Laboratory 9 : SimpleCPU_v1a Serial IO

The aim of this lab is guide you through the development of a more complex SimpleCPU based system. This example will implement a serial communication link i.e. RS232 serial link, between the FPGA and the host PC. To minimise hardware requirements this serial IO link (SIO) can be implemented using software emulation i.e. bit-banging:

“slang for various techniques for data transmission in which software is used to generate and process signals instead of dedicated hardware”

Using this technique general purpose IO lines are used to receive and transmit text characters, allowing us to implement that compulsory program that you must write on any new computer: printing “Hello World” :). Finally we will add an application specific peripheral device to the system, a: Universal Asynchronous Receiver and Transmitter unit (UART), or more commonly referred to as a serial port, allowing the user to read the status of the FPGA boards switches and control its output LED displays. At the end of this practical you will understand how to:

- Implemented software time delays.
- Use and process ASCII text data.
- Examine RS232 data packets using an oscilloscope.
- Debug and implement the RS232 serial protocol in software.
- Add application specific peripheral devices to a system.

An ISE project called `simpleCPU_v1a_SIO.zip`, has been created and can be downloaded from the webpage. Using your preferred web browser download this zip file to `c:\temp`. Right click on this file selecting ‘Extract all…’ to unzip it. To start this practical click on the start button and select the Xilinx ISE project navigator.

 `-> Xilinx 64bit project navigator`

This may take a few seconds (a minute when the network is busy) to start. To open this project left click on the `File` pull down within the project navigator window :

`File -> Open Project`

Then browse to the directory where you unzipped this project and select : `simpleCPU_v1a_sio.xise`. Next, within the Hierarchy window double click on the top level schematic: `computer.sch`, as shown in figure 1. This system is a simplified version of the hardware used in the previous laboratory, having an 8bit input port and an 8bit output port, both memory mapped to address 0xFC.

Note, the input and output ports are again implemented using the MUX and REG_8 components, as discussed in the previous laboratory script.

When a program reads address 0xFC (`LOAD 0xFC`) the multiplexer switches the `DATA_IN` bus of the processor from `RAM` to the external inputs. The top 7bits of this input port are unused, therefore, tied to a fixed logic 1. The `LSB` is connected to the serial data-in pin (`RX`). When a program writes to address 0xFC (`STORE 0xFC`) this data is written to the output register. The `Q` output pins of this register are used to

drive the external outputs. The top 6bits are unused, therefore, not connected. The LSB is connected to the serial data out pin (TX) and bit-1 to test pin (TP1).

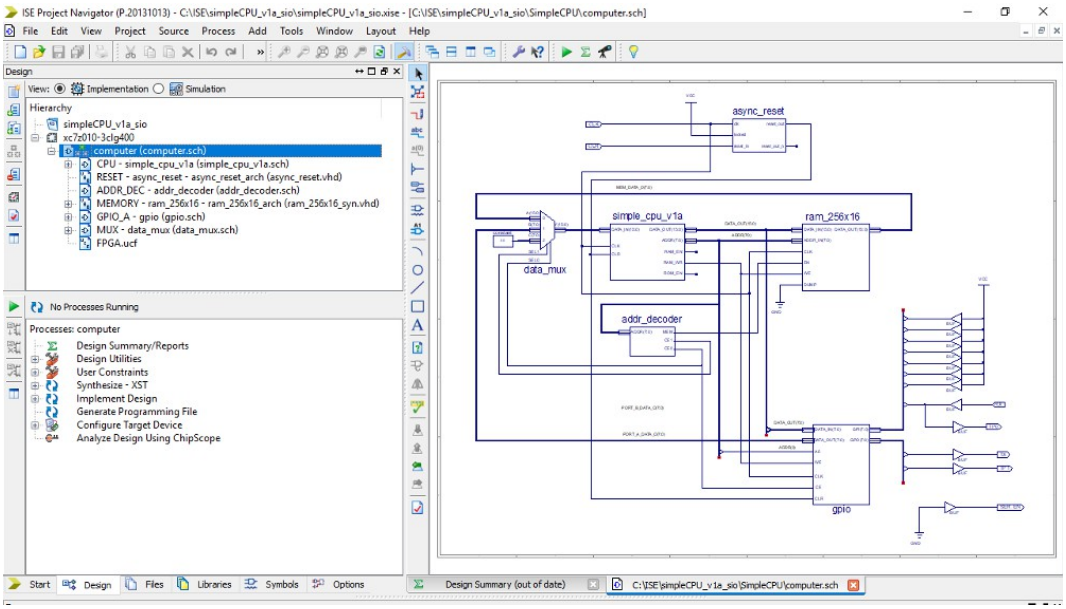


Figure 1 : top level schematic

Digital systems normally communicate data using parallel data buses, each data bit is transmitted on a separate wire at the same time i.e. in parallel. Therefore, the wider the data bus e.g. 32, 64, 128bits, the higher the data transfer rate. On the PCB shown in figure 2, two Quad Flat Package (QFP) surface-mount integrated circuits (ICs) are connected using a parallel bus: a groups of tracks (wires), highlighted in RED . A simulation of this type of data transfers is shown in the bottom waveform diagram. Here the data values 0x01, 0x0F, 0xF0 and 0xFF are transferred across this bus, the STROBE signal indicating to the receiving IC when this data is valid. The main advantage of a parallel data bus is its increased communication bandwidth i.e. the number of bits communicated per second, in this example:

$$\begin{aligned} \text{data rate} &= \text{data width} \times 1 \div \text{data transfer time} \\ &= 8 \times 1 \div (100 \times 10^{-9}) \text{ bits per second (bps)} \\ &= 80 \times 10^{6} = 80 \text{ Mbps} \end{aligned}$$

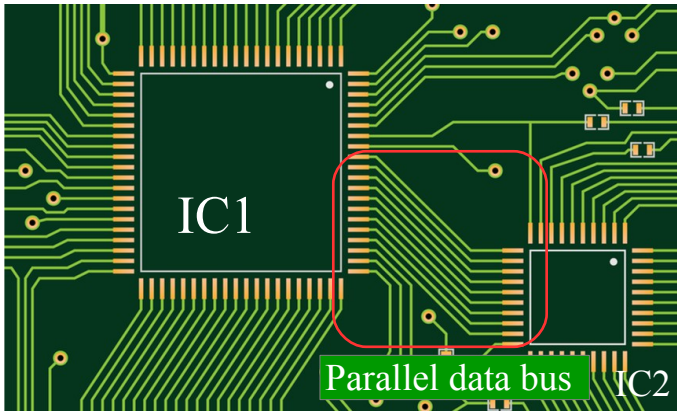
Note, owing to signal rise-time, fall-time and variations in track length the STROBE signal is delayed by the setup-time i.e. the time required for the data bits on the parallel bus to reach a stable voltage on the receiving IC's inputs.

The disadvantages of a parallel data buses are the hardware resources required e.g. multiple signal drivers and receivers (buffers), and the large amount of routing resources (wires) needed. Therefore, early computers tended to be based on a serial communication buses e.g. EDSAC. Now, rather than data bits being transfer in parallel, data is transmitted using a single wire, one bit at a time, as shown in figure 3.

