The Moncky computer

Kris Demuynck

17th August 2022

Abstract

This document describes the architecture and implementation of the Moncky computer which is based on the Moncky-3 processor. This document is a work in progress and will be updated regularly. The home of the Moncky project is

https://hackaday.io/project/181269-the-moncky-project

and

https://gitlab.com/big-bat/moncky

for all source code.

Copyright

Kris Demuynck

The Moncky computer

Contents

Chapter 1

Introduction

This document describes the Moncky computer which was constructed around the Moncky-3 processor. It contains 128 Kib RAM, 128 KiB input/output memory, and SPI interface, a VGA interface, and a PS/2 interface for a keyboard.

The SPI interface is connected to an SD card reader so SD cards can be used as 'floppies'.

It can be used to play games or program in a simple interpreted programming language.

The next sections will describe each element of the computer separately.

The home page for the Moncky project is:

https://hackaday.io/project/181269-the-moncky-project

All code and documentation can be found in gitlab:

https://gitlab.com/big-bat/moncky

Chapter 2

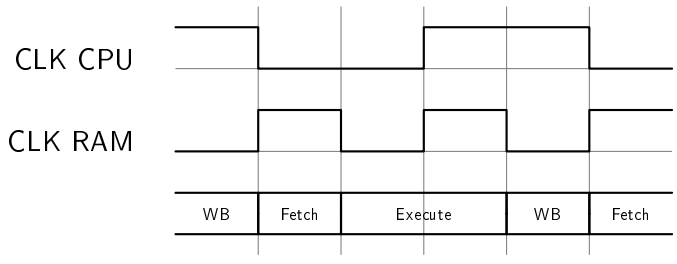
The RAM

In order to make the computer flexible, it should be possible to load a program into memory (as data) and execute it. The Harvard architecture does not allow that since data and code are stored in different locations.

Therefore the Harvard architecture is altered in the Moncky-3 computer so that code and data can be in the same RAM. This is done using dual-port RAM. In this way the processor does not need to be aware that code and data are stored in the same location. It can however load code into memory and then jump to it. The dual-port RAM has one port for reading and writing data and another for reading the code.

The dual-port RAM is made with BRAM on the FPGA. There is however a big drawback doing this: BRAM is synchronous memory. This means it will only read and write data at the rising edge of the clock. For writing this is a good thing because the address can stabilise before the data is written to a specific location. But for reading it causes an extra delay. For instance, the Moncky-3 processor will put the address of the next instruction at a falling edge of the clock. It has to wait for the next rising edge to be able to read the instruction at that address.

In order to overcome this problem, there are two possible solutions. The first is to double the clock speed of the RAM and let it respond to the falling edge of its clock. This results in the following:

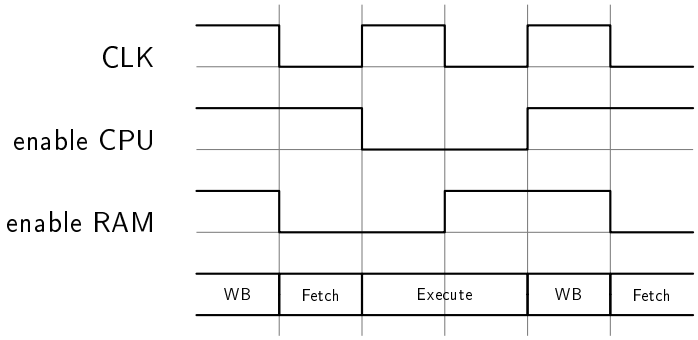


There are now 3 distinct phases: fetch, execute, and write back (WB). In the first phase the processor will put the value of PC on the address bus. The memory will output the instruction at the end of the fetch cycle. Now the processor can execute the instruction. If the processor needs to read data from the RAM, the data will become available in the WB phase. If the processor needs to write data to the RAM, this is also done during the WB phase. Remark that these phases have a high correlation with the Von Neumann cycle. There is however no distinct decode phase.

