16-Bit processor with a memory mapped UART interface

Documentation

Matija Žagar 29.9.2021

Contents

1. INTRODUCTION

The designed circuit is a 16-bit processor with access to the UART(„Universal Asynchronous Receiver-Transmitter“) module for receiving and transmitting data via the UART protocol. The Harvard architecture was used, meaning the processor contains separate memories for data and instructions. Programming the processor is done using an assembly-like language whose instructions perform all the tasks needed to manage registers, calculations, and memory. The UART module is accessed in the same way as normal memory, with the difference being that it is located at a virtual address from which we load received or write data to be transmitted using the UART protocol.

2. Project overview

The project consists of 4 main components. These components are the processor or the Central Control Unit (CPU), ROM memory, RAM memory and UART module. The diagram of the relations of these components is shown in Figure 1.

ROM memory is used to store the 15-bit instructions the CPU reads from it depending on the instruction counter (program counter). The only way to write to this memory is in the VHDL module before project synthesis because the processor itself does not have the ability to save new instructions.

RAM is used to store important program data that is not used intensly because it takes at least one processor cycle to access it while access to the registers is instant.

The UART module is a way for the processor to communicate with any external device that can transmit a signal via the UART protocol and enables a wider range of possibilities and interactivity of this project.

The CPU or processor is the main controller of this project that processes the instructions from ROM and controls the UART module.

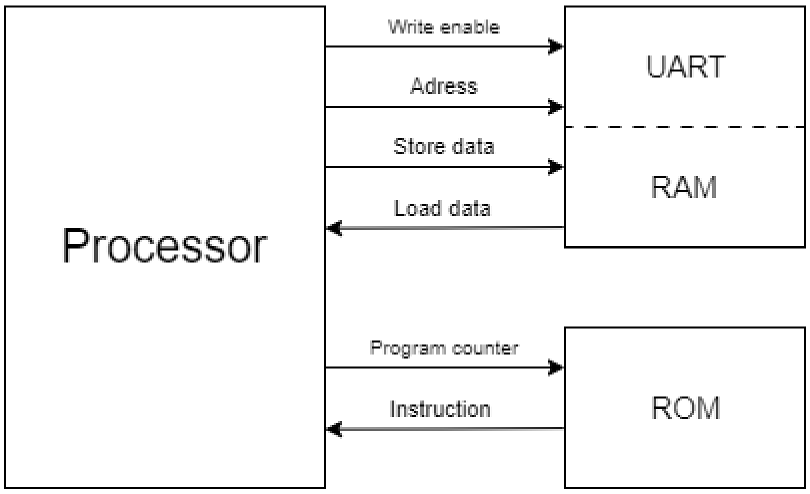
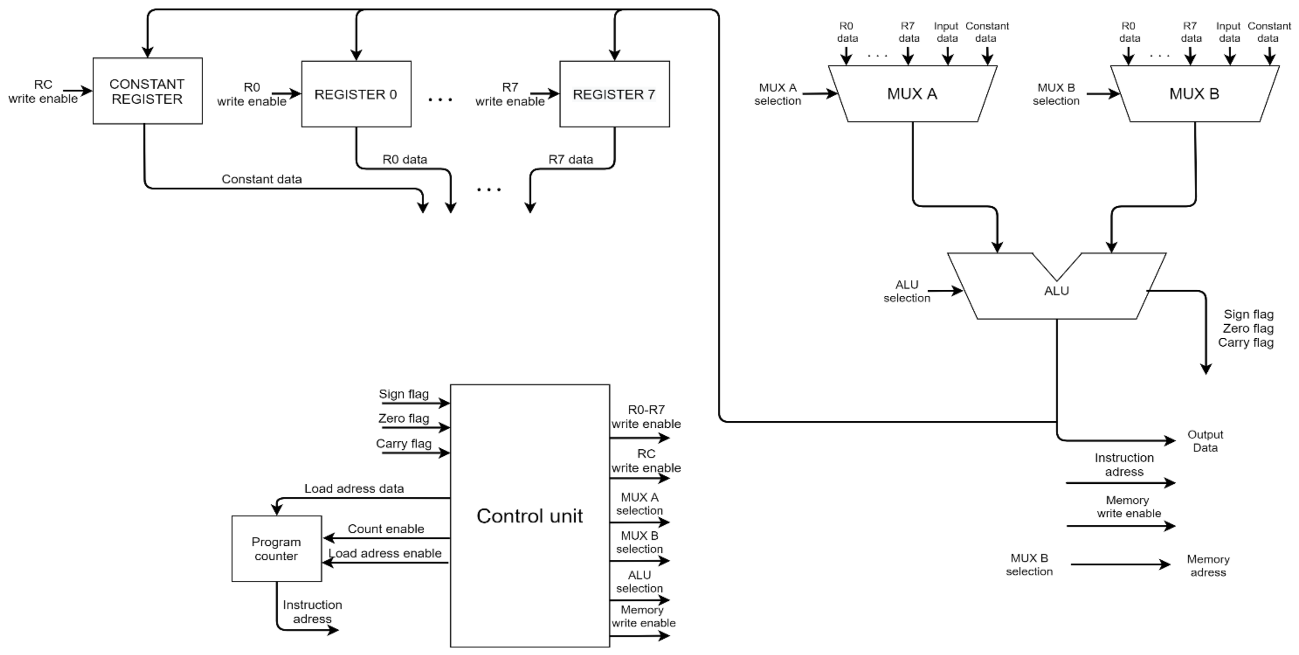