In this example the data value 01001000_2 or 0x48 is being transmitted. When not in use the serial data line is set to a logic 1. Each data and control bit is active on the line

for the same amount of time. In this example data is transferred at 300 bps, therefore:

```
bit period  = 1 ÷ 300 = 3.3 ms
```



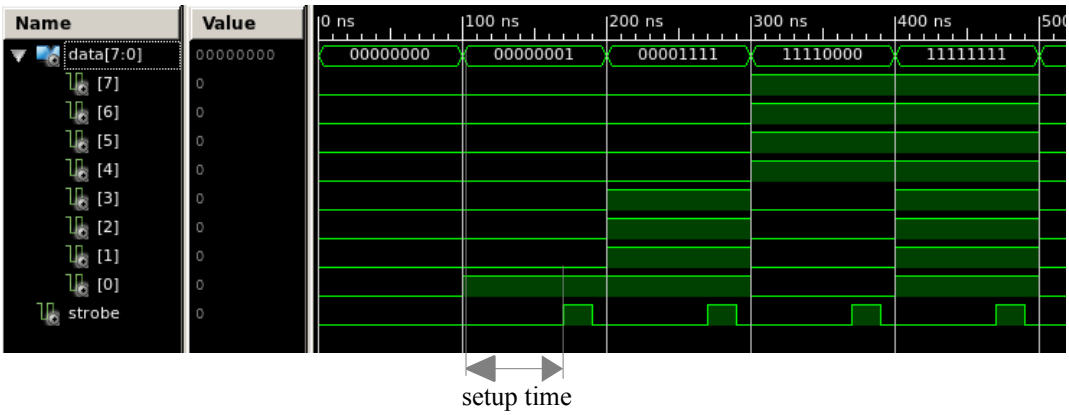


Figure 2 : parallel data packet

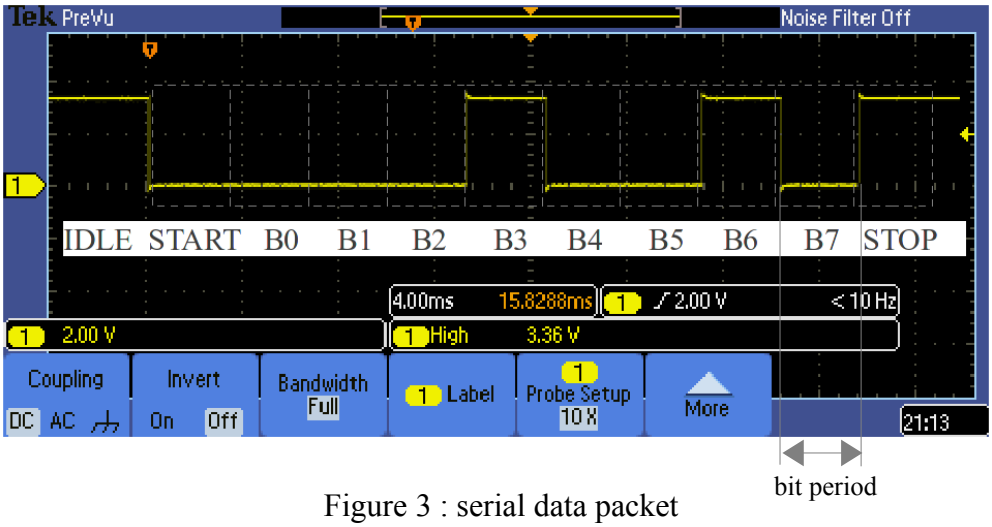


Figure 3 : serial data packet

To indicate to the receiving unit that data is going to be transferred the transmitter first sets the serial line to a logic 0 i.e. the `START` bit. The receiving unit detects this synchronising control bit and waits 1.5 bit periods, before sampling the incoming data every 1 bit period time delays. This is repeated nine times, until the complete data packet has been received i.e. 8 data bits and 1 stop bit.

Note, the `STOP` control bit is used to separate data packets, ensuring that there is always a transition from logic 1 to logic 0 on the initial `START` control bit.

The advantage of a serial bus is that it significantly reduces routing resources i.e. only need one data wire. It also helps reduce electrical noise problems, as there is only one data line to shield. The disadvantages of a serial data bus is that the communication data rate is significantly reduced. Comparing this approach to the previous parallel bus and assuming the same setup-time i.e. bit period, of 100ns, a serial buses data rate will be reduced by the data width:

$$\begin{aligned} \text{data rate} &= 1 \div (\text{data width} \times \text{data transfer time}) \\ &= 1 \div (8 \times 100 \times 10^{-9}) \text{ bits per second (bps)} \\ &= 1.25 \ \ 10^{6} = 1.25\text{Mbps} \end{aligned}$$

To operate at the same data rate the serial bus would need to increase its transfer frequency from 10MHz (100ns) to 80MHz (12.5ns).

In general most modern processing architectures are based around parallel buses, therefore, we will need additional hardware i.e. a parallel to serial converter, to transmit or receive serial data. **Note**, in the land of hardware these are called an Universal Asynchronous Receiver and Transmitter units (UART), or more commonly referred to as a serial ports.

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_ 0	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_ 16	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_ 32	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_ 48	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_ 64	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_ 80	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_ 96	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_ 112	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F

Letter Number Punctuation Symbol Other

Figure 4 : ASCII character codes

Task 1

To communicate text data the transmitting and receiving units must agree on what binary value corresponds to what character. A common character encoding scheme used in computers is the American Standard Code for Information Interchange (ASCII). The original standard encoded the English alphabet using a seven bit code i.e. 128 characters, as shown in figure 4. This included 95 printable characters: 0

to 9, lowercase and uppercase letters A to Z, and punctuation symbols. In addition there are 33 non-printing control codes e.g. bell, carriage return, line feed and tab etc.

Task : identify the ASCII character codes for the message “Hello World”. The table in figure 4 is organised such that the most significant hex digit is the row number and the least significant hex digit is the column digit e.g. capital “H” is on row four, column eight, therefore, its hexadecimal value is 0x48.

The first implementation of the “Hello World” program will use bit-banging i.e. the system will use GPIO lines to implement the serial port. The pseudo code to implement a serial transmitter is shown in figure 5.

```
set serial line low
wait 3.3ms
for i in range 0 to 7
     set serial line to DATA(i)
     wait 3.3ms
set serial line high
wait 3.3ms
```

Figure 5 : serial transmitter pseudo code

Task : before we implement this solution consider the types of instructions, addressing modes and data that will be required e.g. how will the ASCII text string be stored in memory, how will the processor access this data, how will it extract each data bit of the ASCII character to be transmitted etc.

Task 2

One of the functions identified in the pseudo code description is a 3.3 ms time delay i.e. the bit period (1/300). This will be implemented using a software time delay loop i.e. knowing the processor's clock speed and the number of clock cycles needed per instruction, we can implement a `FOR` loop to cause the processor to execute this number of instructions and therefore, the required time delay.

Task : if the processor is running at 10MHz and each instruction takes 3 clock cycles to execute, how many instructions will need to be performed to provide a delay of 3.3 ms?

$$\text{Instruction count} = (3.3\times10^{-3}) \div (1\div10\times10^{6})\times 3 = ???$$

To implement this delay the code shown in figure 6 could be used. This program uses two nested `FOR` loops to burn through the required number of instructions.

