Laborator 11

Automate cu stări finite / Comunicație serială

1. Obiective

Studiul, proiectarea, implementarea și testarea:

- **Automate (mașini) cu stări finite**
- **Comunicație serială**

2. Fundamente teoretice

2.1. Mașini cu stări finite

Automatele sau mașinile cu stări finite (Finite State Machine – FSM) se pot folosi pentru a descrie o unitate de control. Un FSM constă dintr-un set finit de stări, tranzițiile între stări, și acțiunile asociate cu fiecare stare.

În mod implicit Xilinx-ul încearcă să recunoască FSM-urile scrise în codul VHDL. În acest scop, în anexa 7 se prezintă cele 3 tipuri de descriere pentru un FSM, care sunt recunoscute automat de Xilinx, la sintetizare: cu 1, 2, sau 3 procese.

Practic, stările FSM-ului sunt declarate în mod generic, cu numele lor sub formă de enumerare:

```
…
        type state_type is (s1,s2,s3,s4);
        signal state   : state_type ;
…
```

Numele stării va fi folosit pentru a referi starea în descrierea VHDL, în loc de o anumită codificare numerică. Astfel descrierea VHDL a FSM-ului este de nivel mai înalt, simbolic, ușor de urmărit. În compensare, Xilinx Vivado este capabil să aplice diferite tehnici de codificare a stărilor, utilizatorul putând alege tehnica din **Synthesize Settings – fsm_extraction**: Auto (implicit), One-hot, Gray, Johnson, Sequential. O descriere detaliată (avantaje) a acestor moduri se găsește în XST User Guide.

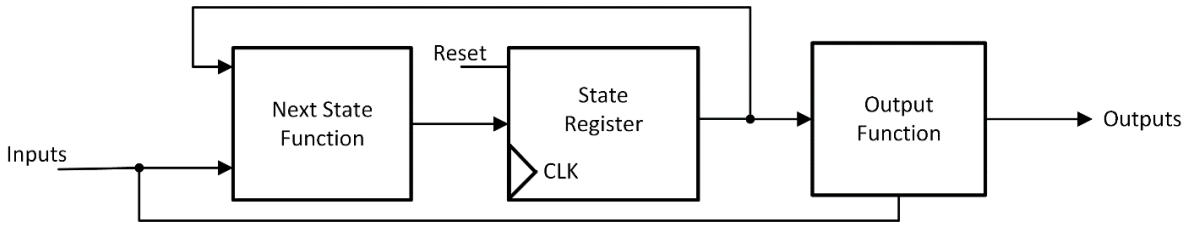

Figura 1: Reprezentarea FSM (include Mealy și Moore) conform XST User Guide

2.2. Comunicație serială - UART

Comunicația serială are ca principiu de bază transmiterea sau recepția datelor bit cu bit, un singur bit fiind transmis/recepționat la un moment dat. Deoarece în sistemele de calcul datele sunt reprezentate pe octet (sau multiplu), se folosește un port serial cu rolul de a converti fiecare octet într-un șir de biți (0 sau 1) și viceversa. Portul serial conține un circuit electronic numit Universal Asynchronous Receiver/Transmitter (UART) care face conversia efectivă.

La transmiterea unui octet, UART transmite mai întâi bitul de START, urmat de biți de date (în mod uzual 8 biți, dar e posibil și cu 5, 6 sau 7 biți), urmați de bitul de STOP. Protocolul e repetat pentru fiecare octet din secvența care trebuie trimisă.

Figura 2: Diagrama de timp pentru transmisia serială, exemplu pe 8 biți

Transmisia serială nu implică existența unui semnal de ceas. În schimb se folosește o rată de eșantionare, numită *baud rate* (**număr de biți transmiși pe secundă**). Valorile uzuale pentru baud rate sunt 2400, 4800, 9600 și 19200. Practic, un bit este valid pe linia de transmisie pentru un interval dat de timp, egal cu inversul baud rate.

Câteva aspecte importante:

- Dacă nu este începută transmisia, linia serială este în starea IDLE, pe 1
- Bitul START este întotdeauna 0, iar biții de date sunt transmiși în ordinea inversă a semnificației, primul este LSB iar ultimul este MSB
- Bitul STOP este întotdeauna 1
- Durata bitului STOP poate avea mai multe valori: 1, 1.5 sau 2 perioade de bit (1/baud rate)
- Pe lângă biți de START și STOP se poate folosi un bit adițional de paritate, împreună cu datele, pentru a detecta eventuale erori de transmisie.

