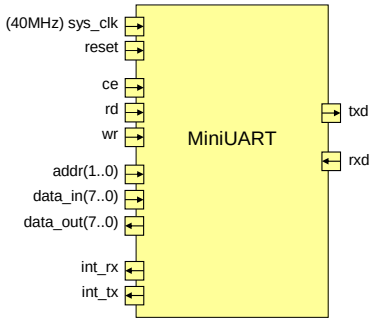
1 - Conception d'une Mini UART

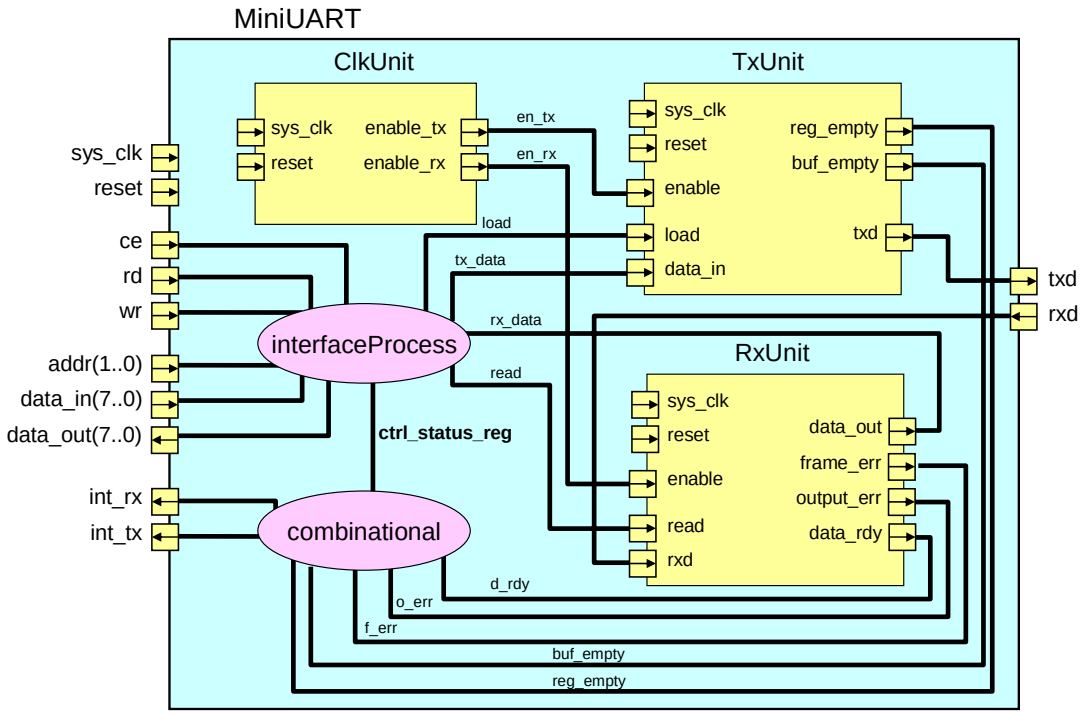
1.1 Présentation

L'objectif est de concevoir une UART très simple dont les caractéristiques sont les suivantes :

- Trame : 8 bits de données, pas de parité, 1 bit de stop
- Transmission à 9600 bauds (TxD)
- Réception à 9600 bauds (RxD)
- Une ligne d'interruption en réception (int_rx)
- Une ligne d'interruption en émission (int_tx)



Nous proposons de raffiner l'UART comme ci-dessous :



1.1.1 Module ClkUnit

Ce module génère un signal en_tx (enable tx) à une fréquence de 9600 Hz. La précision sera de ± 5Hz. Le rapport cyclique sera de 9600Hz/40MHz, soit 1 période de sys_clk (1/40MHz).

Le module génère également un signal en_rx (enable rx) à une fréquence de 16 fois 9600Hz, soit 153600Hz. La précision sera de ± 2KHz. Le rapport cyclique sera de 153600Hz/40MHz, soit 1 période de sys_clk (1/40MHz).

1.1.2 Module TxUnit

Le module convertit une donnée parallèle en donnée série transmise sur la ligne TxD à 9600 bauds. Le process du module se déclenche sur « sys_clk ». Le module comporte 2 principaux registres :

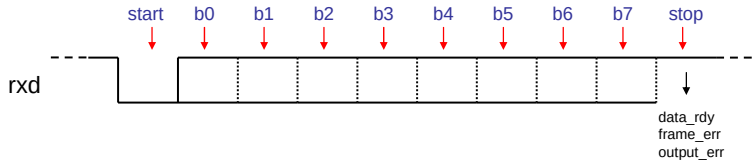
- Un registre « reg » qui permet de sérialiser la donnée (conversion parallèle/série). Une sortie « reg_empty » envoie une information sur l'état du registre (vide ou plein)
- Un registre tampon noté « buf » contient la prochaine donnée à sérialiser. De même, une sortie « buf_empty » envoie une information sur l'état du registre.