Task : calculate the outer loop count value i.e. calculate the value of CNT in line 60. Write a macro called `delay` to perform this function. This macro should be passed the following parameters:

1) The value of CNT
2) Address of the temporary variable used to store the outer loop counter
3) An unique ID used to generate the branch addresses used in this macro

If you would like to check your answers refer to Appendix A.

```
Address    Instruction     Comments

60         move    CNT     # save outer loop count
61         store   71

62         move    0x00    # load inner count
63         sub     0x01    # dec inner delay loop
64         jumpz   66      # exit if 0
65         jump    63      # repeat

66         load    71      # load outer loop counter
67         sub     0x01    # dec outer loop count
68         store   71
69         jumpnz  62      # repeat if not zero

70         jump    0       # exit

71         DATA            # outer loop count variable
```

Figure 6 : software time delay

To confirm that this macro will produce the required time delay enter the program shown in figure 7. This program sets/clears the output port every 3.3ms. You can then measure the pulse width of the resulting 150Hz square wave using an oscilloscope, as shown in figure 8.

```
Address  Label    Instruction            Comments

   0     start:   move   0              # clear port
   1              store  0xFC
   2              delay( 15, COUNT, 1 ) # delay 1

   3              move   0xFF           # set port
   4              store  0xFC
   5              delay( 15, COUNT, 2 ) # delay 2

   6              jump start
   7     COUNT:                        # count variable
```

Figure 7 : test code

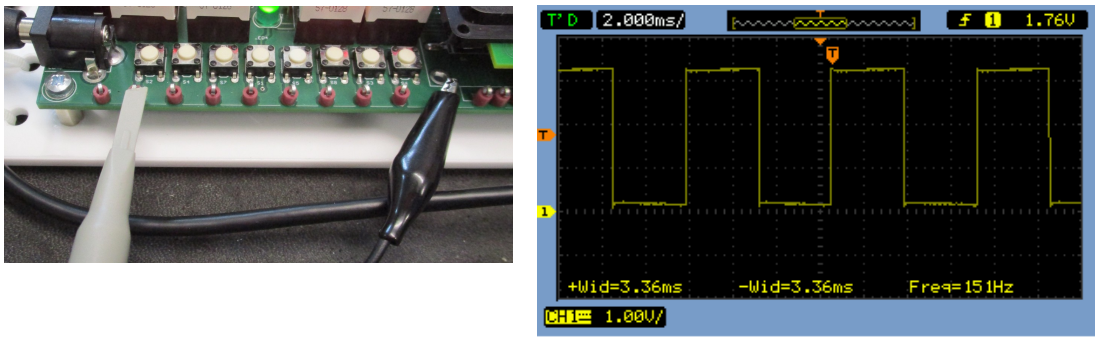


Figure 8 : output waveform

Task 3

Data is transmitted on the serial line one bit at a time, starting at the least significant bit position. Therefore, the software needs to test the state of each bit i.e. is it a logical 0 or 1. One approach would be to use the AND instruction to mask out the desired bit position. Consider the example code shown in figure 9.

```
Address    Instruction     Comments

60         move   0        # zero TX bit in tmp buffer
61         store  66

62         load   67       # load ACC with ASCII char
61         and    0x01     # mask bit-0
62         jumpz  65       # is bit-0 zero?

63         move   1        # no
64         store  66       # update TX bit to 1

65         jump   0        # yes, exit back to code
66         DATA            # TX bit variable
67         DATA            # ASCII char variable
```

Figure 9 : character bit test code

To test other bit positions the AND instruction's operand will be changed to the following hexadecimal values shown in figure 10.

```
Bit position   Hexadecimal value
0              0x01
1              0x02
2              0x04
3              0x08
4              0x10
5              0x20
6              0x40
7              0x80
```

Figure 10 : character bit test code

Task : write a macro called “testBit” to perform this function. This macro should be passed the hexadecimal mask value, the memory location where the ASCII character is stored and the memory location of the TX bit variable. If you would like to check your answer refer to Appendix B.

An alternative approach to solving this problem is to always test the same bit position, but then shift the data bits within the accumulator. Consider the example shown in figure 11. Here the data value is shifted right one bit position each time, inserting a logic 0 into the MSB position. If you look at the resultant value (on the right) you can observe that this results in the data value being divided by 2. An easy but some what inefficient method of performing multiplication or division is through repeated addition or subtraction. Consider the pseudo code shown in figure 12.

```
                0100 1000     =      0x48 = 'H'

                                     TX START
                0100 100 0    =      TX 0   =   0x48
 shift right    0010 010 0    =      TX 0   =   0x24
 shift right    0001 001 0    =      TX 0   =   0x12
 shift right    0000 100 1    =      TX 1   =   0x09
 shift right    0000 010 0    =      TX 0   =   0x04
 shift right    0000 001 0    =      TX 0   =   0x02
 shift right    0000 000 1    =      TX 1   =   0x01
 shift right    0000 000 0    =      TX 0   =   0x00
                                     TX STOP
```

Figure 11 : shifting character bit code

```
DIVIDEND = 100
DIVISOR = 3
QUOTIENT = 0

while DIVIDEND > 0
     DIVIDEND = DIVIDEND -  DIVISOR
     if  DIVIDEND > 0
          QUOTIENT = QUOTIENT +1
```

Figure 12 : simple divide pseudo code

Therefore, one method of shifting the data value to the right is to repeatedly subtract 2 from the ASCII character, counting the number of times this can be performed without producing a negative result. The final count value being the original data divided by 2. Consider the example code shown in figure 13.

```
Address    Instruction     Comments

60         move   0x00     # zero divide count
61         store  56

62         load   59       # load char, sub 2
63         sub    0x02
64         store  59       # save result
65         and    0x80     # test if neg
66         jumpnz 71       # yes exit

67         load   56       # no increment divide
68         add    0x01
69         store  56
70         jumpu  62

71         load   56       # update result
72         store  59
```

Figure 13 : divide by 2 test code

Task : write a macro called "`shiftRight`" to perform this function. This macro should be passed the memory address of the variable to be shifted and the memory locations of any intermediate variables or labels used. If you would like to check your answer refer to Appendix C.

Task 4

The simpleCPU computer is a von-Neumann based architecture i.e. instructions and data are stored in the same memory. To generate the required instructions we can write an assembly language program then use the assembler to produce the corresponding machine code, but how do we load data values into our program?

This is typically done using assembler directives e.g. within CPUSim you can use the `.data` directive to initialise memory locations with the desired values. However, the assembler used with the simpleCPU does not support these commands. Therefore, we need to be a little more creative :).

Question: when is an instruction an instruction, and when is it data? An instruction is just a bit pattern stored in memory, it only becomes an "instruction" if it is accessed during the fetch phase i.e. loaded into the instruction register. If the same "instruction" is accessed during the decode phase then that bit pattern will be loaded into the accumulator i.e. it becomes a data value. Therefore, we can use "instructions" to define our data values.

Task : consider the program in figure 14. What will be the value in the `ACC` when the program implements the `JUMP` instruction at address 64? Will the "instructions" at addresses 65-69 be executed? What bit values are stored in memory locations 65-69.

```
Address     Instruction     Comments
60          load   65       ACC <- M[65]
61          addm   66       ACC <- ACC + M[66]
62          addm   67       ACC <- ACC + M[67]
63          addm   68       ACC <- ACC + M[68]
64          jump   0        exit

65          move   1        DATA 1
66          move   2        DATA 2
67          move   3        DATA 3
68          move   4        DATA 4
69          move   5        DATA 5
```

Figure 14 : test code 1 - initialising memory example

Task : why will the program in figure 15 produce the same result as the program in figure 14? **Hint**, how wide is the `ACC`? What bits of the 16bits stored in each memory location are transferred when the LOAD / STORE instructions are executed?

