Implementing the beta machine on a Terasic DE10 SoC + FPGA development board

Auteur : Polet, Quentin
Promoteur(s) : Fontaine, Pascal; Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil électricien, à finalité spécialisée en "electronic systems and devices"
Année académique : 2020-2021
URI/URL : http://hdl.handle.net/2268.2/11612

Implementing the beta machine on a Terasic DE10 SoC+FPGA development board

Master's thesis carried out to obtain the degree of Master of Science in Electrical Engineering by Polet Quentin

University of Liège - School of Engineering and Computer Science

Academic year 2020-2021

Contents

Acknowledgements

I am grateful to all those who helped and supported me during this work. In particular professors Pascal Fontaine and Laurent Mathy who were very regular in their follow-up by participating with me in regular meetings. In addition to that, they were available to answer all my questions during the whole duration of this work, which was very reassuring and helpful. I would like to thank Professor Fontaine in particular for his very careful review of this work and for the many pertinent suggestions that resulted from it. But also for putting me in touch with people who could guide me on certain choices for this work.

These two people are Alexander Nadel who is a research scientist at Intel Israel and Gyuszi Suto a principal engineer at Intel Oregon. I thank them greatly for the time they devoted to me through several emails and a video conference meeting during which I could present my work and discuss with them. They specifically helped me in the choices of how I could improve the confidence one could have in a Central Processing Unit (CPU) such as the one developed in this work.

I would also like to thank Romain Mormont, a PhD student at the University of Liège in machine learning who provided an assembler software that outputs beta machine code. He also took the time to explain me how it works and how to use it. One last person directly involved in this work is Gauderic Schnackers, a physics engineer who took the time to try out the system designed in this work by programming a 2D video game on it. I thank him for this and for his friendship.

Finally, I could not end my thanks without thinking of all my family and friends who have always been very supportive and who knew how to entertain me when I wanted to work too much.

Introduction

Although some liberties have been taken at the end of the work, the main objective of this master thesis is to provide a laboratory tool for the computation structures course INFO0012-2 at the University of Liège. Indeed, the professors of the course needed a replica of the beta machine, an Harvard architecture based CPU that is learned throughout the course. This machine had to be physically implemented so that the students could really take it in hand rather than using simulations. In order to design it in hardware, it was decided to work on an Field Programmable Gate Array (FPGA), a Cyclone V from Altera to be precise. In fact, everything was done on a development board, the DE10-Nano from Terasic. This allows to focus almost exclusively on the logic and structural implementation rather than on all the electrical details intrinsic to such implementations.

This document therefore starts with a first chapter containing an enumeration of the different functionalities of this board as well as a description of the generalities about FPGAs. A quick comparison with other similar technologies is also made. Then, an overview of the two parts of the FPGA, starting with the programmable logic followed by the ARM processor part is done. The existing interconnection between these two parts is also described afterward. The last part of this chapter is a comparison between the different memories available on the board and on the FPGA.

After introducing the hardware, an introduction to the FPGA development flow is given. In this chapter, a brief discussion on the different specific tools used throughout this work is made. The specificities of the programming language used, i.e., Verilog, are also presented. The chapter finally closes with a discussion on the compilation flow.

Now that the groundwork has been laid, the core of the matter can be addressed. This is why the third chapter contains a description of the beta machine. In this description, the reader is made aware of the dimensioning of the machine: the word size, the endianness of the machine, its instruction set, etc. Following this, the implementation of the CPU on FPGA is presented in detail. The IO unit which provides access to several IOs of the DE10-nano board to the CPU is also described in this chapter.

At this stage, the specifications are fulfilled for the hardware part of this master thesis. That said, it would have been a shame to deliver this machine without any interface other than what is possible using the IO unit. This is why the following chapter details the implementation of a graphics accelerator which will be abusively called GPU (Graphics Processing Unit). The HDMI controller and the protocols allowing its use are introduced beforehand. The clock unit (CLKU) is also presented.

Chapter 5 covers the design of the Memory Access Unit (MAU) which gives access to the memories of the beta machine from the ARM processor. The description of the Control Unit (CTRLU) allowing the startup and shutdown of the machine from the ARM processor is given here. The communication protocol common to both units is first presented before detailing the implementations.

In order to provide easy access to the machine, an operating system must be installed on the ARM part of the FPGA. Chapter 6 discusses the choices made at this level.

Finally, a chapter is dedicated to the different tools developed to facilitate the access and the programming of the machine by the users. This last chapter also lists different demonstrations programmed in assembly language that serve as proof of concept. The assembly libraries written for this work are also briefly presented.

Chapter 1

Overview of the hardware

1.1 Features of the DE10-Nano development board

Numerous devices are made available to the user on the DE10-Nano development board. Some of them are very interesting for this work. The Cyclone V FPGA is one of them. It includes a configurable part (the programmable logic) and an ARM processor (A in Figure 1.1). The Cyclone V is basically the main component of the system as almost everything on the board is driven by it. It is also where all the hardware designed in the work resides. The 1GB DDR3 RAM (B) which is used by the ARM side of the FPGA. An ethernet port (C) which can be useful to connect to the ARM processor from any external environment. For instance, SSH can be used to connect to it if the installed system can handle it. Then there are the SD card reader and the SD card (D, on the other side of the board). The SD card consists in the mass storage of this board. The one used offers 8GB of memory. Another important component is the HDMI output (E) that can be used from the programmable logic side of the Cyclone V. Other less important devices, mainly IOs are available: General Purpose Input / Output pins (GPIOs), LEDs, buttons and switches. All of these devices are discussed at some point in this report. The ones that are not listed (such as the Inertial Measurement Unit, IMU) haven't been used.

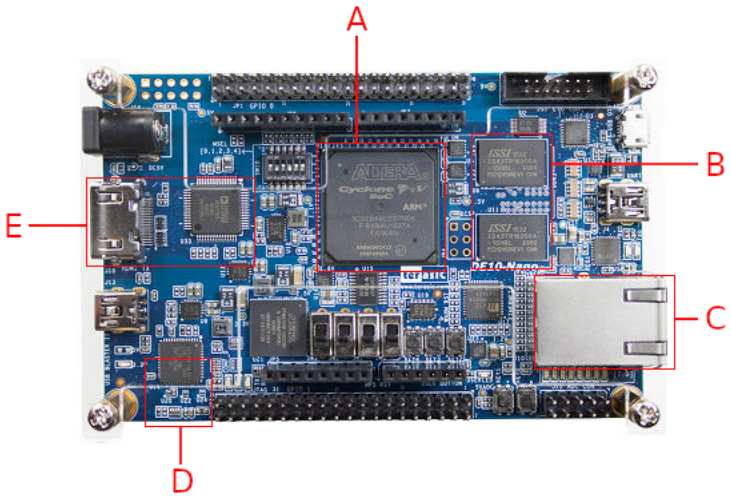

Figure 1.1: DE10 Nano development board main features.

