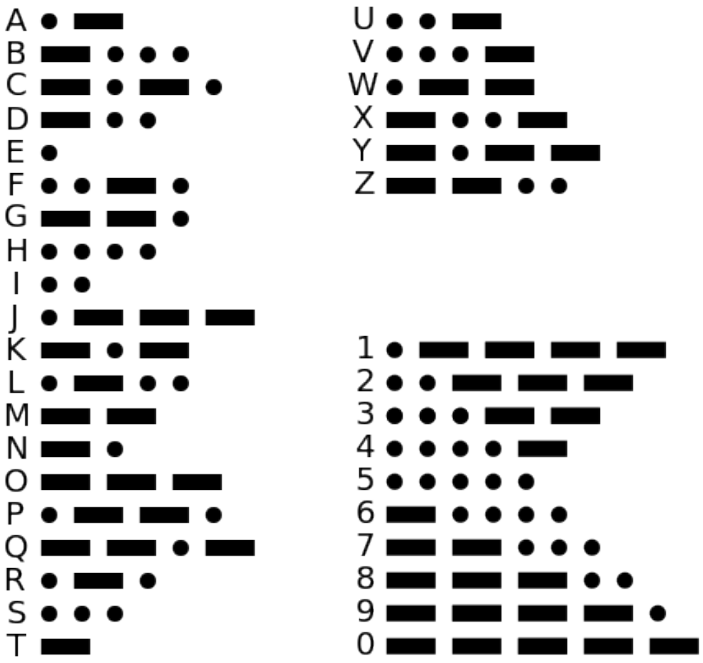
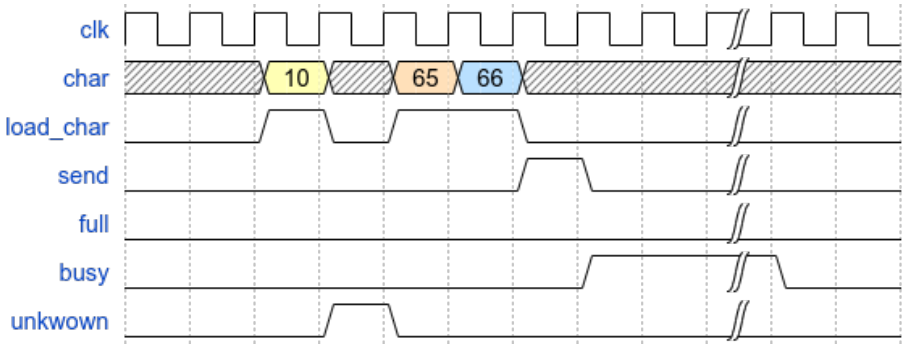


FIGURE 1. *Code morse standard (Source : wikipedia)*

L'activation de `load_char` pendant un cycle permet de charger la valeur `char` dans le FIFO interne. L'activation de `send` permet de lancer l'envoi des données stockées. La valeur présente sur l'entrée `dot_period` lorsque `send` passe à '1' indique le nombre de cycles d'horloge correspondant à la durée de *dot*. La modification de cette entrée à d'autres moments ne doit pas avoir d'influence sur l'envoi en cours.
La sortie `busy` passe à '1' le cycle suivant le démarrage d'un envoi d'un burst, et reste à '1' jusqu'à ce que l'envoi des lettres soit terminé. Il faut noter que le système, lorsqu'il commence l'envoi, va le faire jusqu'à ce que le FIFO soit vide. Ceci implique qu'il est possible d'ajouter des éléments dans le FIFO au fur et à mesure, de manière à avoir un flux continu en sortie.
La sortie `full` indique simplement que le FIFO est plein. Dès lors, si ce signal est actif, activer `load_char` n'aura aucun effet. L'écriture d'un caractère dans le FIFO ne se fait donc que si `load_char` est à '1' ET que `full` est à '0'.
La sortie `unknown` passe à '1' pendant un cycle d'horloge au cycle suivant l'activation de `load_char`, si le caractère `char` n'est pas supporté par le système. Dans ce cas le caractère n'est pas ajouté dans le FIFO et ne sera évidemment pas envoyé sur la ligne de sortie. Les caractères supportés par le système sont tous les codes ASCII correspondants aux entiers de 0 à 9 (ASCII 48 à 57), tous les caractères de 'A' à 'Z' (ASCII 65 à 90), et 'a' à 'z' (ASCII 97 à 122), ainsi que l'espace (ASCII 32). Il est important de noter que le système accepte donc les majuscules et les minuscules, mais que le code morse ne permet de pas de les distinguer. Il y a donc une seule manière de code 'A' et 'a'.
Le chronogramme suivant illustre le protocole d'entrée/sortie.