Using this technique we can now initialise memory with the required character codes. The next step is to implement the pseudo code shown in figure 16. This is where we hit the next speed bump: the `LOAD` instruction only supports the absolute addressing

mode i.e. the address read is hard-coded and can not be changed at run time.

```
Address    Instruction     Comments

60         load   65       ACC <- M[65]
61         addm   66       ACC <- ACC + M[66]
62         addm   67       ACC <- ACC + M[67]
63         addm   68       ACC <- ACC + M[68]
64         jump   0        exit

65         add    1        DATA 1
66         jump   2        DATA 2
67         move   3        DATA 3
68         sub    4        DATA 4
69         jumpz  5        DATA 5
```

Figure 15 : test code 2 - initialising memory example

```
for I in range 0 to 10
     transmit DATA(I)

DATA: H,E,L,L,O, ,W,O,R,L,D,\0
```

Figure 16 : print message pseudo code

Task : how can we implement the `FOR` loop shown in figure 16 using only the instructions supported by the simpleCPU processor?

One solution to this problem is to use self-modifying code. Using this technique the address field of the `LOAD` instruction is overwritten with the address of the next character each time the transmit function is performed.

Note, to restate what was discussed in lectures self-modifying code is not a recommended programming technique, however, it was used extensively in old computers as it helped reduce hardware costs and improved memory usage. To remove the need for this programming technique modern processors support additional addressing modes e.g. register-indirect, memory-indirect and indexed. We shall be looking at some of these in the next laboratory where we will be using an improved version of the simpleCPU processor: `simpleCPU_v1d`.

Task : consider the code shown in figure 17, can you identify how this code implements the same functionality as that shown in figure 15.

Hints, what will be loaded into the `ACC` when the instruction at address 64 is read? As discussed in laboratory script 7, the `STORE` instruction at address 69 will overwrite the opcode and operand of the instruction stored at address 64. What new instruction will be stored at this address? Can you see how this program reads through the “data” values stored in addresses 73-77?

```
Address    Instruction     Comments

60         move    0       zero TOTAL variable
61         store   78
62         move    77      set MAX address variable
63         store   79

64         load    73      load data value
65         addm    78      add to TOTAL
66         store   71      update TOTAL

67         load    64      load LOAD
68         add     1       increment
69         store   4 64    update

70         subm    79      has all data been added?
71         jumpnz  64      no repeat

72         jump    0       exit

73         add     1       DATA 1
74         jump    2       DATA 2
75         move    3       DATA 3
76         sub     4       DATA 4
77         jumpz   5       DATA 5

78         DATA            TOTAL
79         DATA            MAX
```

Figure 17 : test code 3 - accumulating data example

```
PNTR = 0
loop:
     CHAR = DATA[PNTR]
     if CHAR = 0
          exit

     set serial line LOW
     wait for 3.3ms

     for I in range 0 to 7:
          set serial line to CHAR[I]
          wait for 3.3ms

     set serial line HIGH
     wait for 3.3ms

     PNTR = PNTR + 1

DATA : H,E,L,L,O, ,W,O,R,L,D,0
```

Figure 18 : print message pseudo code

Task 5

Using the previously defined macros and self modifying code we can now implement the program functions required to print “Hello World” on the screen. One possible pseudo code implement is shown in figure 18.

Note, the end of the string that will be transmitted is indicated using a NUL character i.e. “\0”, the value 0.

Task : using the previously designed macros write an assembly code program to implement the pseudo code shown in figure 18. If you would like to check your answer refer to Appendix D. Appendix E shows a solution using “subroutines”.

When you have completed your solution save this file to `helloWorld.asm`. Then at the command prompt assemble and link this file to produce the required `memory.vhd` configuration file:

```
m4 simpleCPUv1a.m4 helloWorld.asm > code.asm
simpleCPUv1a_as.py -i code -o code
simpleCPUv1a_ld.py -i code
```

Add this file to the ISE project, highlight the top level schematic: `computer.sch`, then generate the programming file: `computer.bit`, as shown in figure 19.

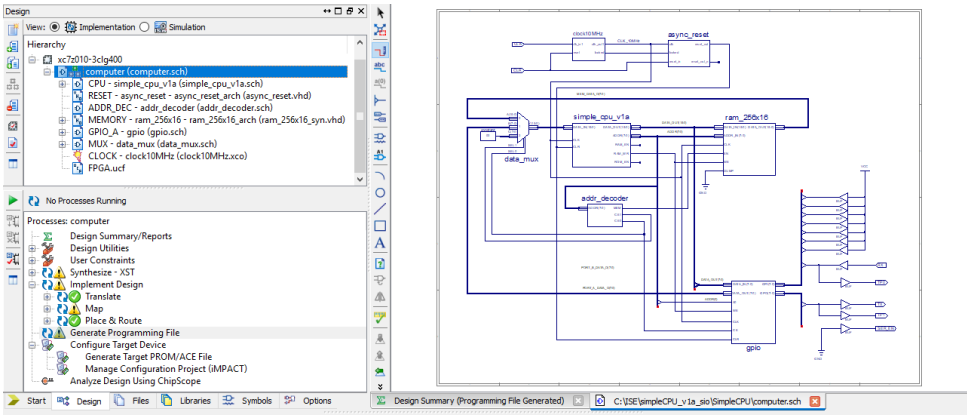


Figure 19 : generating bit file.

Upload this file into the FPGA board as previously described. You can view the serial packets by connecting the scope to the test point TP1, as shown in figure 20.

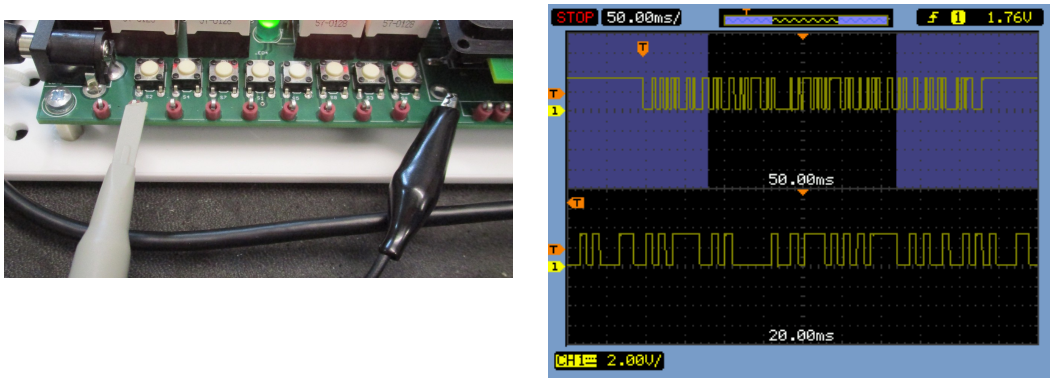


Figure 20 : TX data packet

Task : can you identify the ASCII values being transmitted from their serial bit patterns displayed on the scope?

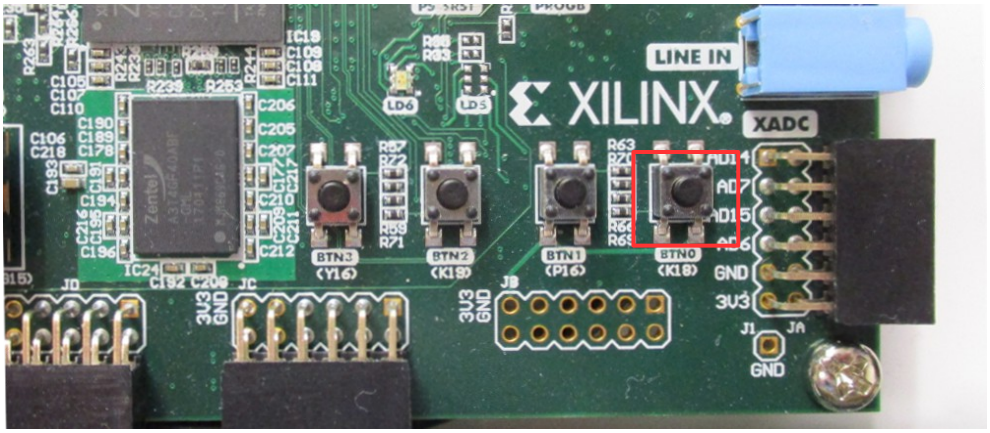


Figure 21 : Reset button

Note, you can trigger the processor to resend the “Hello World” message by pressing the reset button shown in figure 21.