1.2 Programmable logic devices

Field Programmable Gate Arrays (FPGAs) are a type of Programmable Logic Device (PLD) which are, as the name suggests, programmable logic circuits. Programmable means that the user can choose which circuitry is implemented in these devices. As seen later in this chapter, some of these devices are very versatile! They can implement almost any kind of digital circuit.

PLDs come in different flavors but follow a similar pattern for most types. In fact, they almost all are composed of an AND array and an OR array that are both programmable or not. Here, a discussion of the different types of PLDs and their evolution is made before describing FPGAs. The advantages and disadvantages of these technologies are also detailed.

1.2.1 ROM based PLDs

This first type is very simple, as shown in Figure 1.2. In fact, the circuit, which can only be combinatorial, is simulated by a ROM. Each address corresponds to an input of the circuit and each word in memory to an output of the circuit. The ROM therefore simply contains the truth table of the logic circuit to be implemented.

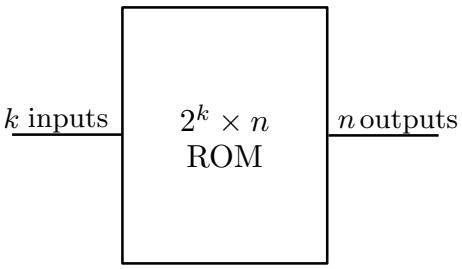

Figure 1.2: An external view of ROM based PLDs.

The inside of a $2^2 \times 4$ ROM can be seen as a 2-to-4 decoder followed with a 4-by-4 OR gates array, as shown in Figure 1.3, where the connections in the array are configurable. They can be opened or closed while programming the device. The decoder can be interpreted as the AND gates array here. Thus, the AND gates array is fixed and the OR gates array is programmable for this kind of devices.

In the example of Figure 1.3, the array is represented by a grid. All horizontal lines represent a 1-bit signal that can be connected to an OR gate. The signal is connected if a red x is present at the connection point. In the figure, each OR gate can thus be connected to at most 4 decoder bits. As can be seen, the OR gates array is here programmed to describe the truth tables of these boolean equations

$$\begin{cases} A_0 &= I_0 \cdot \bar{I_1} \\ A_1 &= \bar{I_1} \\ A_2 &= 0 \\ A_3 &= \bar{I_0} \cdot I_1 \end{cases}$$

Obviously, real devices contain a lot more signals and gates to be cost effective and useful. As stated previously, the ROM PLDs can only represent combinatorial circuits. There is no register or feedback loop in the circuits that are described with this technology.

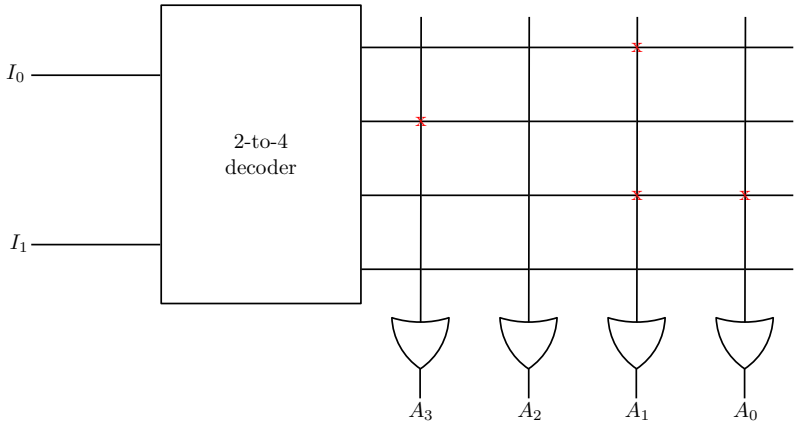

Figure 1.3: An internal view of ROM based PLDs.

1.2.2 PAL and PLA

The Programmable Array Logic (PAL) and Programmable Logic Array (PLA) devices differ from the ROM based PLDs since they both have a programmable AND gates array. However, only the PLAs have both AND and OR gates arrays that are programmable, the OR array is not programmable for PALs. The PLAs are thus the most flexible devices of the three. As expected, this flexibility makes them also more complex and complicated to program. The outputs of these devices can often be complemented through programming too. It should also be noticed that both the inputs and the inverted inputs are directly available in the arrays here, as illustrated on Figure 1.4.

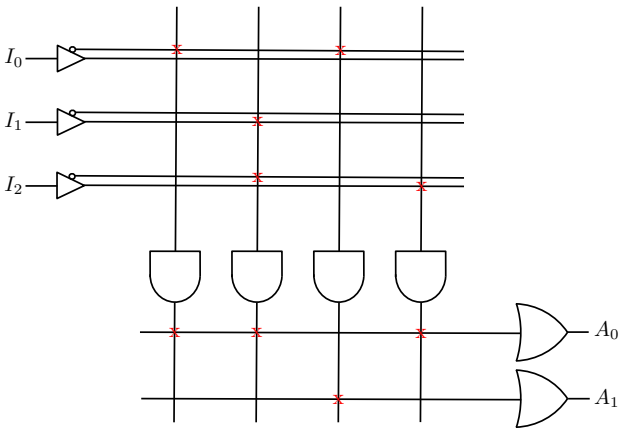

Figure 1.4: An internal view of PLAs.

1.2.3 Programming methods

This section contains a brief discussion of the various usual programming techniques.

Initially, these devices were mainly programmable in two ways. The first was to use fuses and anti-fuses at each interconnection. The user then had to apply destructive currents to the right places for the fuses to open the connections and the anti-fuses to close them. The second method was to leave it to the user to complete the IC design. One would then make a mask to select which interconnects to open and close during a lithography process. To provide some examples, the fuse methods are used in PROMs and masks for ROM based PLDs. Both of these programming methods make the devices non-erasable, which at the same time makes them non-reprogrammable. However, they are both non-volatile which can be an advantage when the user don't want any boot latency at startup, as no programming phase is needed at boot time.

Later, other programming techniques were developed following the emergence of transistors. Indeed, these transistors are placed at the interconnections and controlled by their gate with the help of a bit kept in a memory. The memory can then be programmed to define the behaviour of the connections without them being frozen forever. As the memory is usually SRAM, the devices become reprogrammable but still volatile. Other technologies have subsequently made it possible to have non-volatile memory in addition to memory by using, for example, floating gate transistors. These two types of interconnections are usually configured by applying a high current to set or reset the memory state.