Datele transmise prin comunicație serială sunt codificate folosind coduri ASCII (vezi anexa 8). De exemplu, dacă se dorește trimiterea caracterului ‘C’, se va transmite codul ASCII pe 8 biți al caracterului, în acest caz 01000011 (43h). Dacă se folosesc 8 biți de date, 1 bit de STOP, fără bit de paritate, atunci șirul de biți care trebuie transmis pentru caracterul ‘C’ este de forma (START) (DATA) (STOP):

LSB **(0 1 1 0 0 0 0 1 0 1)** MSB.

Pe acest exemplu, pentru un caracter reprezentat pe 8 biți trebuie transmiși în total 10 biți. Astfel se poate calcula numărul de caractere care se pot trimite pe secundă. La un baud rate de 9600, rezultă 9600/10 = 960 caractere pe secundă.

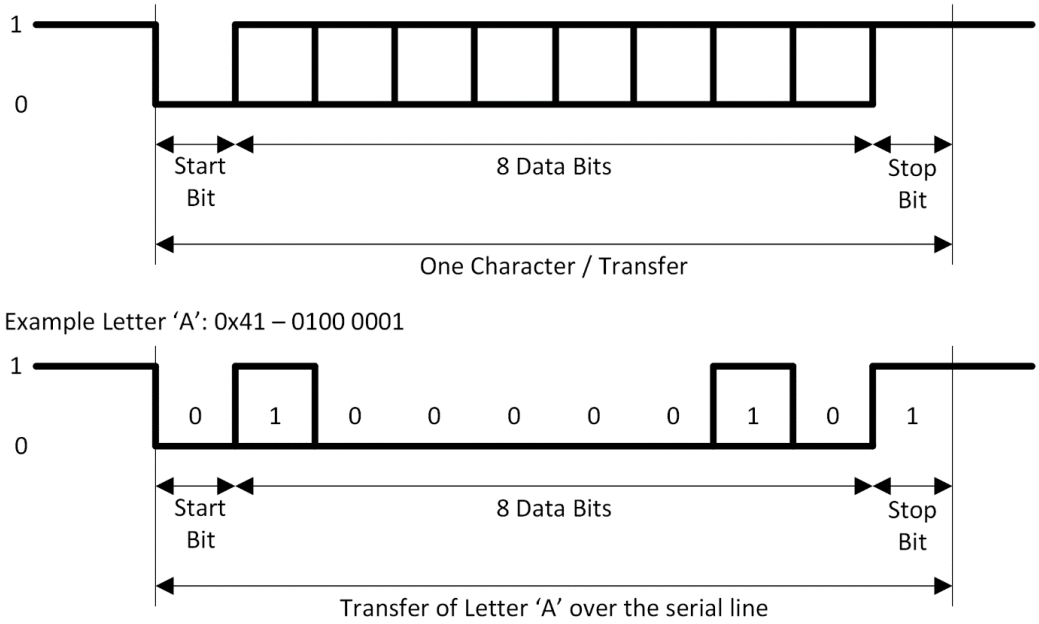

Figura 3: Exemplu de transmisie serială (litera A, 8 biți de date, fără paritate)

În cazul recepției, trebuie citit (eșantionat) semnalul de pe linia serială la o rată mai mare decât rata de transmisie. Dacă s-ar citi la rata de transmisie, din cauza nesincronizării perfecte (baud rate este generat independent la sursa, respectiv destinație), există riscul unor decalaje care cauzează citirea dublă a aceluiași bit, sărirea peste un bit, ratarea bitului de start, etc. Astfel este importantă detecția mijlocului bitului de start, biții de date fiind apoi citiți aproximativ în jurul mijlocului de interval de bit, eliminând practic riscul de decalaj. Cea mai uzuală schema de supra-eșantionare este folosirea unei rate de 16 ori mai mare decât baud rate. Concret, fiecare bit care vine pe serial este eșantionat (citit) de 16 ori, însă doar una dintre citiri este salvată (cea din mijloc).

Orice circuit UART conține un registru de deplasare (shift register), acesta fiind folosit în mod clasic pentru conversia datelor din forma paralelă în forma serială, și invers.