L'entrée « enable » de période 9600Hz permet d'activer la sérialisation.

L'entrée « load » signifie qu'une donnée est présente sur l'entrée « data_in » et qu'il faut la charger dans le registre tampon « buf », sans oublier de mettre la sortie « buf_empty » à false.

1.1.3 Module RxUnit

Ce module réceptionne une trame série. La ligne série (RxD) est échantillonnée à 16 fois la fréquence de transmission. L'information est prélevée au $8^{ème}$ échantillon, c'est-à-dire au milieu de chaque bit. La figure suivante illustre le principe de capture.



Lors de la réception du bit de stop, différents signaux sont mis à jour :

- Le signal data_rdy (data ready in buffer) mis à vrai et mis à faux dès la lecture dans le registre « data_out » (signal read).
- Le signal frame_err égale à vrai si le bit de stop détecté est égale à 0 sinon frame_err = faux
- Le signal output_err égale à vrai si le signal data_rdy est vrai. Cela signifie que la donnée précédente n'a pas été lue.
- Le module comporte au moins un registre « shift_reg » qui convertit la donnée en parallèle. Lors du stop, le registre « shift_reg » est transféré dans « data_out »

1.1.4 Les processes du module MiniUART

Ce module renferme un registre « ctrl_status_reg » qui permet de connaître l'état de l'UART. Le format de ce registre est le suivant :

7	6	5	4	3	2	1	0
-	-	-	-	-	empty	frameErr	OutputErr

Process « interfaceProcess »

Le process « interfaceProcess » permet de gérer les signaux load, tx_data, rx_data et read. Les signaux sont spécifiés à l'aide de table de vérité.

La table de vérité des signaux load et tx_data :

ce	wr	addr(1..0)	load	tx_data
0	X	X	0	0…0
1	0	X	0	0…0
1	1	00	1	data_in
1	1	X1 ou 10	0	0…0

La table de vérité des signaux read et data_out :

ce	rd	addr(1..0)	read	data_out
0	X	X	0	Z…Z
1	0	X	0	Z…Z
1	1	00	1	rx_data
1	1	01	0	ctrl_status_reg
1	1	1X	0	Z…Z

Process « combinational »

Le process « combinational » génère les interruptions en émission/réception ainsi que la mise à jour la concaténation de signaux pour former le « ctrl_status_reg »

L'interruption en émission (int_tx) est vraie lorsque le signal « buf_empty » est vrai et que le signal « reg_empty » est faux.

L'interruption en réception (int_rx) est vrai lorsque le signal « d_rdy » est à vrai, c'est-à-dire qu'une donnée est présente dans le buffer du module TxUnit. Le signal « int_rx » est à faux autrement.

Le registre « ctrl_status_reg » est juste une concaténation des signaux buf_empty, f_err et o_err.

1.2 Conception de la MiniUART

1.2.1 Le module ClkUnit

- Ecrire le module ClkUnit (clk_unit.h et clk_unit.cpp)
- Pour tester ce module, écrire dans un fichier (test_clk_unit.cpp) une fonction « test_clk_unit() » qui sera appelé à partir du main(). Un fichier header contient toutes les méthodes de tests qui seront écrits (test.h). La description est en Annexe I.

1.2.2 Le module TxUnit

- Ecrire le module TxUnit (tx_unit.h et tx_unit.cpp)
- Le test se fera comme pour le module ClkUnit (cf. Annexe II)

1.2.3 Le module RxUnit

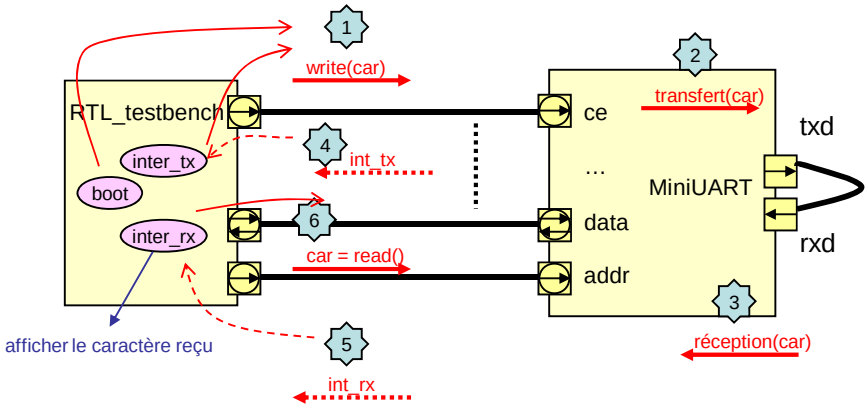
- Ecrire le module RxUnit (rx_unit.h et rx_unit.cpp)

- Ecrire le test en reprenant le test du module TxUnit et en y ajoutant l'instance RxUnit et les connexions nécessaires. Un signal txd_rxd permettra de connecter la sortie txd à l'entrée rxd.