1.3 The case of FPGAs

Now that PLDs and programming methods are introduced, it is time to get to the core of the matter with a description of FPGAs. This section aims to be general enough to cover the operation of a large number of FPGAs. More details specific to the FPGA actually used in this work are given in the next section.

FPGAs have three main programmable levels. These three levels are discussed in turn, starting with the programmable logic blocks, then the programmable interconnect and finally the programmable I/Os.

1.3.1 Programmable logic blocks

Programmable blocks are present in very large numbers in modern FPGAs (usually tens to hundreds of thousands). They are the basic elements of the circuits implemented on FPGAs. Due to their architecture, which can be seen in Figure 1.5, they allow both combinatorial and sequential circuits to be implemented.

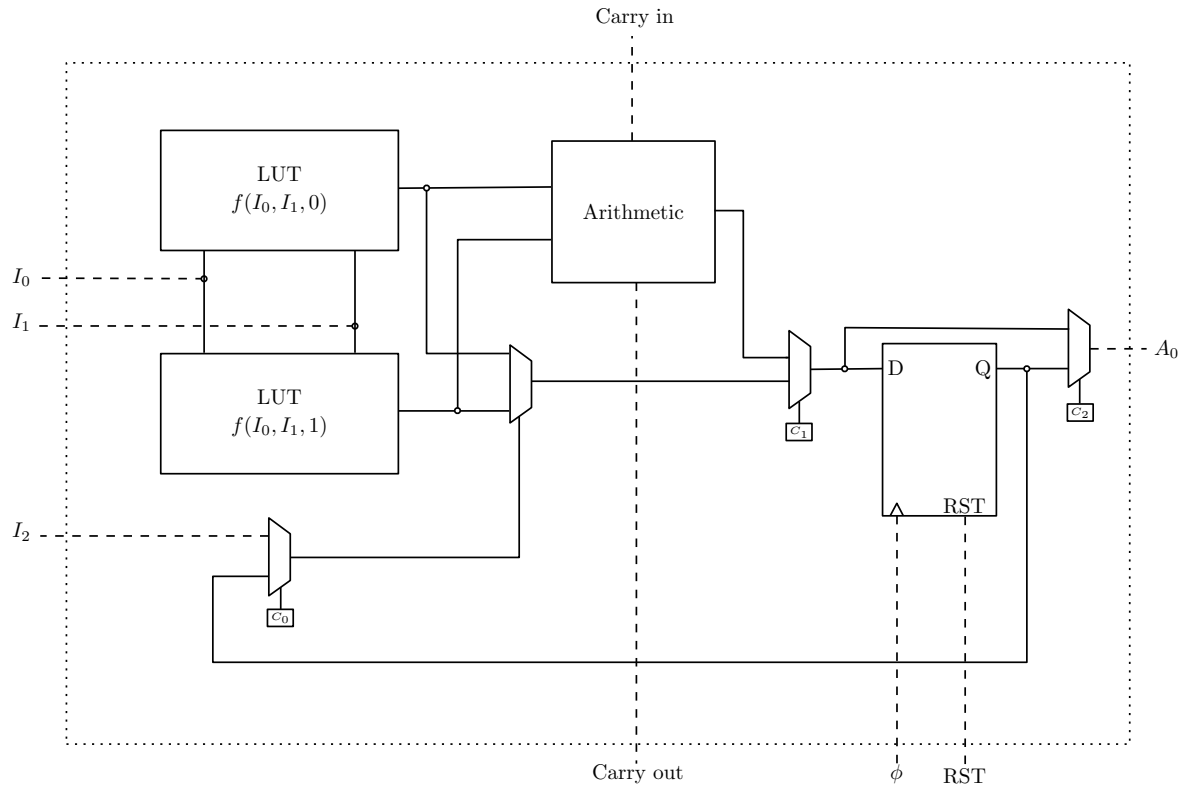

Figure 1.5: An FPGA programmable logic block.

The combinatorial part of the circuits is implemented using a LookUp Table (LUT). The truth table is simply stored in this LUT as for the ROM PLDs but the bits are now stored in SRAM. If one LUT is not sufficient, the logic function can be implemented in several LUTs connected at the output by a multiplexer that is controlled by the remaining inputs. This is made fully valid by Shannon's expansion theorem.

Theorem 1 *Any boolean function* $f(x_1, x_2, ..., x_n)$ *can be expressed as*

$$f(x_1, x_2, ..., x_n) = x_n \cdot f(x_1, x_2, ..., x_{n-1}, 1) + \bar{x_n} \cdot f(x_1, x_2, ..., x_{n-1}, 0)$$

As shown in Figure 1.5, the blocks also have an arithmetic sub-block and a register. Having this arithmetic block in each single programmable block greatly increases the efficiency of the FPGAs and reduces the latency of the blocks, even if most of the time these circuits are simple carry adders. Fixed hardware is always faster than programmable one as they are better optimized. In addition to the programmable LUTs, other configuration bits are available to control the behaviour of the multiplexers: C_1, C_2 and C_3 allowing respectively to add a feedback loop to the block, to select the result of the boolean function or that of the arithmetic sub-block and to make the output of the block registered or not.

In brief, these blocks are the heart of the FPGA. All their features and configuration bits allow them to describe almost any kind of digital circuit in an efficient way.

1.3.2 Non-programmable blocks

Other non-programmable or less programmable blocks are also available within FPGAs. They are divided into two main groups: memories and utilities. Memories are simply blocks that can store data. A specific discussion of these memory blocks is made in a later section. On the utility side, there can be a wide variety of blocks: multipliers, PLLs, DSPs and even whole CPUs. In short, enough to make FPGAs even more powerful!

1.3.3 Programmable interconnect

Programmable blocks are very versatile and efficient, but they need to be connected to each other to make large circuits. As can be seen in Figure 1.6, the blocks are positioned on a grid and the interconnection passes around them.