To view these ASCII characters on the host PC connect the serial cable to the 9-way D-sub connector (DE-9) on the FPGA board, as shown in figure 22. **Note**, this cable should already be plugged into the host PC and accessible at the back of the desk.

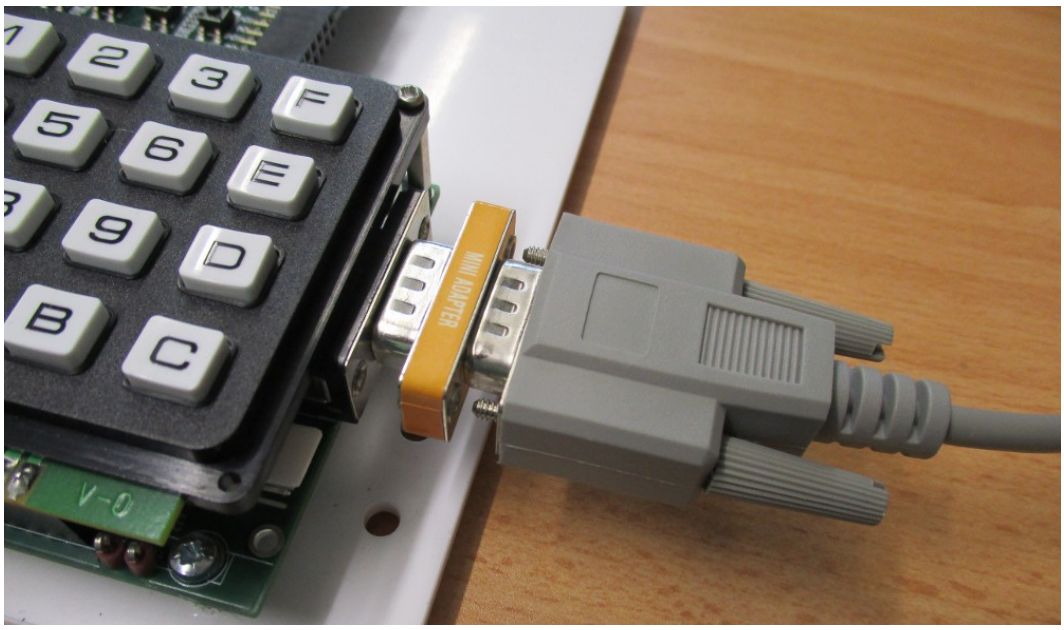


Figure 22 : Serial connector

The transmitted ASCII message can then be displayed on the PC's terminal program `PuTTy`. This program monitors the PC’s serial port displaying any received ASCII characters. From the Start menu type:

`-> PuTTy`

To configure the PuTTy terminal click on the Serial radio button, then enter Serial line (port): COM1 and Speed (baud rate): 300, as shown in figure 23.

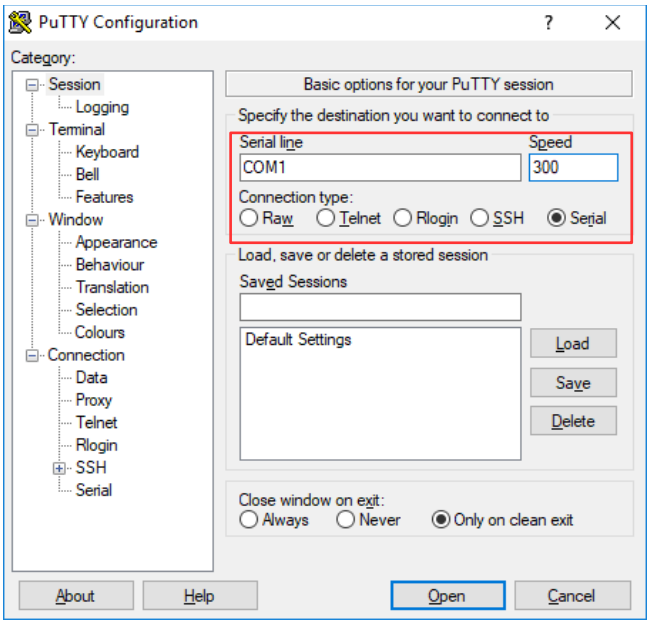


Figure 23 : configuring COM port

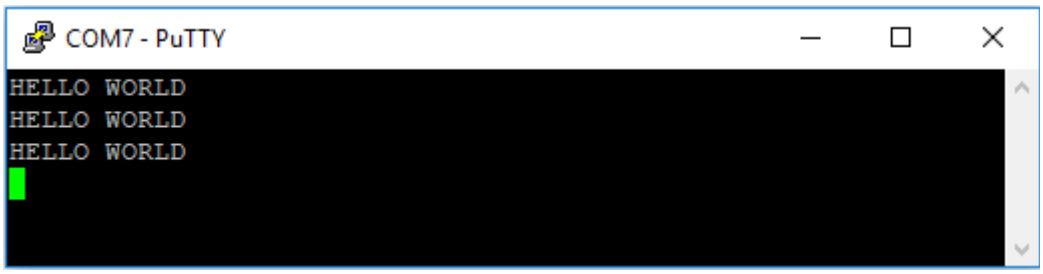


Figure 24 : “Hello World” display

You should now see “Hello World” printed to the terminal window, as shown in figure 24. You may need to press the RESET button on the FPGA, as shown in figure 21 to resend the message.

Task 6

Bit-banging a serial port can be useful in some situations e.g. when you only have a GPIO port, but the resulting solutions do tend to operate at a very slow bit rate. This is due to the overheads in executing “house keeping” functions e.g. division routine (shifting data bits) etc. The processing time required to perform these operations needs to be insignificant when compared to the bit period, otherwise the resulting bit periods will vary, causing the transmitting and receiving units to become unsynchronised.

To overcome this issue we can move this functionality out of software into dedicated hardware i.e. a UART. A new ISE project called simpleCPU_v1a_UART.zip, has been created and can be downloaded from the module's VLE page. Using your preferred web browser download this zip file to c:\temp. Right click on this file

selecting ‘Extract all…’ to unzip it. Then browse to the directory where you unzipped this project and select : `simpleCPU_v1a_sio.xise`. Next, within the Hierarchy window double click on the top level schematic: `computer.sch`, as shown in figure 25.

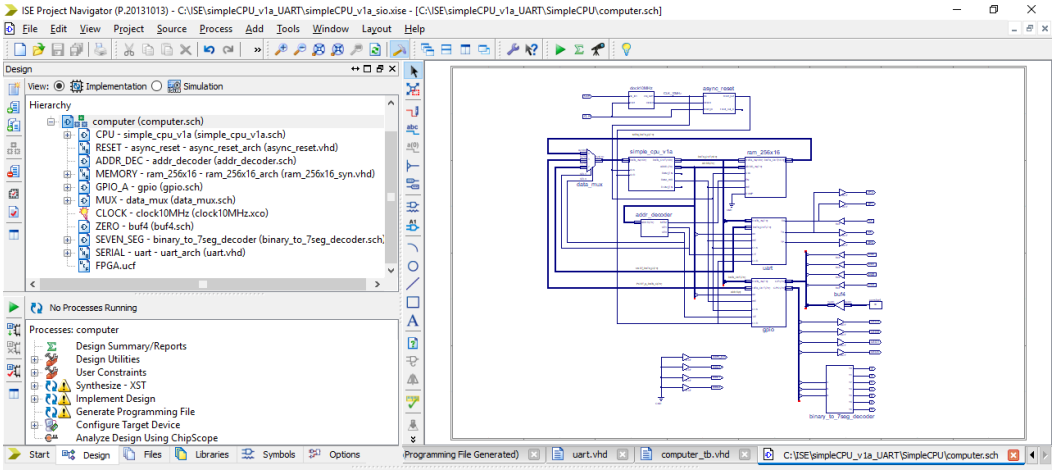


Figure 25 : top-level schematic.