1.2.4 Le module MiniUart

- Ecrire le module qui est composé de 3 instances et 2 processes (SC_METHOD).
- Pour le test, nous allons réaliser un module RTL_testbench (RTL_testbench.h et RTL_testbench.cpp). La connexion se fera, comme d'habitude dans une méthode test_miniuart() appelé à partir du sc_main().

  Le test est d'envoyer un message sou interruption et de le réceptionner sous interruption. La figure ci-dessous illustre le principe de test.



  Etapes :
  1) Le process « boot » écrit le premier caractère du message (pour déclencher l'interruption int_tx pour les caractères suivants).
  2) Le caractère est transmis (L'interruption int_tx peut être déclenchée)
  3) Le caractère est reçu
  4) L'interruption Tx est déclenchée et la routine d'interruption inter_tx écrit le caractère suivant
  5) L'interruption Rx est aussi déclenchée
  6) La routine d'interruption inter_rx lit le caractère et l'affiche
- Ecrire dans le module RTL_testbench 3 méthodes d'aides (Helper Functions).
  - resetTest() : active le signal reset pendant 20 cycles horloges
  - write() : écriture d'un octet sur le bus
  - read() : lecture d'un octet sur le bus

  L'annexe III donne le code de la méthode read() et write().
- Ecrire les processes (SC_THREAD ou SC_METHOD ?) du module RTL_testbench
- Le test ne fonctionne peut être pas correctement ?
  – Etat X sur le bus de donnée ? utilisez sc_signal_rv.
  – Au démarrage, le process boot écrit une valeur dans le buffer. Cependant, une interruption est déclenchée en même temps que l'écriture du boot et une seconde écriture réalisé par le process inter_tx est faite en parallèle. Cela engendre un conflit !

    Comme solution, utiliser un mutex (sc_mutex) qui permet d'obtenir une seule exécution de lecture/écriture à la fois.

Annexe I - Module ClkUnit

I.1 Fichier « test_clk_unit »

```
#include <systemc.h>
#include "clk_unit.h"

void test_clk_unit()
{
          sc_set_time_resolution(1, SC_NS);
          sc_signal<bool> reset, enable_tx, enable_rx;
          sc_clock clk("clk",25,SC_NS);         // 40MHz

          ClkUnit ClkUnit_inst("ClkUnit");
                    ClkUnit_inst.sys_clk(clk.signal());
                    ClkUnit_inst.reset(reset);
                    ClkUnit_inst.enable_tx(enable_tx);
                    ClkUnit_inst.enable_rx(enable_rx);

          sc_trace_file *tf = sc_create_vcd_trace_file("wave_clkunit");
          sc_write_comment(tf, "Simulation of Clk Unit");
          ((vcd_trace_file*)tf)->sc_set_vcd_time_unit(-9);// 10exp(-9) = 1ns

          sc_trace(tf,clk.signal(), "clk");
          sc_trace(tf,reset,"reset");
          sc_trace(tf,enable_tx,"enable_tx");
          sc_trace(tf,enable_rx,"enable_rx");

          for (int i=0; i<5; i++)
          {
                    sc_start(1, SC_MS);
          }
          sc_close_vcd_trace_file(tf);
}
```

I.2 Fichier « main.cpp »

```
#include <systemc.h>
#include "test.h"

int sc_main(int argc, char* argv[])
{
          test_clk_unit();

          return 0;
}
```