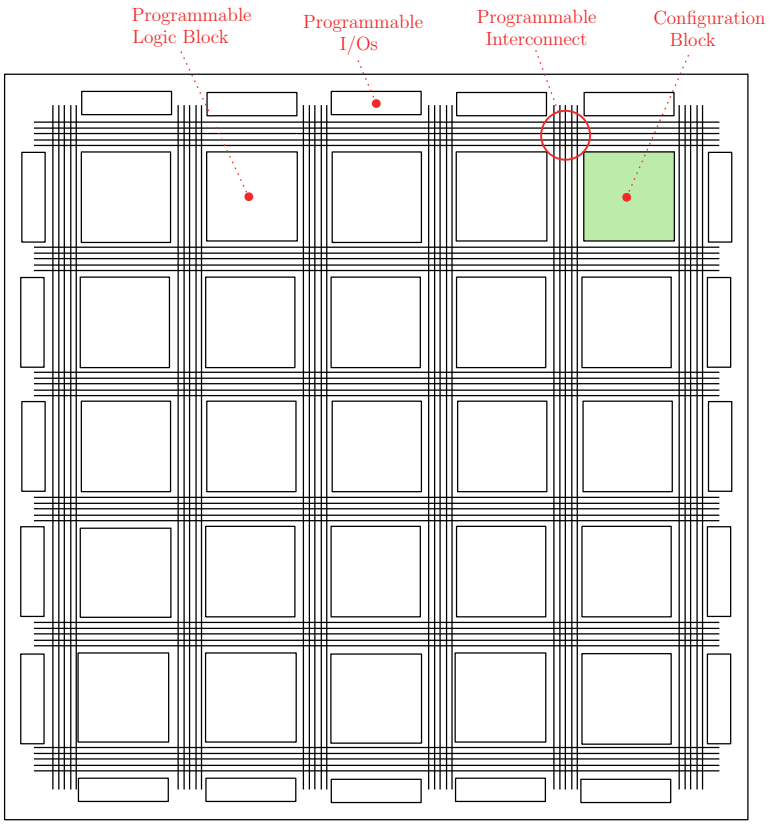

Figure 1.6: High level view of an FPGA.

For this purpose, there is the programmable interconnection that allows the signals to be routed from one block to another. To do this, configurable switches are placed at the crossings of the tracks. However, everything must be connected, whether the blocks are close or distant. As this cannot be done directly without causing problems in terms of time and power constraints, several hierarchical connection levels are present. However, in order to keep the explanations simple, it is considered here that there is only one level. It should also be noted that each switch adds latency to the signal, so the interconnection is optimised to limit the number of switches between blocks.

In addition to this configurable interconnection, there are also tracks to bring global signals everywhere. Among these signals are the clock, the resets, ... On a more local level, some signals are also shared between the blocks. These include, for example, the carry signals of the arithmetic sub-blocks.

1.3.4 Programmable I/Os

Now that almost the entire interior of the FPGA has been described, all that remains is to connect it to the outside. This is why there are these I/O pins all around the FPGA in Figure 1.6. These as well as the other parts of the FPGA are also configurable. As it is desirable for the FPGA to adapt to a wide variety of external hardware, these pins can be configured to the standard used. They can also be used as input, output, bi-directional, differential pairs, ... In short, they allow a large number of configurations so that the designed circuit can be adapted and coupled with a maximum of other circuits.

1.3.5 Programming the FPGA

Due to their complexity, the configuration circuitry of the FPGA is just as complex. For this reason, few details are provided in this section. That said, it should be noted that part of the FPGA is assigned to this function. This block takes care of configuring all the configuration bits in the relevant memories. However, since the memories holding the configuration within the FPGA are volatile, it is often necessary to couple a non-volatile memory to the FPGA to retain its configuration. The configuration therefore has to be re-configured each time the FPGA is rebooted, which can create some latency. It is also important to note that the memory holding the configuration must be relatively large, as the number of configuration bits is very large. More details on compiling designs and configuring FPGAs in practice are given in a later section.

This closes the theoretical discussion on FPGAs. As already said, FPGAs are extremely powerful and versatile. This enables them to describe a large number of circuits and to adapt to many existing circuits. Another advantage is that they are massively parallel due to their structure. Despite all these advantages, FPGAs still have some drawbacks. One of them is cost. Some FPGAs can cost several thousand euros. The complexity and difficulty of access is another problem for designers, the datasheets are numerous and extremely long, the tools are also numerous and not always easy to use. Electrical power requirements and start-up latency are also two other factors that must be taken into account when a user chooses an FPGA for circuit development. Nothing is perfect, not even FPGAs.

1.4 Inside Cyclone V

1.4.1 Programmable logic side

In the Cyclone V, things are even a little more evolved than previously explained but it is still quite accessible. In fact, the programmable logic blocks is replaced here by Logic Array Blocks (LAB) which are themselves made up of ten Adaptive Logic Modules (ALMs) which are the equivalent

of the programmable logic blocks studied previously. A high level view of the LAB units is given in Figure 1.7. In the LAB, the different ALMs, the inputs and outputs of the LAB, can be interconnected at the user's convenience.

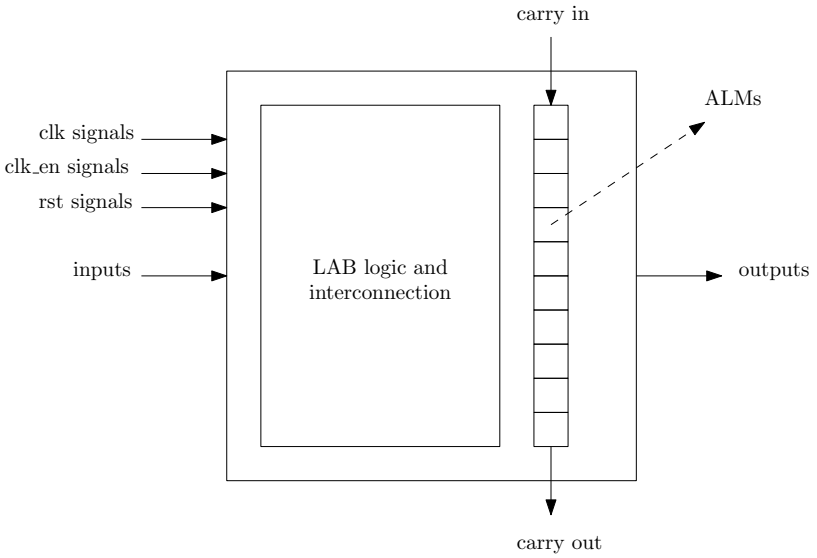

Figure 1.7: High level view of a LAB unit.

There is also another type of LAB, the Memory LAB (MLAB) which allows to describe SRAM. They can store up to 640 bits and represent 25% of all the LABs present on the Cyclone V. These MLABs do not have memory instead of the ten ALMs, they also have the ten ALMs and all the logic necessary to use them. They are a superset of LAB somehow.

These LABs (and MLABs) are placed in the FPGA fabric in an array as explained in the previous section. However, it should be noted that the array is not symmetrical with respect to the vertical and horizontal connections at the local level (this part was not detailed previously). Indeed, the horizontal connections aim to connect the blocks close to each other by ensuring a high speed of operation. These are the inputs and outputs that are mainly linked to it. The columns, although they are also fast, are mainly used for carry signals. At the local level, a LAB can have control over 30 ALMs: its own ten, and the ten of its two horizontal neighbors. This allows to build very complex logic in a restricted space. Of course, a more global interconnection allows all LABs to be connected together. There are many more details about the internal connections of the LABs that could be discussed, but these are not explained in this report.