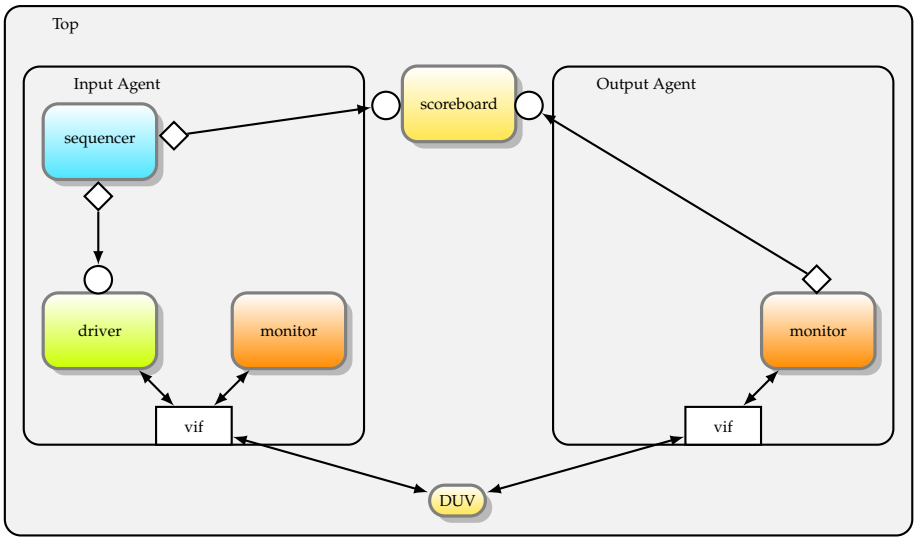
La version du DUT qui vous est fournie est parfaitement fonctionnelle, toutefois un paramètre générique `ERRNO` permettra d'injecter artificiellement des erreurs dans le design. Il s'agira d'un entier qui offre le comportement suivant :

1. S'il vaut 0, le résultat est valide ;
2. S'il est compris entre 1 et 17, le résultat n'est pas valide.

Ce paramètre générique vous permettra de valider votre banc de test, en le simulant avec toutes les valeurs de `ERRNO`.
⚠ L'entité à tester ne doit pas être modifiée. Elle est encryptée, ce serait de toute façon un peu difficile.

Architecture

Le banc de test a été décomposé selon la méthodologie vue en cours, afin de bien différencier les responsabilités (séquenceur, driver, moniteur, scoreboard). La structure globale vous est déjà fournie, avec les instanciations de composants et les connexions via des FIFOs. Passez un peu de temps à bien comprendre cette architecture en regardant les fichiers VHDL.



Vous pouvez noter les points suivants :

1. Les transactions envoyées par l'agent Input au scoreboard viennent du séquenceur plutôt que du moniteur. Dans notre cas précis cela permettra de facilité la détection de certaines erreurs.
2. Il n'y a pas de driver de sortie, car rien n'est en entrée du DUV du côté sortie Morse.

Etape 1

Un banc de test fonctionnel vous est fourni. Il est compilable et simulable, mais comme vous le verrez il n'offre pas de traitement réellement pertinent. A vous de modifier les séquenceur/-driver/moniteur/scoreboard/transactions afin de pouvoir correctement vérifier le DUV. Pour commencer, le composant qui mérite le plus de travail (dans un premier temps) est le moniteur de sortie. En effet, en l'état il ne récupère pas le code Morse observé. Modifiez-le de manière à ce qu'il puisse reconstruire une transaction en fonction de ce qu'il observe sur la ligne.
Pour la détection de la fin de la simulation, vous pouvez noter l'usage d'objections. Vous observerez qu'il y a déjà un moniteur de fin de simulation dans le banc de test. Celui-ci appelle une fonction lorsqu'il n'y a plus d'objections ou qu'il ne détecte plus d'activité.
Le script de simulation peut être lancé depuis le répertoire `sim` (exemple ici, pour le testcase 1, taille de FIFO à 8, et ERRNO à 0) :

```
do ../scripts/sim.do all 1 8 0
```

Etape 2

Il est maintenant temps de réfléchir aux scénarios de test. Un testplan vous est fourni. Il contient déjà les lignes correspondantes aux tests des différents ERRNO. Ajoutez vos propres scénarios de test. N'hésitez pas à stresser un peu le système (ou non).

Etape 3

Implémentez les tests que vous avez prévus. Cette étape nécessitera potentiellement des modifications dans tous les composants du testbench. A vous de réfléchir à l'endroit où faire telle ou telle vérification. N'hésitez pas à en discuter avec votre professeur.
Le testbench a un paramètre générique `TESTCASE`, qui vise à pouvoir choisir le testcase à jouer. Utilisez ce paramètre. Afin de pouvoir automatiser les tests, faites que le testcase 0 lance tous les testcases à la suite.

Etape 4

Un fichier `default.rmdb` vous est fourni. Vous pouvez lancer `vrun directed` pour valider votre système. Les simulations seront lancées avec des tailles différentes, des `ERRNO` différents, et sur les deux designs. Il est également pertinent d'y ajouter les testcases que vous avez décidé d'implémenter, afin de lier votre testplan au fichier rmdb.

Etape 5

N'hésitez pas à exécuter certaines étapes en parallèle. Vous êtes maintenant prêts à vous lancer sur l'étape 1.

Travail à rendre

— Générez une archive du projet à l'aide du script `vsn_rendu.sh`. Ce script vous est fourni avec le code et l'énoncé du laboratoire. Il génère une archive pour autant que le rapport (fichier `rapport.pdf`) et le fichier testbench soient présents. Attention à lancer le script depuis le répertoire où il se trouve. Il faut donc que le script se trouve dans le même répertoire que le fichier `rapport.pdf` et que le répertoire `code`.
— Copiez l'archive sur Cyberlearn.

Barème de correction

Documentation	20%
Pertinence des testcases	20%
Architecture du code	15%
Détection / non détection correcte	25%
Codage	10%
Commentaires au niveau du code	10%