I.3 Fichier « tests.h »

```
void test_clk_unit();
```

Annexe II - Module TxUnit

II.1 Fichier « test_tx_unit.cpp »

```
#include <systemc.h>
#include "clk_unit.h"
#include "tx_unit.h"

void test_tx_unit()
{
	sc_set_time_resolution(1, SC_NS);
	sc_signal<bool> reset, enable_tx, enable_rx, load, reg_empty,
buf_empty, txd;
	sc_signal<sc_uint<8> > tx_data;
	sc_clock clk("clk",25,SC_NS);  // 40MHz

	ClkUnit ClkUnit_inst("ClkUnit");
		ClkUnit_inst.sys_clk(clk.signal());
		ClkUnit_inst.reset(reset);
		ClkUnit_inst.enable_tx(enable_tx);
		ClkUnit_inst.enable_rx(enable_rx);

	TxUnit TxUnit_inst("TxUnit");
		TxUnit_inst.sys_clk(clk.signal());
		TxUnit_inst.reset(reset);
		TxUnit_inst.enable(enable_tx);
		TxUnit_inst.load(load);
		TxUnit_inst.data_in(tx_data);
		TxUnit_inst.reg_empty(reg_empty);
		TxUnit_inst.buf_empty(buf_empty);
		TxUnit_inst.txd(txd);

	sc_trace_file *tf = sc_create_vcd_trace_file("wave_txunit");
	sc_write_comment(tf, "Simulation of Tx Unit");
	((vcd_trace_file*)tf)->sc_set_vcd_time_unit(-9); // 10exp(-9) = 1ns

	...
	sc_trace(tf,txd,"txd");

	// Reset
	cout << "Reset ..." << endl;
	load.write(false);
	reset.write(true);
	sc_start(100, SC_NS);
	reset.write(false);
	sc_start(100, SC_NS);
	// Load
	cout << "Load ..." << endl;
	tx_data.write(0x11);
	load.write(true);
	sc_start(500, SC_NS);
	// Send
	cout << "Send on TxD ..." << endl;
	tx_data.write(0x22);
	load.write(false);
	sc_start(12, SC_US);

	sc_close_vcd_trace_file(tf);
}
```

II.2 Fichier « main.cpp »

Il suffit d'ajouter l'appel de la méthode test_tx_unit() et de mettre en commentaire l'ancien test du module ClkUnit.

```
#include <systemc.h>
#include "test.h"

int sc_main(int argc, char* argv[])
{
          //test_clk_unit();
          test_tx_unit();

          return 0;
}
```

II.3 Fichier « tests.h »

Ajouter la seconde méthode de test.

```
void test_clk_unit();
void test_tx_unit();
```

Annexe III - Test de la MiniUart

III.1 Fichier RTL_testbench.h

```
SC_MODULE(RTL_TestBench)
{
	// Port Declaration
	sc_out<bool> sys_clk, reset;

	sc_out<bool> ce, rd, wr;
	sc_out<sc_uint<2> > addr;
	sc_in<sc_lv<8> > data_in;
	sc_out<sc_lv<8> > data_out;

	sc_in<bool> int_rx, int_tx;

	// Channel Members

	// Data Members
	sc_time CLK_PERIOD;		// 40 MHz
	char *mess;
	int cpt_car;

	// Sub-modules
	sc_clock clock;
	sc_mutex bus_mutex;

	// Constructor
	SC_CTOR(RTL_TestBench) : CLK_PERIOD(25, SC_NS), clock("Clock",
CLK_PERIOD),
			ce(false), rd(false), wr(false), addr(0),
data_out("ZZZZZZZZ"), cpt_car(0)
	{
			SC_METHOD(sysclk_method);
				sensitive << clock.default_event();
			SC_THREAD(boot_thread);
			SC_THREAD(tx_interrupt_thread);
			SC_THREAD(rx_interrupt_thread);
	}

	// Processes
	void sysclk_method();
	void boot_thread();
	void tx_interrupt_thread();
	void rx_interrupt_thread();

	// Helper Functions
	void resetTest(void);
	void write(unsigned int addr, sc_uint<8> data);
	void read(unsigned int addr, sc_uint<8>& data);
};
```

III.2 Méthode d'aide du fichier RTL_testbench.cpp

```
void RTL_TestBench::write(unsigned int addr_, sc_uint<8> data_)
{
	ce->write(true);
	wr->write(false);
	rd->write(false);
```

```
        addr->write(addr_);
        data_out->write("ZZZZZZZZ");
        wait(CLK_PERIOD);

        wr->write(true);
        data_out->write(data_);
        wait(CLK_PERIOD * 3);

        ce->write(false);
        wr->write(false);
        data_out->write("ZZZZZZZZ");
        wait(CLK_PERIOD);
}

void RTL_TestBench::read(unsigned int addr_, sc_uint<8>& data_)
{
        ce->write(true);
        wr->write(false);
        rd->write(false);
        addr->write(addr_);
        data_out->write("ZZZZZZZZ");
        wait(CLK_PERIOD);

        rd->write(true);
        wait(CLK_PERIOD * 3);

        data_ = data_in->read();
        ce->write(false);
        rd->write(false);
        wait(CLK_PERIOD);
}
```