Doubling the clock of the memory (or decreasing the cpu clock speed) works very well, but on an FPGA this poses a problem for the development tools. There are now two 'clock domains'

that are mixed together. The development tools try to check all the timing constraints. But this cannot be done if a mixture of clock domains is present, unless there is a buffer in between them (this problem is also known as 'Clock-Domain Crossing'). Therefore another approach is used.

In this approach both the CPU and the RAM get an 'enable' signal. Their clock speed is the same, but the enable signals will enable the clock pulse when needed. This is depicted in the following timing diagram:



The RAM is now configured to respond to the rising edge of the clock. But data will only be written when the enable RAM signal is high. This creates the same effect as before, but now there is only one clock domain which makes it easier to check all the timing constraints.

Chapter 3

Clock domains

As discussed before having different clock domains can solve problems, but they can also create them. The RAM and CPU are in the same clock domain, but other components need other clock speeds. The following clock domains are present in the Moncky-3 computer:

- CLK100MHZ: this is the base clock of the FPGA. It is used to derive the other clocks
- clk_cpu: this is the clock given to the cpu. It currently runs at 20 MHz. Since the enable signal is only active once in two clock cycles the cpu effectively runs at 10 MHz.
- clk_mem: this clock is the same as clk_cpu.
- clk_video: this clock is used to generate the VGA signal. For a 640x480 resolution a frequency of 25 MHz is needed.
- clk_io: this clock is given to the keyboard controller, the counters, and the SPI controller. Its frequency is 12.5 MHz.

The FPGA contains a clock generator that can derive the clocks. It outputs a 'locked' signal that becomes high when all clocks are in sync. This signal is used to drive the reset pin of all components so they all reset as long as the clocks have not been synchronised yet.

Chapter 4

The ROM

Having one memory for data and code is very useful, but when the computer boots up, some code must already be available so it can start up. To enable this a ROM is added to the system that will 'shadow' the RAM. When the processor reads code from the lower part of the memory (addresses 0x0000 till 0x0FFF), the data doesn't come from RAM, but from the ROM. There is a MUX in the system that will select the ROM when the 4 most significant bits of the address are equal to 0.

When the processor writes data to the RAM it will be stored in the RAM. Even if the address is within the ROM range the data still goes to RAM. This is used in the boot sequence. The ROM shadowing can be turned off. If there is code in the RAM that code will take its place.

The ROM contains the following code:

- clear the screen
- plot a dot on the screen
- draw a character on the screen
- print a string
- print a hexadecimal number
- scroll the screen up
- wait for a key press
- detect and initialise an SD card
- read a block from the SD card
- read a file from a FAT16 file system on an SD card

When the computer boots the system will show a splash screen and start looking for an SD card. If no SD card is present the computer shows an error and reboots after a key press. If an SD card is detected, the file 'MONCKYOS' is expected to be on the first partition on a FAT16 file system. Currently the partition should be smaller than 32MB. The file is loaded at address 0x0000 in memory. After this the ROM shadowing is turned off and the computer jumps back to address 0x0000.

The ROM shadowing can be turned off by writing the number 0xAAAA in the location 0xFFFF (in normal RAM memory). This means that this location cannot be used for any other purpose.

Chapter 5

Input/output

In order to have as much memory available as possible the input/output is not memory mapped. In stead the io_request pin together with the opcodes 'in' and 'out' are used to access the input/output. This creates a new address space of 65536 entries. A specific i/o component in the computer regulates the data flow between the i/o address space and all the components.

Next to input/output the i/o address space is also used by the firmware to store data. In the next list, they are marked with a '*'.

The following addresses are currently used (more information about the exact values is given per i/o component) (remark that the order of the values is a bit chaotic, due to the way the software and hardware evolved):

- 0x0000 - 0x7CFF: video screen buffer 0
- 0x7D00: video configuration
- 0x7D01: keyboard last key press
- * 0x7D02: cursor position (LSB=x, MSB=y)
- * 0x7D03: shift pressed
- * 0x7D04: background colour
- * 0x7D05 - 0x7D25 : saved image behind cursor
- * 0x7D26: graphics draw colour
- * 0X7D27: graphics draw buffer
- 0x7E00: SPI data
- 0x7E01: SPI control
- 0x7E02: LSB of usec
- 0x7E03: MSB of usec
- 0x7E04: LSB of msec
- 0x7E05: MSB of msec
- * 0x7E06: sd_type (1,2, or 3 for SD1, SD2, and SDHC resp.), high byte contains error if it occurred