Concerning the ALMs, as just said, they are very similar to the Logic Array Blocks of the previous section. The main differences are that they accept more inputs, up to eight. The internal arithmetic block is in fact two separate adders, and four registers are provided in each ALM. Depending on their configuration (set during the programming phase), ALMs are adapted to specialize in a functionality to make the circuits more efficient (for example more entries, an optimization for arithmetic, shared arithmetic with an input of the adder that is fed by an input of the ALM, ...).

Other blocks

The LAB blocks are very customizable and extremely numerous on the Cyclone 5. However, there are several other types of blocks that are worth listing and briefly explaining. The overall structure of the Cyclone V FPGA is shown in Figure 1.8.

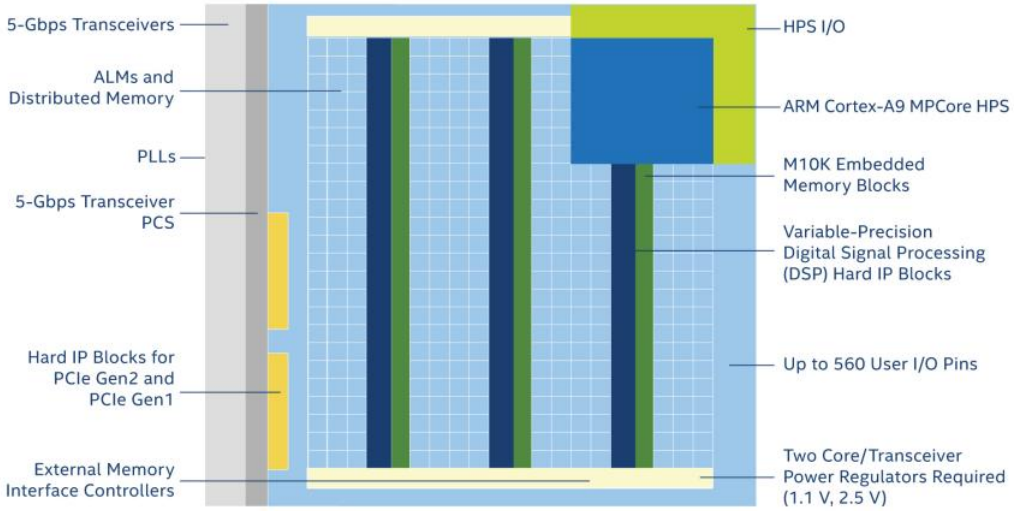

Figure 1.8: Overall structure of the Cyclone V FPGA [1].

The largest block visible on the diagram is the one at the top right which represents the entire ARM system, this block is discussed in the next section. Then one can see many dark green columns that designate the M10K memories. These memories are the main source of on-chip storage. A sub-section of this chapter is dedicated to the memories, and they are studied in more detail there. The PLLs are also visible in the light gray band on the left. PLLs allow to generate clocks at a given frequency from a reference clock. Again, this unit is studied later. Another element present in large quantities is represented by the dark blue columns. It is the Digital Signal Processing (DSP) Hard IP Blocks. These blocks have a multitude of functions. Indeed, they provide arithmetic units of addition, subtraction and multiplication, allow the implementation of filters, ... In this work, they are only used to perform operations such as addition, subtraction and multiplication in the Arithmetic Logic Unit (ALU). It should be noted that using these DSPs is much more advantageous than using ALMs. Indeed, the arithmetic circuits of the DSPs are "hard-made" which means that they have no programmable interconnection logic in their internal circuits, so the delays are much smaller and their architecture is totally optimized. For the user, this allows him to use higher clock frequencies than he would have been able with circuits designed with ALMs. Other units are available but are not discussed.

1.4.2 ARM processor side

Inside the FPGA, a 32bits ARM processor, often named Hard Processor System (HPS), is integrated. It is a Cortex A9. No bare-metal programming is done on this processor. It is used as an interface to access the beta machine. Indeed, an operating system has been installed on it and utilities have been developed so that the use of the architecture developed in this work is easily accessible. Discussions about the operating system and the tools are done at the end of this

document. For the moment no description is made. That said, it was important to mention it in this part of the report as it constitutes a large part of the Cyclone V hardware. Furthermore, it was necessary to introduce it before starting the next two sections.

1.4.3 Cyclone V interconnect

As just mentioned in the previous section, the ARM processor serves as the user interface. Therefore, one needs a way to communicate between the ARM processor and the programmable logic of the FPGA (which will now be refered to as FPGA although the ARM processor is also part of the FPGA).

In fact, there are bridges between the two sides. These bridges are the link between the two parts and can be used for communication. Each of these bridges has a specific purpose, it is not just the same one put several times in parallel. The first and most basic bridge is the Lightweight HPS-to-FPGA bridge. This one is designed for small transactions that are ponctual, the user should avoid for example streaming data on this channel (even if nothing prevents it). The limitation in size comes from the fact that the data bus of this bridge is only 32 bits wide. As the name of the bridge indicates, it is unidirectional and goes from the HPS to the FPGA. Then come two more interesting bridges: the FPGA-to-HPS Bridge and the HPS-to-FPGA Bridge. The directions of these two buses are obvious. What is less obvious is their size. On the HPS side, their dimension is fixed at 64 bits, but on the FPGA side, they have the great advantage of being able to be configured to 32, 64 or 128 bits. These bridges are ideal for large data transfers. Finally, there are the FPGA-to-SDRAM bridges which allow six masters on the FPGA side to have a non cached and non coherent access to the RAM memory. This is discussed in the next section in more detail. These different bridges all allow several masters on one side to be connected to multiple slaves on the other. The selection of the slaves is done by byte aligned addresses. The address space is therefore the only thing limiting the number of slaves on a bridge. In order to better visualize the interconnection, a diagram is provided in Figure 1.9. In this work, only the HPS-to-FPGA bridges are used.

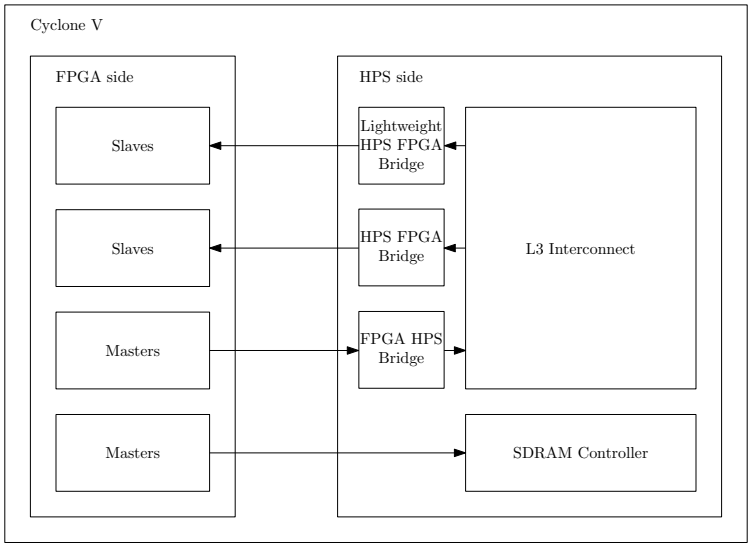

Figure 1.9: Interconnect bridges.