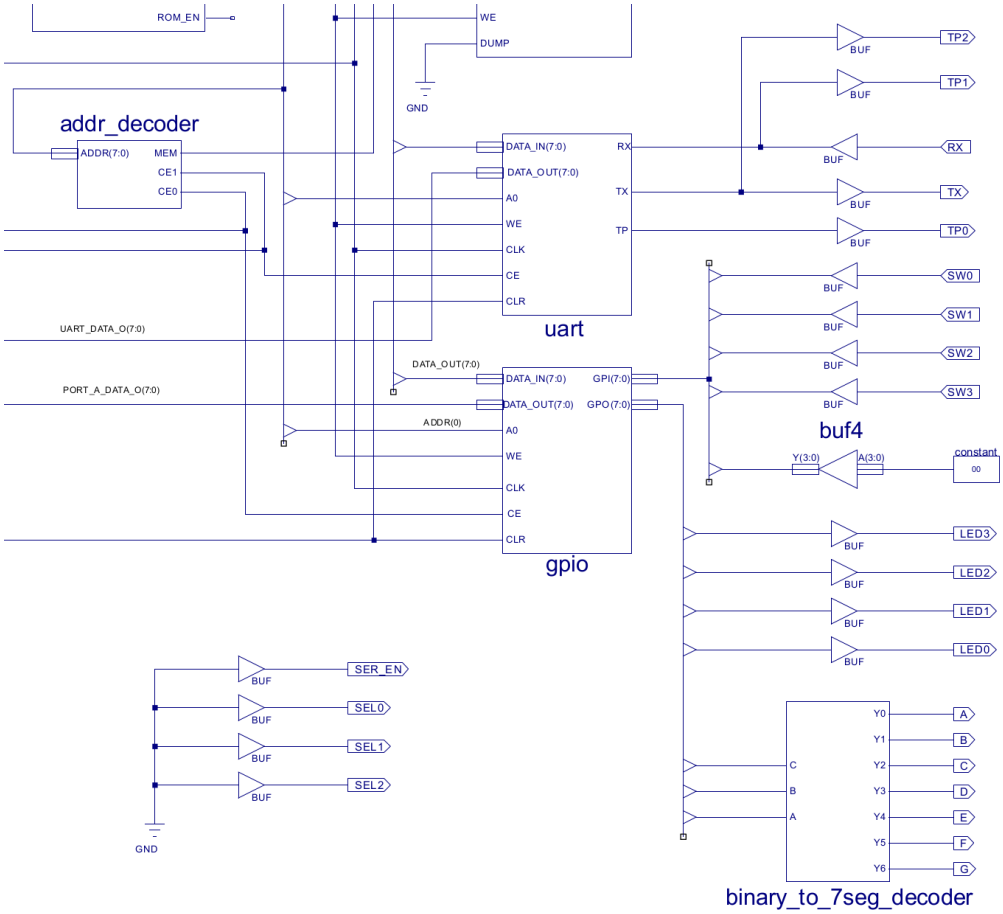


Figure 26 : interface devices – UART and GPIO

This new interface (peripheral) device is again mapped into the processor's memory map i.e. its internal register's are assigned specific addresses, as shown in figure 27.

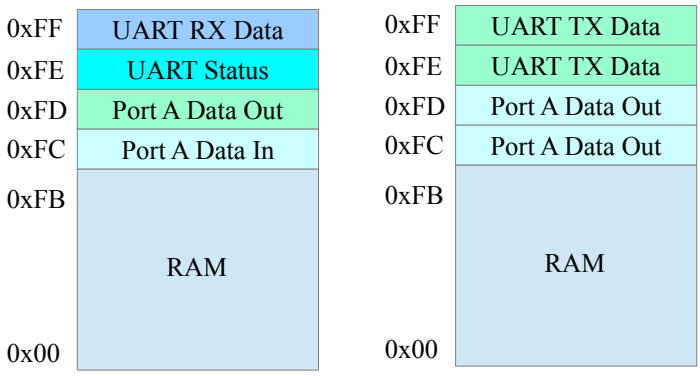


Figure 27 : Read (left) and Write (right) memory maps.

The UART has three internal registers: TX data, RX data and status. When data is written to the TX data register the UART's hardware automatically transmits this serial data packets to the host PC i.e. no further software control is required. The UART also contains hardware to receive serial data packets from the host PC. When received, this data can be read by the processor from the RX data register. This hardware can operate in parallel with the transmitting hardware i.e. the UART can receive and transmit data at the same time.

Note, communication systems are normally classified as: simplex, half duplex, or full duplex. The UART hardware used in this system is full duplex i.e. can receive and transmit data packets at the same time i.e. true hardware parallelism. To save money, sometimes a system will share hardware units. This hardware can still receive and transmit data packets, but not at the same time i.e. a half duplex system. If a system can only receive or transmit data, but not both, it is a simplex system.

To allow the processor to determine the UART's current state i.e. is it transmitting a data packet, or if it has received a data packet, it can read the UART's status register. This register contains three status flags:

BIT	DESCRIPTION	VALUES
7	NU	Not used, constant 0
6	NU	Not used, constant 0
5	NU	Not used, constant 0
4	NU	Not used, constant 0
3	NU	Not used, constant 0
2	TX idle	1=Idle, 0=Busy
1	RX idle	1=Idle, 0=Busy
0	RX data valid	1=True, 0=False

Figure 28 : UART status bits

Task : create a new program file: `uart.asm`, and enter the program shown in figure 29. This program transmits a character, then waits until the UART receives a character from the host PC. It then re-transmits this character back to the PC i.e. performs an echo function. Can you identify what character is initially transmitted and why the

status register is ANDed with the values 0x01 and 0x04? What will happen when the program writes the received data to address 0xFC?

```
start:
     move 0x2A    # what does this code do?
     store 0xFF

loop1:            # what is the purpose of this loop?
     load 0xFE
     and 0x04
     jumpz loop1

loop2:            # what is the purpose of this loop?
     load 0xFE
     and 0x01
     jumpz loop2

     load 0xFF    # what does this code do?
     store 0xFF
     store 0xFC

loop3:            # what is the purpose of this loop?
     load 0xFE
     and 0x04
     jumpz loop3

     jump start
```

Figure 29 : `uart.asm` assembly code

Assemble and link this program, and then add the resultant `memory.vhd` file to this project. Next generate the configuration bit file as shown in figure 25, upload this into the FPGA as previously described.

Relaunch the `Putty` serial terminal then enter Serial line (port): `COM1` and Speed (baud rate): `9600`, as shown in figure 30.

Note, as the serial port is now implemented in hardware we can significantly increase the data rate.