- * 0x7E07 - 0x7E0B: sd_result (5 words containing the result to a command)
- 0x8000 - 0xFCFF : video screen buffer 1
- * 0xFD00 - 0xFDFF: fat_sector
- * 0xFE00 - 0xFE01: fat_startPartition
- * 0xFE02 - 0xFE03: fat_lengthPartition
- * 0xFE04: fat_sectorsPerFat
- * 0xFE05: fat_fatPosition
- * 0xFE06: fat_rootFolderPosition
- * 0xFE07 - 0xFE08: fat_rootFolderPosition
- * 0xFE09: fat_fileStart
- * 0xFE0A - 0xFE0B: fat_fileLength
- * 0xFE0C: fat_currentClusterNumber
- * 0xFE0D: fat_sectorsPerCluster
- * 0xFE0E: fat_currentSectorNumber

Chapter 6

The VGA controller

The VGA controller generates the signals to drive a VGA monitor at a resolution of 640x480 at 60 Hz. Due to the limitation of the video memory the pixels are grouped in 2x2 pixels. Only the middle 400 lines of the screen are used. This means that the effective resolution is 320x200. When displayed on a 4:3 monitor the pixels are square.

The VGA controller only has a graphics mode. There is no text mode. That means that text is rendered by drawing pixels. While this is much less efficient when only text is needed, it is much more simple and flexible. Text and graphics can for instance easily be mixed.

The values for the colour of each pixel are stored in a separate RAM (called VRAM). Eight bits are used per pixel as follows:

red	red	red	green	green	green	blue	blue

Since the memory contains 16 bit numbers, each memory cell holds the values of two pixels. The LSB defines the left pixel and the MSB the right pixel. This means that the video memory is at least 320x200 bytes long. This equals to 32000 words of 16 bit.

The total video memory however is 65536 words long. This means that 2 screen buffers can fit in the VRAM. This makes it possible to do double buffering animations. The idea is that the computer will write to one buffer while the other is shown on the screen. After the buffer is ready the VGA controller is asked to switch the active buffer. In this way a fluent animation can be done without flickering. Switching buffers should be done at the vertical retrace. The memory location 0x7D00 is used to specify which buffer needs to be displayed (0 or 1). The processor can write the right value in this location and the VGA controller will immediately switch to the new buffer.

In order to detect the vertical retrace of the VGA controller, it will send a hardware interrupt to the processor. The processor thus gets 60 interrupts per second from the VGA controller. This interrupt might also be used to do other tasks (like multitasking).

The VRAM is organised as follows:

- 0x0000 - 0x7CFF: video buffer 0
- 0x7D00: video control register (defines which buffer needs to be displayed)
- 0x7D01 - 0x7FFF: free for other uses (this address space is used for other I/O components and firmware variables)
- 0x8000 - 0xFCFF: video buffer 1
- 0xFD00 - 0xFFFF: available for other uses

Chapter 7

The keyboard controller

The keyboard controller is able to interface with a PS/2 keyboard. It is only capable of receiving data from the keyboard. This means that the LEDs on the keyboard do not function. When a key is pressed the keyboard will send the scan code of the key to the keyboard controller. The data is sent one byte at a time as follows: 1 start bit, 8 data bits, a parity bit, and a stop bit. Most scan codes are 8 bits long but there are also keys with a 2-byte scan code. When a key is released the keyboard will first send a 'break' code followed by the scan code of the key.

The keyboard controller is split into 3 parts: the keyboard debouncer, keyboard interface, decoder, and buffer.

7.1 The keyboard debouncer

A PS/2 keyboard generates signals for the key pressed as well as a clock. This clock runs at about 10KHz. Since the wires between the keyboard and the computer are quite long and since the computer runs at a much higher speed the signals can be distorted. Therefore they must be debounced. When the value of the input doesn't change for 2.56 usec, the value is passed on to the keyboard interface.

There is a debouncer for both the data and the clock input.