In Figure 1.10 the address mappings from different viewpoints are shown. For the two buses going from the HPS to the FPGA, it is the MPU column (Memory Protection Unit, the unit that protects memory accesses on ARM architectures) that is of interest. One can see that the address space is much larger for the HPS-FPGA bridge than for its lightweight equivalent. These addresses are those to which the slaves on the FPGA side are associated and therefore those to which one has to send the transactions from the ARM processor in order to communicate with these slaves. The protocol is of course studied in this document but this is done when the transactions between the two parts are discussed.

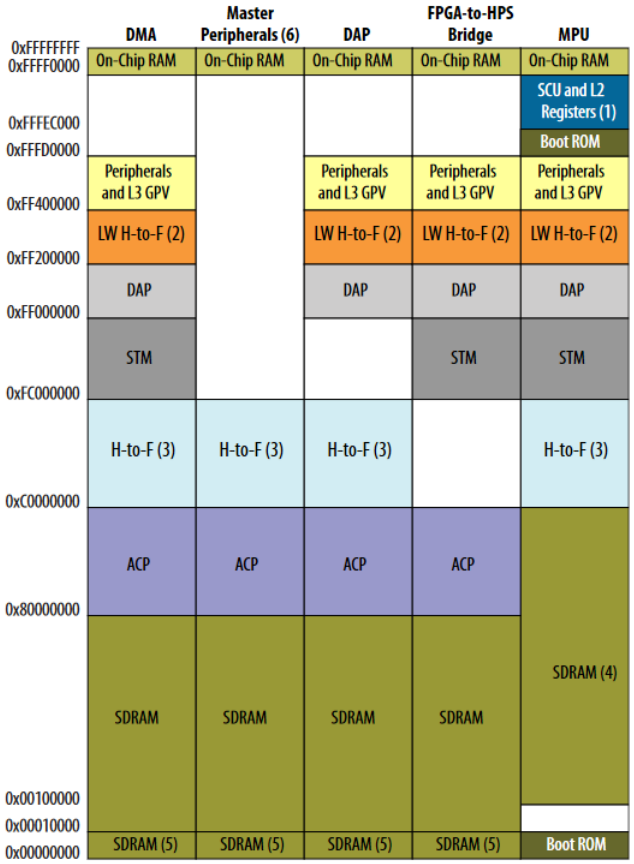

Figure 1.10: Address maps for system interconnect masters [2].

1.5 On-board and on-chip memories comparison

As seen in the previous sections, several types of memories are available. Directly on the board there is the SD card storage and DDR3 RAM memory. On the Cyclone V there are also two types of memory which are MLAB units and M10K blocks. Each memory has its own features. Since both the beta machine and the ARM system require memory, it is interesting to consider the use of these different memories, compare them and finally make a choice.

Let's start with the simplest, the ARM processor. In fact, on this side no choice is left to the user. The DDR3 memory serves as the main memory and the SD card as the secondary memory for non-volatile storage.

Concerning the Beta machine, the choice is less obvious. Which memory to choose to store instructions and data, as main memory? Given the nature of the SD card memory, it can immediately be ruled out of the possible choices. What about the other memories?

1.5.1 DDR3 memory

The DDR3 memory offers 1GB shareable between the ARM processor and the FPGA, which is really huge. However, an almost direct access to this memory is only possible if the access is done in a non-caching and non-coherent way with respect to the ARM processor caches. Another possibility is to use the ACP ID mapper which ensures a cached and coherent access (using the L2 cache). The two access paths are identified in Figure 1.11. The problem with these two accesses is that they are not instantaneous, the writes and reads do not take the same number of cycles and the number of cycles is not unitary. To use these memories, it is therefore necessary to either slow down the whole system on the FPGA or to have a cache memory on the FPGA with faster memory next to it. In addition to this, in the first case (direct access) it is necessary to configure the operating system on the ARM processor to ensure that it never accesses the part of memory reserved for the FPGA system and in the second case (with the ACP) it is necessary to configure the ACP correctly so that its operation is as expected. In both cases this means modifying the kernel of the operating system, which is not necessarily simple. For these two reasons: the delays and the necessary modifications at the kernel level, this memory has not been chosen. However, it could be a very interesting modification of the system built in this work to provide more memory to it.

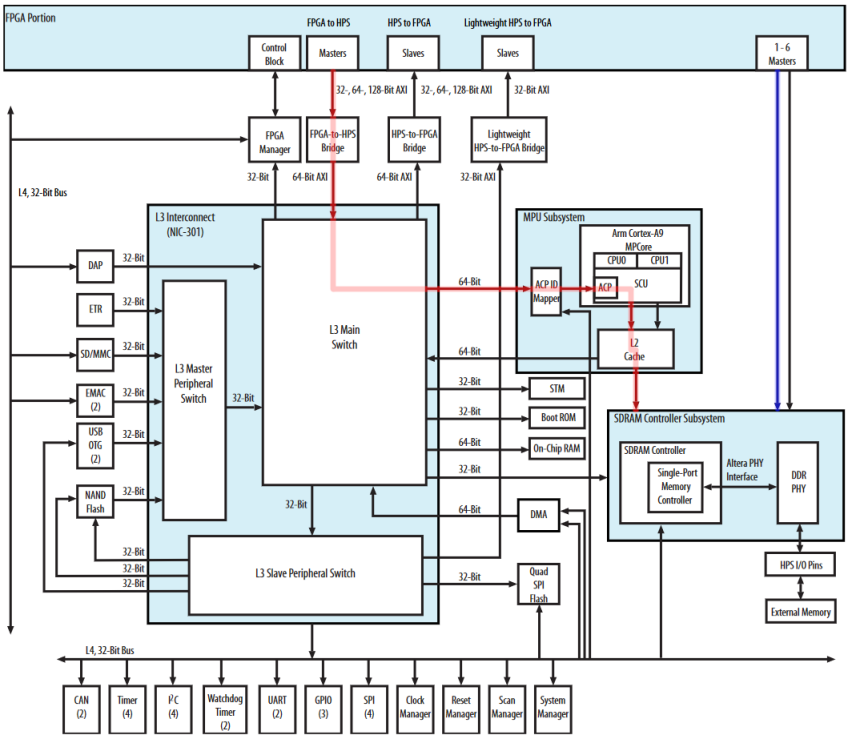

Figure 1.11: DDR3 accesses from the FPGA. Red path uses the ACP and blue path is a direct access to the memory controler [2].