3. Activități practice

3.1. Pmod USB-UART

Pentru plăci de dezvoltare Basys 1 & 2 (ignorați dacă lucrați pe plăcile din laborator): Pentru comunicarea serială UART între placa de dezvoltare și calculator este necesar un modul de extensie. Citiți manualul de referință al acestui circuit Pmod USB-UART.

În figura următoare se prezintă modului Pmod USB-UART conectat la placa de dezvoltare FPGA pe unul dintre porturile de expansiune.

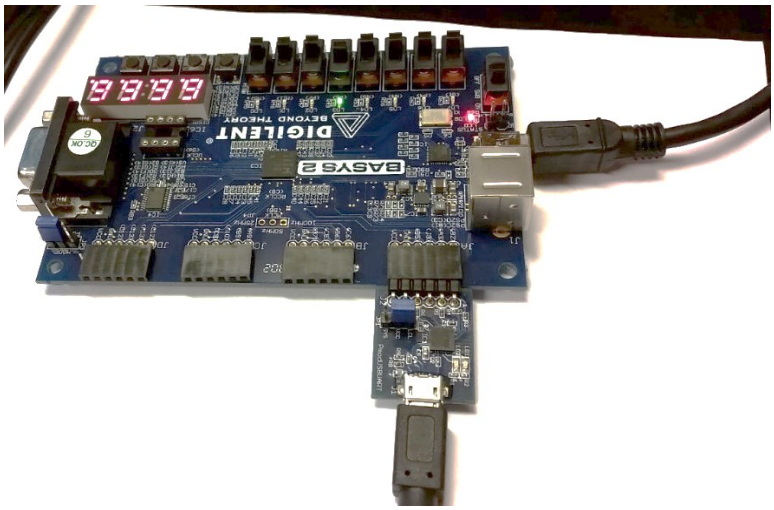

Figura 4: Pmod USB-UART conectat la placa FPGA

Folosiți un cablu de tip USB-Micro pentru conectarea modulului UART la portul USB al calculatorului, când veți ajunge la partea de testare.

Pentru plăci de dezvoltare Basys 3 (Artix 7): placa este dotată cu un circuit USB UART (marca FTDI) care comunică prin același cablu cu care se face programarea plăcii. Consultați manualul de referință pentru Basys 3 (secțiunea cu USB UART).

Descărcați și deschideți aplicația HTERM care va oferă un terminal pentru comunicare serială.

Trebuie sa definiți două porturi noi (std_logic) la test_env pentru comunicarea serială (liniile de transmisie și recepție) RX (in) și TX (out). Atribuiți pini pentru aceste două porturi în fișierul XDC (vezi primul laborator pentru metodologia de atribuire a unui pin nou). Folosiți manualul de referință (primul laborator) al plăcii de dezvoltare pentru a determina cei doi pini din portul de extensie unde veți lega modulul UART.
Atenție: TX din test_env (placa de dezvoltare) trebuie să corespundă fizic cu RX de pe modulul Pmod USB-UART / FTDI iar RX din test_env trebuie să corespundă cu pinul TX al modulului Pmod USB-UART / FTDI.

3.2. FSM pentru transmisie serială

Mai întâi trebuie să descrieți în test_env un generator pentru semnalul de baud rate, pentru o valoare de 9600 (biți pe secundă). Folosiți un numărător pentru a genera semnalul BAUD_ENable (’0’ valoare normală, este pus pe ‘1’ la intervalul de bit, interval care se măsoară în tacți de ceas, după care se reîncepe numărarea). Atenție, durata cât stă BAUD_ENable ‘1’ pe este egală cu perioada de ceas.

Pentru generarea baud rate (Basys 3 are 100 Mhz):

- La ceas de 25 MHz, se generează ‘1’ la fiecare 25Mhz/9600=2604 tacți.
- La ceas de 50 MHz, se generează ‘1’, la fiecare 5208 tacți.
- La ceas de 100 MHz, se generează ‘1’, la fiecare 10416 tacți.

Definiți o nouă entitate pentru FSM-ul de transmisie, cu porturile conform figurii următoare.