Figure 1.

2.1. Central Processing Unit (CPU)



Slika 2.

The central processing unit is the part of this circuit that manages calculations, writing to memory and reading it. Each processor cycle (clock) reads one 15-bit instruction from the ROM and executes it. Execution of the instruction begins with the selection of the operands using multiplexers (MUX) A and B. The outputs of the multiplexers are the inputs of the Arithmetic Logic Unit (ALU). The ALU performs the operation specified in the instruction and forms the result and the status bits or flags that describe the result. Flags indicate that the result is zero (ZERO flag), that the result is negative (SIGN flag) or that the result had a bit transfer (CARRY flag). After that, the result is forwarded to the general purpose registers, the constant register, the program counter and the processor output. The CPU then, according to the instruction code, sends permission signals that allow the container to store the data. To the processor output is then fowarded the 16-bit output data towards RAM, the 16-bit RAM address to the RAM, the 16-bit value of the program counter which is used as the memory address for the instruction ROM and the write enable signal for the RAM memory.

2.2. Instruction set architecture

The available processor instructions are divided into arithmetic instructions, jump instructions, and instructions for reading and writing to memory. Because the VHDL hardware description language was used to describe the assembly language, the instructions were defined as symbolic constants for ease of use. This way in the ROM an example of the instruction would look like this:

```
mov &R0 &R1 &N3
```

The '&' character is used to concatenate constants that can be instructions such as "mov", "jmp", "ld" and others. General purpose register names numbered from 0 to 7 are also constants, they would be listed as follows: "R0", "R5" etc. Another special constant called "N3" is used to indicate that this part of the instruction is not used, but these bits must exist for the instruction to be of a given length. In the jump instructions we use constant numbers that must be 9 bits long ie. 3 octal characters.

2.2.1. Arithmetic instructions

These instructions are calculated directly by the arithmetic logic unit. The instruction code is as follows:

```
I I I I I I  Z Z Z  X X X  Y Y Y
```

Where the letters 'I' represent the 6-bit instruction code and the letters 'Z', 'X' and 'Y' represent the addresses of general purpose registers used in the function. In all instructions, the result is stored in register 'Z'. Registers 'X' and 'Y' act as inputs (operands). If an instruction uses only one operand, the other can be defined arbitrarily. The syntax "RZ <- [RX]" means that the contents of register 'X' are entered in register 'Z'.