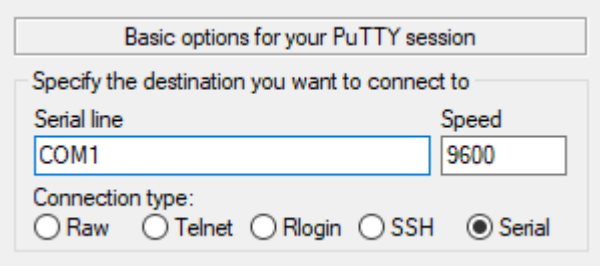


Figure 30 : configuring COM port

Task : within the `Putty` terminal window enter the values 0123456789. What characters are displayed in the terminal? What values are displayed on the seven

segment display? Looking at the ASCII character codes can you see why these values are displayed?

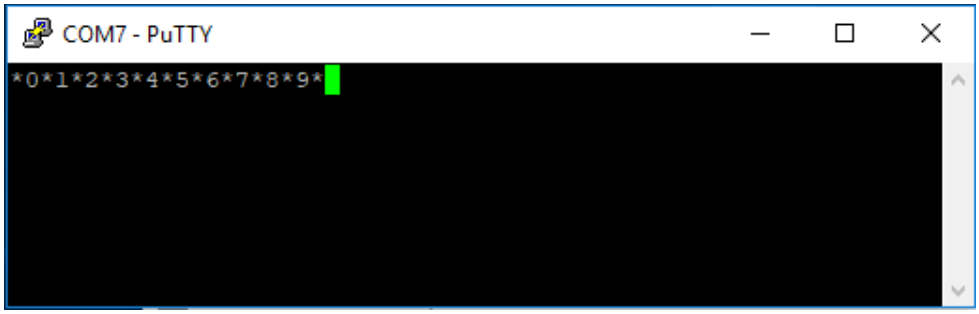


Figure 31 : echo display.

Hint, from figure 26 identify what is connected to the GPIO port. Examine the memory maps shown in figure 27. How is the GPIO and UART peripheral devices mapped into the processor's address space?

Additional Tasks

Task : using the `simpleCPU_v1a_SIO` project can you write a program to bit-bang a serial receiver i.e. a program that will receive ASCII characters from the host PC. Operating in a half duplex mode the system should receive a serial data packet then transmit back to the host PC this data so that it is displayed in the `Putty` terminal i.e. an echo program.

Hint, the software will need to poll (read) the RX input to detect the initial start bit, it should then wait 1.5-bit periods before it samples this data i.e. delay reading the first data bit until it has started. It will then sample the remaining data bits after a 1-bit period delay.

Task : using the `simpleCPU_v1a_UART` project can you write a program to receive and process ASCII characters:

- If the processor receives the letter "`S`" the system should transmit back to the host PC the state of the four toggle switches connected to the input port i.e. the ASCII character: `0 - F`, representing the bit input ports value 0000 – 1111.
- If the revived character is the letter "`L`" the system will wait for the next character: `0-F`, this defines the state of the four green LEDs connected to the output port: bits 3 – 6. The system will then update these outputs to this value i.e. if the character `C` is revived the output port will be set to the value 1100.

Hint, reading the input port you will need to mask out all unused input bits. When updating the output port you will need to ensure that only bits 3 – 6 are updated. To do these types of operations you will need to use bitwise AND and OR. You may also need to shift a bit pattern, this can be done using multiplication and division.

Summary

Communication delays within a system are an important consideration when assessing its processing performance i.e. it doesn't matter how fast, or how many processing elements a processor has, a system is limited by how fast it can feed data to these units or store the results they produce. This limitation is commonly referred to as the von-Neumann bottle-neck. To overcome this limitation we could increase the speed or width of the data buses used i.e. a wide memory interface, as shown in figure 32. Now rather than reading one 16bit instruction at a time the system reads two instructions, from two separate memory devices i.e. 32bits data bus, buffering unused values in the processor until required.

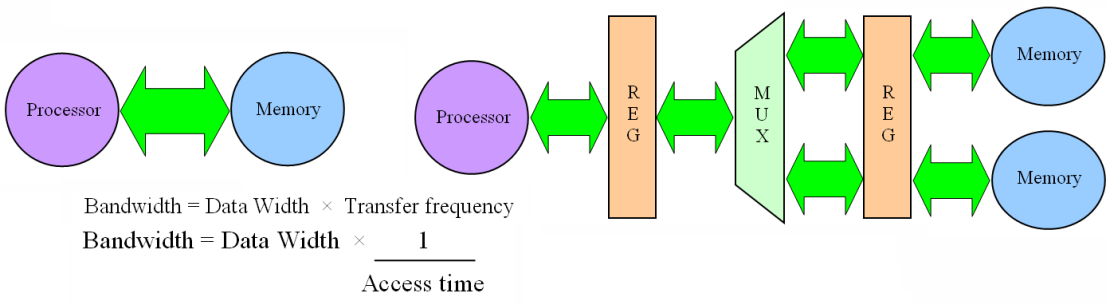


Figure 32 : von-Neumann architecture, Normal (left) and Wide (right) memory architectures

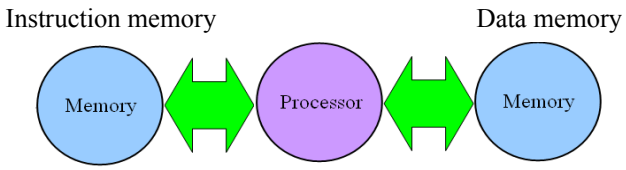


Figure 33 : Harvard architecture

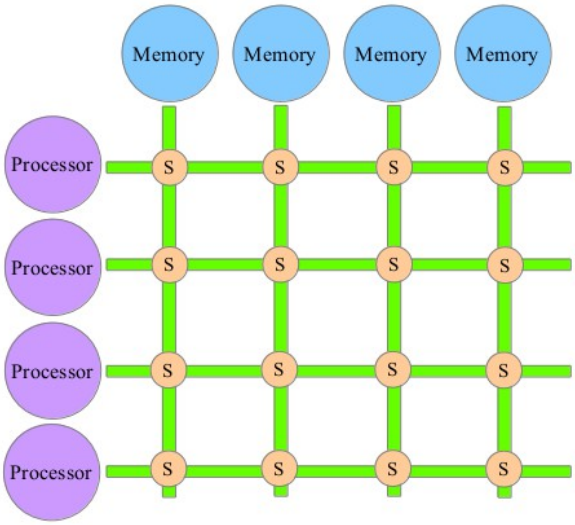


Figure 34 : crossbar switch

We can extending this idea e.g. a 256bit data bus allowing 16 instructions to be read at the same time. However, increasing these attributes also tends to result in increased signal noise levels i.e. increases the chance of SEU, reducing the system's reliability.

Therefore, rather than trying to increase the width or speed of the bus we can duplicate these communication links, allowing multiple components to transmit and receive data at the same time. This is commonly seen in the Harvard computer architecture, as shown in figure 33. Unlike the von-Neumann architecture where instructions and data are stored in the same memory, the Harvard architecture has two separate memory devices, one storing only instructions, the other only data. This has the advantage that when an instruction is storing its result to data memory, the processor is free to read the next instruction from the instruction memory i.e. overlap operations, “pre-fetch” the next instruction. Again, extending this idea, in systems with multiple processors we can move from a bus based architecture to one based on a crossbar switch, allowing different data paths to be dynamically switched between processing and memory elements, as shown in figure 34. Here any processor can access any memory by selecting the correct switching node (S), allowing up to four simultaneous (parallel) data transfers to occur at the same time. However, this assumes that different processors don't need to communicate to the same memory at the same time, if they do then again we hit a bottle-neck i.e. someone has to wait :).