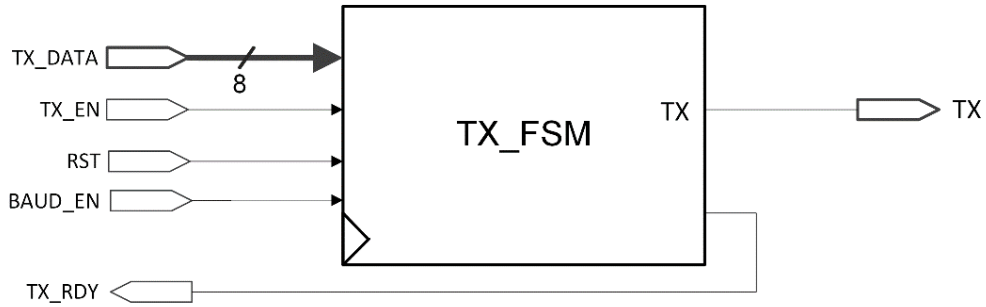

Figura 5: Entitatea cu FSM-ul de transmisie TX_FSM

Diagrama în detaliu a TX_FSM este prezentată în figura de mai jos. O tranziție între stări poate avea loc pe frontul crescător de ceas doar dacă BAUD_ENable este ‘1’. Astfel se asigură că un bit rămâne pe linia de transmisie pentru intervalul dat de baud rate. Semnalul BIT_CNT are o funcționalitate de numărător în interiorul FSM-ului, el reprezentând poziția bitului curent de transmis din TX_DATA. El trebuie incrementat în starea *bit* și trebuie resetat după fiecare caracter transmis (se poate reseta în starea *idle* sau în toate stările exceptând *bit*).

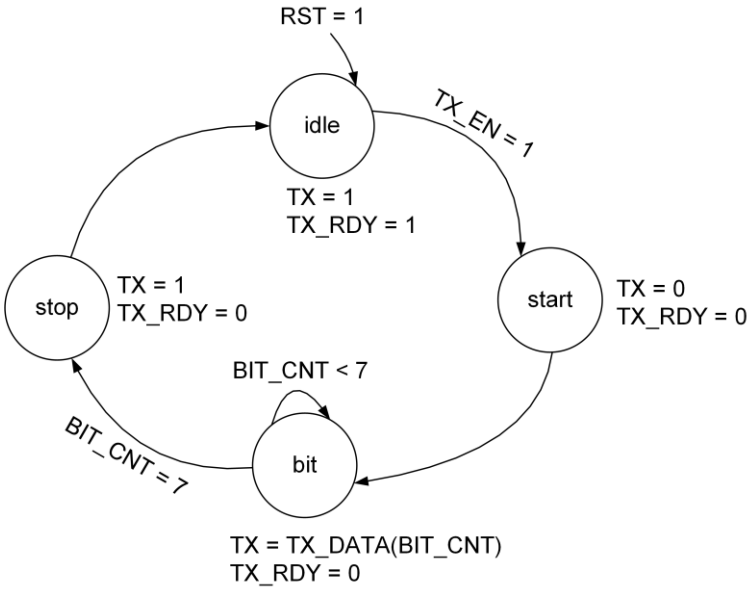

Figura 6: Diagrama TX_FSM

Descrieți în VHDL comportamentul entității TX_FSM, din test_env. Folosiți o descriere FSM cu 2 sau cu 3 procese (vezi anexa 7).

Pentru a testa transmisia serială de la placa de dezvoltare la calculator, conectați intrarea TX_DATA a FSM-ului la cele 8 switch-uri ale plăcii, pe care veți alege diferite coduri ASCII.

Trebuie generat semnalul TX_EN care să pornească transmisia. Acesta trebuie controlat de la un MPG, dar (!) nu direct, deoarece el trebuie sa rămână pe '1' până apare și BAUD_ENable='1' (altfel nu se va ieși din starea *idle*). Se va descrie un bistabil D, pe front crescător, care are set de la MPG, și reset de la BAUD_ENable. Ieșirea din acest bistabil este semnalul dorit TX_EN. Pe semnalul RST al FSM-ului legați '0' sau un alt MPG.

Acum se va testa comunicația între placa de dezvoltare și calculator. Parametrii pentru comunicația serială sunt: 1 bit de START, 1 bit de STOP, 8 biți de date, fără bit de paritate, baud rate de 9600. Asigurați-vă că aceste setări apar în aplicația de pe calculator HTERM / hyper terminal, și alegeți corect portul serial în aplicație (COM1, 2, 3 etc). Pentru a identifica exact care port serial este cel cu placa, vizualizați în aplicație lista de porturi seriale înainte și după legarea modulului prin cablul USB la calculator.