1.5.2 M10K and MLAB memories

What about the M10K and MLAB memories? These two memories are directly present on the FPGA, which means that they can be read and written in one clock cycle. Moreover, they are not used by the ARM processor, which avoids having to take care of the coherence of the caches. How to choose between these two memories? The first difference is the amount of memory available. Indeed, there are 553 M10K memory blocks in the Cyclone V SE A6 used in this work. As a M10K memory contains 10Kb, the whole of these blocks represents a little more than 5Mb which is already very large for a machine like the beta machine. There are also 994 MLABs blocks containing 640 bits each, as seen previously. This makes a total of a little more than 600kb. However, there are three reasons why the choice is made for the M10K rather than the MLABs. The first reason is that MLABs use up programmable logic; the more MLAB memory is used the less custom logic can be added to the FPGA, up to a loss of 25% if all memory is used. The second is that the datasheet advises to use these memories in cases where shallow and large memories are needed. This is because it is complicated to create large memories locally with MLABs. Last but not least, MLABs do not provide dual-port access, which is a huge handicap because it is necessary to duplicate the memories if the user wants this kind of access. As the choice of M10Ks seems to be the most suitable for this work, it is decided to use this source of memory. To keep things simple, hybrid systems are avoided. Another difference between MLABs and M10Ks is that MLABs have no problem running at frequencies up to 420 MHz while M10Ks do not go beyond 315 MHz. However, the system of this work does not exceed a 50 MHz clock frequency, so frequencies are of little concern here.

Chapter 2

Hardware design flow on FPGA

Now that the reader is aware of the internal structure of FPGAs, the design flow on these FPGAs can be discussed. Indeed, a series of more or less compulsory steps are required to carry out the design of an embedded system on FPGA. This chapter is dedicated to the description of this design flow and the tools used to carry it out. All the tools that are discussed in this chapter are part of a super-tool named Quartus. Quartus is the main software used for the development on Intel FPGA. In fact, it is a kind of IDE for hardware development. It interfaces many tools together to ease the work of the users.

2.1 Design flow

The very first step is of course to choose an FPGA, once this is done, one must start by assigning the inputs and outputs of the FPGA. This step is named I/O Planing in Figure 2.1.

2.1.1 I/O Planning

During this I/O Planing step, the user has to choose which input or output of his design goes to which physical input or output of the FPGA. It is also during this step that the types of I/Os, the standards they should use etc. are set. In the context of this work, this step could be skipped. Indeed, the manufacturer of the DE10-Nano board provides with its board a Quartus project containing the definition of the I/Os in order to correspond to what is present on the board. This greatly simplifies the task and avoids digging through all the datasheets of the different components of the board to find out how to set up all the IOs.

2.1.2 Design Entry

In this step, the user defines the different logical entities that are needed to build up the desired circuit. A logical entity or a module is simply a sub-circuit with a given interface, an example is shown later in this section.

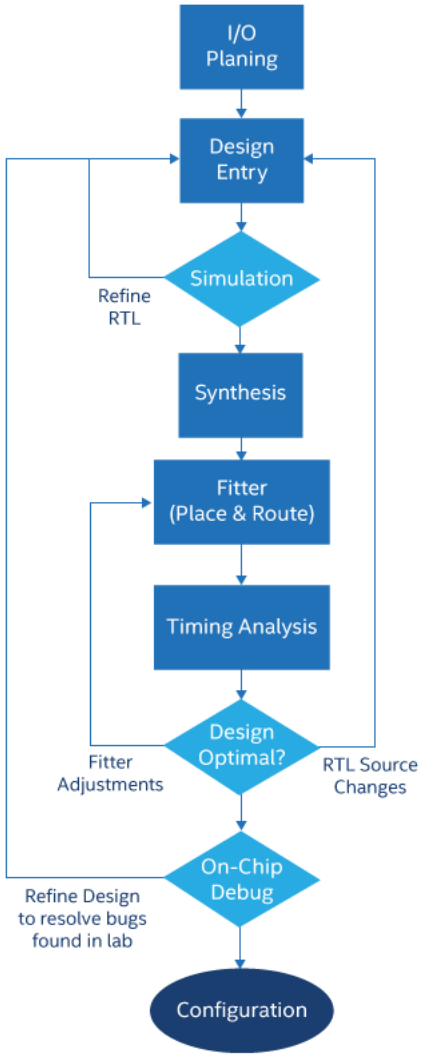

Figure 2.1: Intel FPGA design flow on Quartus [3].

All this is done in a so-called hardware description language. There is no need for special tools here, a simple text editor will do. However, Quartus contains an editor that can be used for this purpose. Concerning the languages, there is a multitude of choices. But there are three frequently used languages. These are VHDL, Verilog and System Verilog. VHDL is a high level language that is highly typed. Verilog and System Verilog are weakly typed and lower level. For example, in VHDL it is frequent to work directly on data-structures, complex types that can quickly hide the logic of an architecture. In Verilog and System Verilog, few types are provided and the hardware architect does regularly work with registers and words in which the bits are assignable one by one. Since System Verilog is a newer language and not always (correctly) supported by the tools, it was decided to use Verilog for this work.

It should be noted that these languages, although similar to software programming languages, do not work in the same way. Indeed, in a Verilog file for example, one defines so-called modules. A module has inputs and outputs. Each input and output can either be defined as a wire, which means that this I/O have no specific functionality, it just serves as a connection, or as a reg, which means that this I/O is registered. A register is therefore placed at this I/O.

Inside the module, the developer then defines how these inputs and outputs are linked. Here again, two possibilities are available to the user. Indeed, it is possible to define circuits that are either combinatorial or sequential. For the combinatorial circuits, they can take any input (registered or not) and transmit its outputs to non-registered outputs. It is important that the output is non-registered because combinatorial circuits are asynchronous, the compiler would not understand who is driving the register if it was put directly at the output of a combinatorial circuit. The other possibility is to describe sequential circuits. For these, a sensitivity list must be chosen. That is to say, signals that have the effect of updating the circuit. In general, only the clock is used, and the state of the circuit changes at each clock cycle. Note that any other signal specified in that list might result in an asynchronous response of the circuit, as the circuit responses to a clock change and also to a change of this signal. This is useful to implement asynchronous resets in synchronous circuits. Inputs of sequential circuits can be registered or not but sequential circuits can only transmit their output to registers. It should be noted that a combinatorial circuit and a sequential circuit can be cascaded since sequential circuits accept non-registered inputs. This makes it possible to put a combinatorial output on an output register for example, even though it is impossible with the combinatorial circuit alone.