7.2 The keyboard interface

The keyboard interface receives bytes of data from the keyboard. It does this by waiting until 11 bits are received. The start bit, stop bit, and parity is checked. When a byte was successfully received it is presented at the output and a 'ready' signal is put high.

The keyboard interface is in fact a finite state machine having 2 states: 'waiting' and 'receiving'. There is a watchdog timer that will force the machine into 'waiting' mode when the machine stays too long in 'receiving' mode.

If an error occurs the keyboard interface will set its 'error' output to high.

7.3 The keyboard decoder

The keyboard decoder will receive the bytes of the keyboard interface and turn them into one 16-bit number. Bits 8-0 contain the scan code of the key that was pressed or released. Bit 9 will indicate if the key was pressed or released.

The keyboard decoder is implemented as a finite state machine. It has 4 states: 'waiting', 'extended', 'released', 'extended released'. In the first state the decoder is waiting for an incoming byte. If this byte indicates an extended scan code the state turns into 'extended'. If a break code is received the state changes to 'released' or 'extended released' according to the current state.

When a complete scan code has been detected the value is passed to the keyboard buffer.

7.4 The keyboard buffer

The keyboard buffer contains a circular FIFO queue. It can hold up to 15 key strokes. The keyboard buffer will store all incoming key strokes from the keyboard decoder. When the processor reads from keyboard register (mapped at address 0x7D01) the keyboard buffer will deliver the data and remove it from the buffer.

This component takes away the need for an interrupt whenever a key is pressed. A drawback is that the keyboard buffer will not be emptied automatically when a reset occurs. This needs to be done in software by reading the buffer until it outputs a value of 0.

Chapter 8

The counters

In order to measure time, two counters run in hardware and are available to the processor. The first counter counts microseconds since startup and the other count milliseconds since startup. Both counters are 32 bits long and are stored in the following locations in i/o memory:

- 0x7E02: LSB of usec
- 0x7E03: MSB of usec
- 0x7E04: LSB of msec
- 0x7E05: MSB of msec

The counters can be used by the processor when it needs to wait for a specific amount of time, or to detect a timeout.

A possible extension of this component would allow it to send an interrupt to the processor in a configurable amount of time. This is currently not implemented.

Chapter 9

The SPI controller

The SPI controller can control up to 4 SPI slaves. The configuration register (located at 0x7E01) contains the following data:

d7	d6	d5	d4	d3	d2	d1	d0	ss3	ss2	ss1	ss0	ready	m1	m0	transfer

The values d7 - d0 define the speed of the SPI clock. If one wants to set the clock to a certain frequency, the value must be set according to the following formula:

$$d = \frac{12,500,000}{2 \cdot f} - 1$$

The values ss3 - ss0 are the slave-select signals for each of the slaves. At most one of these bits should be 0 at any time.

The values m1 - m0 define the SPI mode (CPHA and CPOL).

The ready bit will be put to 1 when the SPI controller is waiting to send and receive data.

When the processor puts the value of 'transfer' from 0 to 1 (rising edge), the SPI controller will put the 'ready' bit to 0 and start sending the data that is present in location 0x7E00. The data in this location is automatically overwritten by the received data. After sending and receiving the 'ready' bit is put back to 1. The transfer bit will always be read as 0.

Chapter 10

Statistics

The Moncky-3 computer is implemented on an Xilinx Spartan 7 FPGA. It uses the following resources:

```
Report Cell Usage:
+------+---------+------+
|      |Cell     |Count |
+------+---------+------+
|1     |clk_wiz  |     1|
|2     |CARRY4   |    39|
|3     |LUT1     |    77|
|4     |LUT2     |   127|
|5     |LUT3     |    92|
|6     |LUT4     |   174|
|7     |LUT5     |   171|
|8     |LUT6     |   735|
|9     |MUXF7    |    73|
|10    |MUXF8    |    19|
|11    |RAM32M   |     2|
|12    |RAMB36E1 |    66|
|18    |FDRE     |   593|
|19    |FDSE     |    16|
|20    |IBUF     |     4|
|21    |OBUF     |    19|
+------+---------+------+
```

It consumes 0.251 W of power.