Table 1. Arithmetic instructions

Instruction	Function	Instruction code
`mov    Rz, Rx`	RZ <- [RX]	000000
`add    Rz, Rx, Ry`	RZ <- [RX] + [RX]	000001
`sub    Rz, Rx, Ry`	RZ <- [RX] - [RX]	000010
`and1   Rz, Rx, Ry`	RZ <- [RX] & [RX]	000011
`or1    Rz, Rx, Ry`	RZ <- [RX] | [RX]	000100
`not1   Rz, Rx`	RZ <- not [RX]	000101
`inc    Rz, Rx`	RZ <- [RX] + 1	000110
`dec    Rz, Rx`	RZ <- [RX] - 1	000111
`shl1   Rz, Rx`	RZ <- shl [RX]	001000
`shr1   Rz, Rx`	RZ <- shr [RX]	001001
`neg    Rz, Rx`	RZ <- - [RX]	001010
`ashr   Rz, Rx`	RZ <- ashr [RX]	001011

2.2.2. Jump instructions

Jumps are an essential part of any programming language because they allow the inevitable actions of executing code inconsistently. Using the same we can achieve essential aspects of the program such as conditional branching or loops. The jump instruction code has the following format:

```
I I I I I I  A A A A A A A A A
```

Where the letters 'I' represent the 6-bit instruction code and the letters 'A' represent the 9-bit address to be jumped to. The status bits generated by the ALU are used as jump conditions. The ZERO status bit is marked with the letter 'z', the SIGN bit with the letter 's' and the CARRY bit with the letter 'c' in the instruction name, and if we want the jump to occur if that particular bit is not active, the name also contains the letter 'n'. Thus the instruction for an unconditional jump to address 0 is used like this:

```
jmp &"000000"
```

Conditional jump instruction if the ZERO flag is active:

```
jmpz &"000000"
```

The instruction to jump conditionally if the ZERO flag is not active would look like this:

```
jmpnz &"000000"
```

Table 2. Jump instructions

Instruction	Function	Instruction code
`jmp ADDR`	PC <- ADDR	010000
`jmpz ADDR`	if Z=1 PC <- ADDR	010001
`jmps ADDR`	if S=1 PC <- ADDR	010010
`jmpc ADDR`	if C=1 PC <- ADDR	010011
`jmpnz ADDR`	if Z=0 PC <- ADDR	010101
`jmpns ADDR`	if S=0 PC <- ADDR	010110
`jmpnc ADDR`	if C=0 PC <- ADDR	010111

2.2.3. Memory access instructions

To access RAM we use the instructions STORE (save) and LOAD (load). In order to be able to define which RAM address we are accessing, in the instruction we will specify in which register the address is stored.

The LOAD instruction (code "ld") loads the value stored at the RAM address located in the 'Y' register into the 'Z' register. The STORE instruction (code "st") saves the value from register 'X' to the RAM address written in register 'Y'.

Table 3. Memory access intructions

Instruction	Function	Instruction code
`ld Rz, Ry`	RZ <- [[RY]]	100000
`st Ry, Rx`	[RY] <- [RX]	110000

2.2.4. Instructions for entering constant values

In order to be able to use user constants, there are commands to enter an arbitrary constant. Since the instruction is 15 bits long and 6 bits are reserved for the instruction code, it is not possible to enter a 16-bit number using one instruction. Therefore, the LOAD CONSTANT instruction (named "`ldconst`") has the following format:

```
I I I I I I  B  D D D D D D  D
```

Where the letter 'I' indicates the 6 bit code of the instruction, the letter 'B' indicates one control bit that determines whether the byte we enter is upper or lower, ie whether we enter bits 0. to 7. (the bit is set to 0) or bits 8 to 15 (bit is set to 1) and the letters 'D' denote 8 data bits (one byte). If the control bit is set to '0', then the upper register byte will be set to 0 and the lower byte to the 8-bit data value from the instruction. If the control bit is set to '1', the data from the instruction will be written to the upper byte and the lower byte of the register will remain unchanged, which must be kept in mind if we do not want to keep the data in the lower byte. The instruction is designed in this way so that two commands do not have to be used to write constants that require 8 or fewer bits to write. To store these constants, general purpose registers are not used, but a special constant register that cannot be accessed other than via the `ldconst` and `mvcreg` commands.

Once we have saved the desired constant in the register for constants, we can move it to a general purpose register with the command `mvcreg` (meaning "move constant to register"). It stores the value from the register for constants ("RC") to the register RX.