3.3. Comunicare I/O cu procesorul MIPS

Conectați TX_FSM cu procesorul MIPS pe care l-ați implementat (puteți folosi MIPS cu ciclu unic sau versiunea pipeline, o versiune funcțională).

Trebuie să trimiteți 16 biți de date din procesorul vostru către PC. Acești biți ar trebui să reprezinte rezultatul final al programului vostru (sau unul dintre rezultate). În funcție de unde este stocat acest rezultat (în blocul de regiștrii, în memoria de date), adăugați o instrucțiune nouă la finalul programului vostru care să acceseze acea locație (implicit valoare va apărea în căile de date, și o puteți transmite).

Exemplu:
La terminarea programului, rezultatul este în $7, iar PC este 0x0020. Adăugați o nouă instrucțiune la program: addi $7, $7, 0, la adresa 0x0021 în memoria ROM de instrucțiuni. Definiți un registru de 16 biți care este scris cu valoarea din semnalul RD1/ALURes, scrierea fiind validată de valoarea lui PC (egală cu adresa instrucțiunii adăugate, atenție dacă testați PC+1…).

Definiți și descrieți în VHDL metodologia de transfer a celor 4 caractere conținute în registru (pe calculator trebuie să apară codificarea alfa numerică, la fel ca pe SSD). Folosiți un decodificator/ROM pentru a genera reprezentarea ASCII (8 biți) pentru o cifra hexa (4 biți). Trebuie să implementați un mecanism de baleiere a celor 4 cifre din registru, una câte una, si fiecare cifră trebuie transmisă prin intermediul TX_FSM. Indiciu: amintiți-vă de mecanismul de baleiere a cifrelor la SSD (sau implementați la registrul de 16 biți un mecanism de deplasare cu 4 poziții), și folosiți TX_RDY pentru a trece la următoarea cifră (TX_RDY arată că TX_FSM este în *idle*, deci poate începe o nouă transmisie).

4. References

- XST User Guide
- Digilent Basys Board – Reference Manual
- Digilent Basys 2 Board – Reference Manual
- Digilent Basys 3 Board – Reference Manual
- Digilent Pmod USB-UART – Reference Manual
- http://www.asciitable.com/
- http://www.der-hammer.info/terminal/

Appendix 7 – Finite State Machine Implementations (XST User Guide)

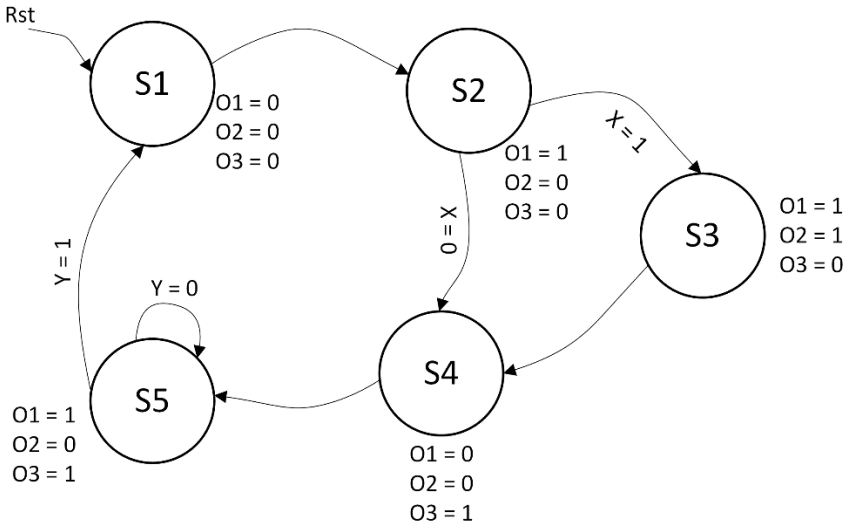

Figura 7: Finite State Machine Example (XST User Guide)

IO Pins	Description
clk	Positive Edge Clock
Rst	Asynchronous Reset (Active High)
X, Y	FSM Inputs
O1, O2, O3	FSM Outputs

Table 1: FSM Pin Descriptions

Descrierea acestui FSM în formate recunoscute de Vivado se face pe paginile următoare.