Appendix A : delay calculation

$$\text{Instructions executed} = 2 + (1 + (3\times256)+4) \times CNT$$
$$= 2 + 773 \times CNT = 11 \times 10^3$$

$$CNT = (11 \times 10^3) \div 773 = 15$$

Delay MACRO

```
define(delay, `
  move   $1
  store $2

delayLoop$3:
  move 0
innerLoop$3
  sub 1
  jumpz outerLoop$3
  jump innerLoop$3

outerLoop$3:
  load $2
  sub 1
  store $2
  jumpnz delayLoop$3'
)
```

This macro is called within the program as:

```
delay(15, COUNT, 1)
```

Note, the first parameter is the outer loop count value, second the address of the variable used to store it, symbolic name COUNT in this example, and lastly the delay ID, used to ensure unique label names if this macro is used multiple time within a program.

Appendix B : testBit macro

```
define( testBit, `
  move 0
  store $1
  load $2
  and $3
  jumpz exit_$3
  move 1
  store $1
exit_$3:'
)
```

This is called with the program as:

```
testBit( TX, CHAR, 0x01 )
testBit( TX, CHAR, 0x02 )
testBit( TX, CHAR, 0x04 )
testBit( TX, CHAR, 0x08 )
testBit( TX, CHAR, 0x10 )
testBit( TX, CHAR, 0x20 )
testBit( TX, CHAR, 0x40 )
testBit( TX, CHAR, 0x80 )
```

Note, the first parameter is the address of the variable used to store the value of the bit to be transmitted, in this example symbolic name TX. The second parameter is the address of the character being processed, in this example symbolic name CHAR. Finally the value of the bitwise mask. This value must only contain one logic 1, selecting the bit position to be transmitted. The final parameter is also used to generate an unique labels within the macro, this assumes that the macro is only called once for each bit mask.

Appendix C : shiftRight macro

```
define( shiftRight, `
  move 0
  store $1
div_loop:
  load $2
  sub 2
  store $2
  and 0x80
  jumpnz div_exit
  load $1
  add 1
  store $1
  jump div_loop
div_exit:
  load $1
  store $2'
)
```

This is called with the program as:

```
shiftRight( TMP, CHAR )
```

Note, the first parameter is the address of the temporary variable used to store intermediate values produced during the division, in this example symbolic name TMP. The second parameter is the address of the character being processed, in this example symbolic name CHAR. The data stored at this address will be overwritten with the final result. This macro does not produce unique label names, therefore, it is assumed it is only called once within a program.

Appendix D : HELLO WORLD code (macros)

```
# INTERFACE - GPIO: ADDR 0xFC
# Q7 to Q1    /* NU */
# Q0;         /* TX */

start:
     move   0x01              # set default state = 1
     store  GPIO

     move   0x00              # zero char count
     store  charCount

txLoop:
     load   charCount         # load char count
     add    message           # add base offset
     store  4 txChar          # overwrite load address

txChar:
     load   0x00              # read char
     jumpz  exit              # finish if char=NULL

     store  txBuff            # buffer char
     move   0x08              # set bit count
     store  txBitCnt

     load   charCount         # load char count
     add    0x01              # inc
     store  charCount

     move   0x00              # start bit = 0
     store  GPIO

     delay(15, delayCnt, 1)

txCharLoop:
     load   txBuff            # load buffer char
     and    0x01              # mask bit
     store  GPIO              # update port

     delay(15, delayCnt, 2)

     load   txBitCnt          # load bit count
     sub    0x01              # dec
     store  txBitCnt
     jumpz  stopBit           # finished, TX stop bit

     shiftRight(tmp, txBuff)

     jumpu txCharLoop         # repeat until all bits TX
```

```
stopBit:
     move   0x01              # stop bit = 1
     store  GPIO

     delay(15, delayCnt, 3)

     jump   txLoop            # repeat

exit:
     jump   exit              # trap

# VARIABLES

charCount:
     move   0x00
txBuff:
     move   0x00
txBitCnt:
     move   0x00
delayCnt:
     move   0x00
tmp:
     move   0x00

# DATA CHARACTERS TO DISPLAY

message:
     move   0x48      # H    - 0100 1000
     move   0x45      # E    - 0100 0101
     move   0x4C      # L    - 0100 1100
     move   0x4C      # L    - 0100 1100
     move   0x4F      # O    - 0100 1111
     move   0x20      # SP   - 0010 0000
     move   0x57      # W    - 0101 0111
     move   0x4F      # O    - 0100 1111
     move   0x52      # R    - 0101 0010
     move   0x4C      # L    - 0100 1100
     move   0x44      # D    - 0100 0100
     move   0x0A      # CR   - 0000 1010
     move   0x0D      # LF   - 0000 1101
     move   0x00      # NUL  - 0000 0000
```

Appendix E : HELLO WORLD code (“subroutines”)

```
# INTERFACE - GPIO: ADDRESS 0xFC
# Q7 to Q1    /* NU */
# Q0;         /* TX */

# MEMORY

# delay count    0x37 - 55
# divide count   0x38 - 56
# bit  count     0x39 - 57
# char count     0x3A - 58
# char buffer    0x3B - 59
# data string    0x50 to 0x5B
# gpo            0xFC

move   0x01      # set default state = 1
store  0xFC

move   0x00      # zero char count
store  58

load   58        # load char count
add    80        # add base offset
store  4 0x07    # overwrite load address

load   0x00      # read char
jumpz  47        # finish if char=NULL
store  59        # buffer char
move   0x08      # set bit count
store  57

load   58        # load char count
add    0x01      # inc
store  58

move   0x00      # start bit = 0
store  0xFC

move   17        # load return address
jump   60        # jump to delay

load   59        # buffer char
and    0x01      # mask bit
store  0xFC      # update port

move   22        # load return address
jump   60        # jump to delay
```

```
load   57        # load bit count
sub    0x01      # dec
store  57
jumpz  42        # finished, TX stop bit

move   0x00      # zero divide count
store  56

load   59        # load char, sub 2
sub    0x02
store  59
and    0x80      # did subtraction generate a negative
jumpnz 39        # number

load   56        # no, increment divide count
add    0x01
store  56
jumpu  30

load   56        # replace TX char with one divided by 2
store  59
jumpu  19        # repeat until all bits TX

move   0x01      # stop bit = 1
store  0xFC

move   44        # load return address
jump   60        # jump to delay

jump   04        # repeat

jump   47        # trap

move   0x00      # zero variable memory
move   0x00
move   0x00
move   0x00
move   0x00
move   0x00
move   0x00
move   0x00
move   0x00
move   0x00
move   0x00
move   0x00

# DELAY (ADDR 60)

add    0x02      # generate return address
store  8 72      # update jump with return address
move   15        # save loop count
```

```
store  55

move   0x00      # load count
sub    0x01      # dec delay loop
jumpz  68        # exit if 0
jump   65        # repeat

load   55        # dec loop count
sub    0x01
store  55
jumpnz 64        # repeat if not zero

jump   0x00      # return

move   0x00      # pad
move   0x00
move   0x00
move   0x00
move   0x00
move   0x00
move   0x00

# DATA CHARACTERS TO DISPLAY (ADDR 80)

move   0x48      # H    - 0100 1000
move   0x45      # E    - 0100 0101
move   0x4C      # L    - 0100 1100
move   0x4C      # L    - 0100 1100
move   0x4F      # O    - 0100 1111
move   0x20      # SP   - 0010 0000
move   0x57      # W    - 0101 0111
move   0x4F      # O    - 0100 1111
move   0x52      # R    - 0101 0010
move   0x4C      # L    - 0100 1100
move   0x44      # D    - 0100 0100
move   0x0A      # CR   - 0000 1010
move   0x0D      # LF   - 0000 1101
move   0x00      # END  - 0000 0000
```