Table 4. Instrukcije za rukovanje konstantama

Instruction	Function	Instruction code
`ldconst lower/upper bit,DATA`	RC <- DATA	100110
`mvcreg Rx`	RX <- [RC]	100111

2.3. UART module

In order for the processor to have a way of communicating with external circuits, it has the ability to access a module that performs the receiving and transmitting of data via the UART protocol.

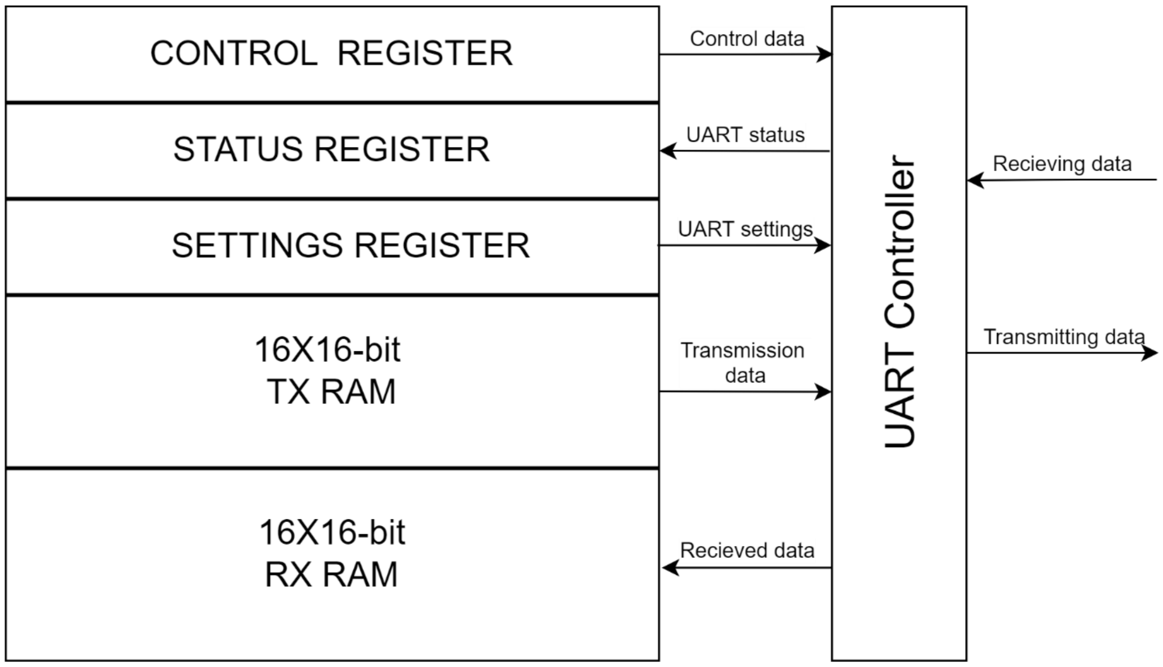


Figure 3. UART module diagram

Using the already described LOAD and STORE instructions, the processor manages the UART module via the control register and the UART settings register, and enters the data for transmission, ie reads the received data at the virtual addresses assigned to them. The UART controller receives data (if the receive permission bit is active) and saves it in RX RAM and transmits the data (if the broadcast permission bit is active) stored in TX RAM. The processor defines how many data words it wants to keep in RX RAM by setting a value in the settings register, and when that number is exceeded, the oldest data starts to be overwritten. For transmission, a part of the settings register is also defined, which indicates how many data words from those stored in TX RAM we want to send before they start to be repeated, starting from the most recently sent.

The UART protocol standard used to receive and transmit data is as follows:

First one Start bit with the value of a logic '0' is transmitted, then 8 data bits are transmitted and finally one stop bit with a value of '1'. If this standard is not met, the information cannot be considered valid.