FSM with One Process VHDL Coding Example

```
entity fsm_1 is
        port (
                clk, rst, x, y    : IN std_logic;
                o1, o2, o3        : OUT std_logic
        );
end entity;

architecture beh1 of fsm_1 is
        type state_type is (s1, s2, s3, s4, s5);
        signal state    : state_type;
begin

        process (clk, rst, x, y)
        begin
                if (rst ='1') then
                        state <=s1;
                        o1<='0'; o2<='0'; o3<='0';
                elsif (clk='1' and clk'event) then
                        case state is
                                when s1 =>   state <= s2;
                                             o1<='1'; o2<='0'; o3<='0';
                                when s2 =>   if x = '1' then
                                                     state <= s3;
                                                     o1<='1'; o2<='1'; o3<='0';
                                             else
                                                     state <= s4;
                                                     o1<='0'; o2<='0'; o3<='1';
                                             end if;
                                when s3 =>   state <= s4;
                                             o1<='0'; o2<='0'; o3<='1';
                                when s4 =>   state <= s5;
                                             o1<='1'; o2<='0'; o3<='1';
                                when s5 =>   if y = '1' then
                                                     state <= s1;
                                                     o1<='0'; o2<='0'; o3<='0';
                                             else
                                                     state <= s5;
                                                     o1<='1'; o2<='0'; o3<='1';
                                             end if;
                        end case;
                end if;
        end process;

end beh1;
```

FSM with Two Processes VHDL Coding Example

```
entity fsm_2 is
        port (
                clk, rst, x, y    : IN std_logic;
                o1, o2, o3        : OUT std_logic
        );
end entity;

architecture beh1 of fsm_2 is
        type state_type is (s1, s2, s3, s4, s5);
        signal state    : state_type;
begin
        process1: process (clk, rst, x, y)
        begin
                if (rst ='1') then
                        state <=s1;
                elsif (clk='1' and clk'event) then
                        case state is
                                when s1 =>    state <= s2;
                                when s2 =>    if x = '1' then
                                                      state <= s3;
                                              else
                                                      state <= s4;
                                              end if;
                                when s3 =>    state <= s4;
                                when s4 =>    state <= s5;
                                when s5 =>    if y = '1' then
                                                      state <= s1;
                                              else
                                                      state <= s5;
                                              end if;
                        end case;
                end if;
        end process process1;

        process2: process (state)
        begin
                case state is
                        when s1 => o1<='0'; o2<='0'; o3<='0';
                        when s2 => o1<='1'; o2<='0'; o3<='0';
                        when s3 => o1<='1'; o2<='1'; o3<='0';
                        when s4 => o1<='1'; o2<='0'; o3<='0';
                        when s5 => o1<='1'; o2<='0'; o3<='1';
                end case;
        end process process2;
end beh1;
```

FSM With Three Processes VHDL Coding Example

```
entity fsm_3 is
        port (
                clk, rst, x, y    : IN std_logic;
                o1, o2, o3        : OUT std_logic
        );
end entity;

architecture beh1 of fsm_3 is
        type state_type is (s1, s2, s3, s4, s5);
        signal state, next_state      : state_type;
begin
        process1: process (clk, rst)
        begin
                if (reset ='1') then
                        state <=s1;
                elsif (clk='1' and clk'event) then
                        state <= next_state;
                end if;
        end process process1;


        process2: process (state, x, y)
        begin
                case state is
                        when s1 =>   next_state <= s2;
                        when s2 =>   if x = '1' then
                                             next_state <= s3;
                                     else
                                             next_state <= s4;
                                     end if;
                        when s3 =>   next_state <= s4;
                        when s4 =>   next_state <= s5;
                        when s5 =>   if y = '1' then
                                             next_state <= s1;
                                     else
                                             next_state <= s5;
                                     end if;
                end case;
        end process process2;


        process3: process (state)
        begin
                case state is
```

```
                    when s1 => o1<='0'; o2<='0'; o3<='0';
                    when s2 => o1<='1'; o2<='0'; o3<='0';
                    when s3 => o1<='1'; o2<='1'; o3<='0';
                    when s4 => o1<='1'; o2<='0'; o3<='0';
                    when s5 => o1<='1'; o2<='0'; o3<='1';
                end case;
        end process process3;
end beh1;
```

Appendix 8 – ASCII Codes Table (http://www.asciitable.com/)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com