2.3.1. UART module registers

The **control register** of the UART module can be read and written to at the virtual address x "1000" and has the following division**:**

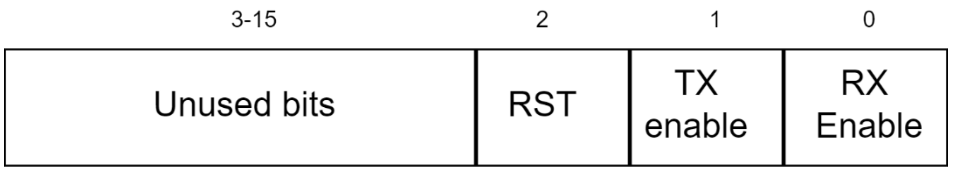


Figure 4. Meaning of the control register bits

RX Enable or receive permission, if active, allows the circuit to receive data and write to RX RAM.

TX Enable or allow permission allows the assembly to send data stored in TX RAM.

The RST or reset bit ,which is activated with the logic '1' , sets the transmit and receive counters to zero. This reset is of the "Clear on write" type, which means that, if activated, it will set to logic '0' in the next cycle, saving a processor cycle where it would need to be cleared.

The UART module settings register can be read and written to and is located at the virtual address x "1001". It is designed as follows:

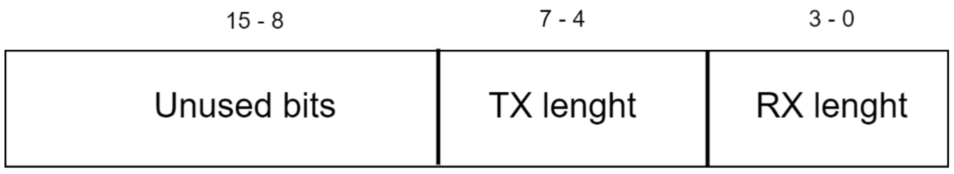


Figure 5. Meaning of the settings register bits

RX lenght is a 4-bit number that indicates how many words of RAM are used. The maximum possible size is 16, and the smallest is 1. When all the words that are allowed to be written on (RX lenght limit) are filled in, the data starts from the beginning.

TX lenght is a 4-bit number that indicates how many first words of those stored in TX RAM will be transmitted. When all words have been sent, they start being retransmitted from the beginning of TX RAM (if the write permission signal is active).

The status register contains information about the operation of the UART circuit and can only be read. It is located at the virtual address x "1002" and stores the following data**:**

15 - 10	9 - 6	5 - 2	1	0
Unused bits	TX position	RX position	TX busy	RX busy

Figure 6. Meaning of the status register bits

RX busy or „the receiver is busy“ bit indicates that data is currently being received.

TX busy indicates that data is currently being transmitted.

TX and RX position are 4-bit numbers that indicate which word is currently being transmitted (in the case of TX position) or at which position the next received word will written next.

3. Program for calculating over UART

Using all the listed functionalities of this circut, a program in the previously defined assembly language was written. This program handles simple calculation requests received via the UART protocol. When the program starts, it waits to receive the first number, the operator and then the second number on which the operation will be performed. The ';' sign indicates the end of the request, and after it has been recieved, processing and calculation begins. When the result is ready, it is transmitted letter by letter via UART protocol to the transmittion output bus of the processor. Since the processor uses a 16-bit architecture, the maximum range of numbers it can store is from -32768 to 32767,consequently, if the numbers or the result are outside that range, incorrect data will be sent.

The supported operations are addition (operator '+'), subtraction (operator '-'), multiplication (operator '*') and division (operator '/').

Transaction example:

The following string of ASCII characters is received on the UART receiver:

```
1000/50;
```

The processor would then perform the required calculation and send the following string of characters to the transmitter using the UARt protocol:

```
20
```

4. Conclusion

The project that has been realized is a very powerful circut for executing user programs. The developed assembly language is very important for creating an abstraction around the implementation of instructions, which also brings the programming of this assembly closer to the user. The biggest limitation of this processor is the 16-bit word length, which significantly limits its use for many mathematical functions and the like. This limit equally affects the amount of working memory we can assign to the processor and limits it to 64KB. The great advantage of describing this circuit in the VHDL hardware description language is that it can be implemented on most FPGA boards even though it was developed and tested on the Zync ZIBO-7000 FPGA development system.