At the level of sequential circuits, another subtlety compared to software programming languages should be noted. The assignment of registers is done in a blocking way. This means that these assignments are only effective at the next clock cycle (or next sensitivity signal update) and that all these assignments are made at the same time in parallel. One should not forget that the language describes circuits. And so all outputs appear at the same time once the rising or falling edge of the clock is perceived. For example assigning x to a and then a to b results in $a[t]$ containing $x[t-1]$ and $b[t]$ contains $a[t-1]$, which might not be obvious for a software developer.

To better illustrate the above notions, a small example of a module described in Verilog is given below. It simply describes an AND logic gate whose output is registered. The corresponding circuit is shown in Figure 2.2.

Verilog Registered AND gate example.

```
module registered_and(
    clk,
    input0,
    input1,
    output0
);

// clk is defined as an input wire
input wire clk;
// input0 is defined as an input wire
input wire input0;
// input1 is defined as an input wire
input wire input1;
// output0 is defined as an output and it is registered
output reg output0;


// An internal wire is defined
wire and_output;
```

```
// Describes an and gate between input0 and input1
// and put the result on the wire and_output asynchronously
assign and_output = input0 & input1;

// Sequential circuit that updates at rising edge of clk
always @(posedge(clk)) begin
    // The circuit simply directs the value present on "and_output" at
    // that moment to the registered output (i.e., it sets the register).
    output0 <= and_output;
end

endmodule
```

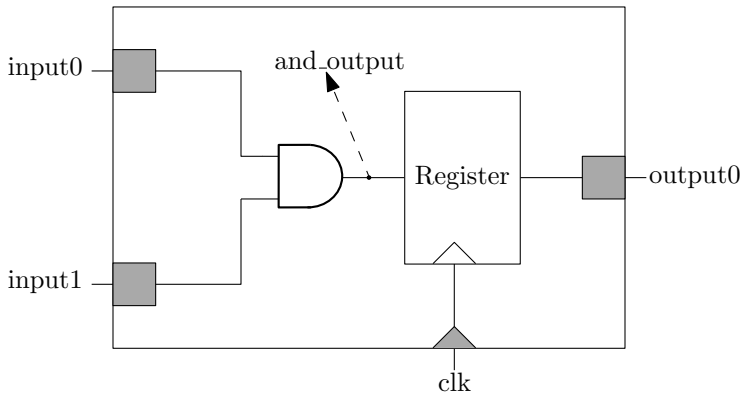

Figure 2.2: Intel FPGA design flow on Quartus.

Other tools than programming allow to define modules. A particularly useful tool is the IP creation Mega Wizard shown in Figure 2.3. This tool allows the user to create some common blocks such as on-chip RAMs using a graphical interface where one can choose the different options. For a RAM the user can choose the word size, the number of words, whether the memory has two ports or not, etc. At the end of the configuration, the wizard generates the Verilog file describing the module just defined.

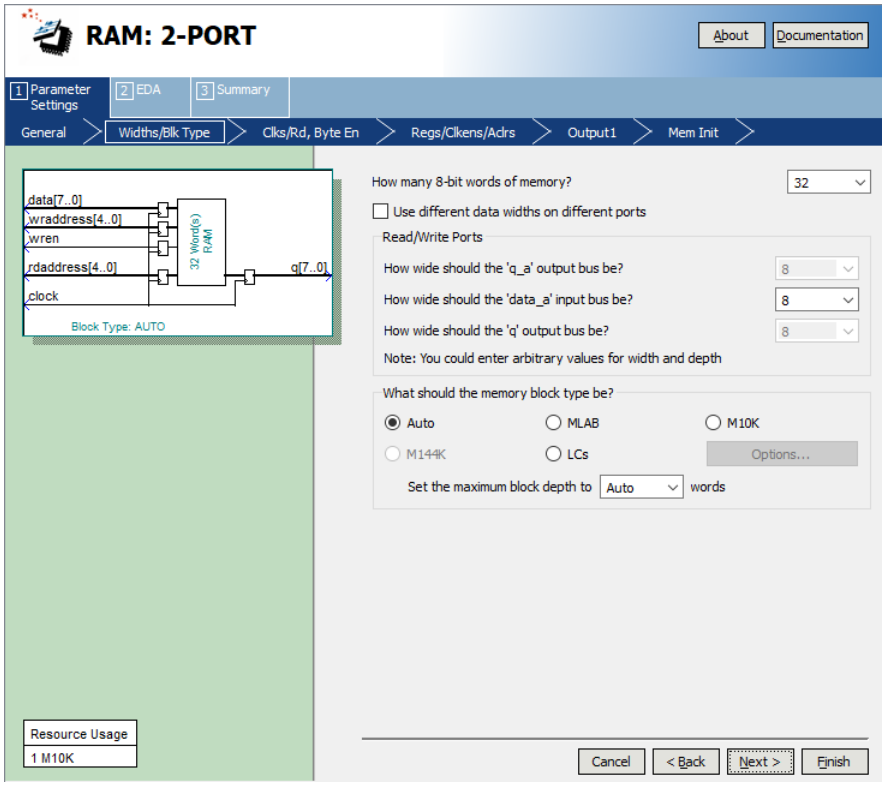

Figure 2.3: IP creation Mega Wizard window.

2.1.3 Interconnect Design Entry

This step is equivalent to the simple design entry, i.e., the goal is to describe the architecture. However, as the tool to design the interconnect is specific, a section is dedicated to it. As discussed previously the interconnect is the part of the architecture that links the FPGA and the ARM. The use of the buses is done through several signals, which can quickly become intricate to maintain directly through Verilog files. The specific tool for this, Qsys, allows to link the masters and slaves of the FPGA to those of the ARM processor in a graphic interface. In fact, one can add several slaves or masters in QSys and link them to the desired bridge. In this tool, the address space of each slave has to be specified. Qsys checks if the specified addresses are not outside the address space of the master to which they are connected. Figure 2.4 shows the interface to the HPS (ARM processor) and the connections to a slave.

In addition to the signals that are useful for communication on the interconnect, others allow interaction with the FPGA. This means that these signals are inputs or outputs of the Verilog code that is generated by QSys. They can thus be connected to other circuits on the FPGA side, contrary to the other signals shown in QSys. These signals are called exported signals and the user can chose which one is exported, usually the ones that are not direct links of the interconnect. Once a system is described, it only remains to ask Qsys to generate the system. It then creates all the necessary Verilog files that can be used as a single module later in user's Verilog code. This module receives as inputs and outputs all the signals that have been exported. To summarize, QSys is a kind of graphical programming software.

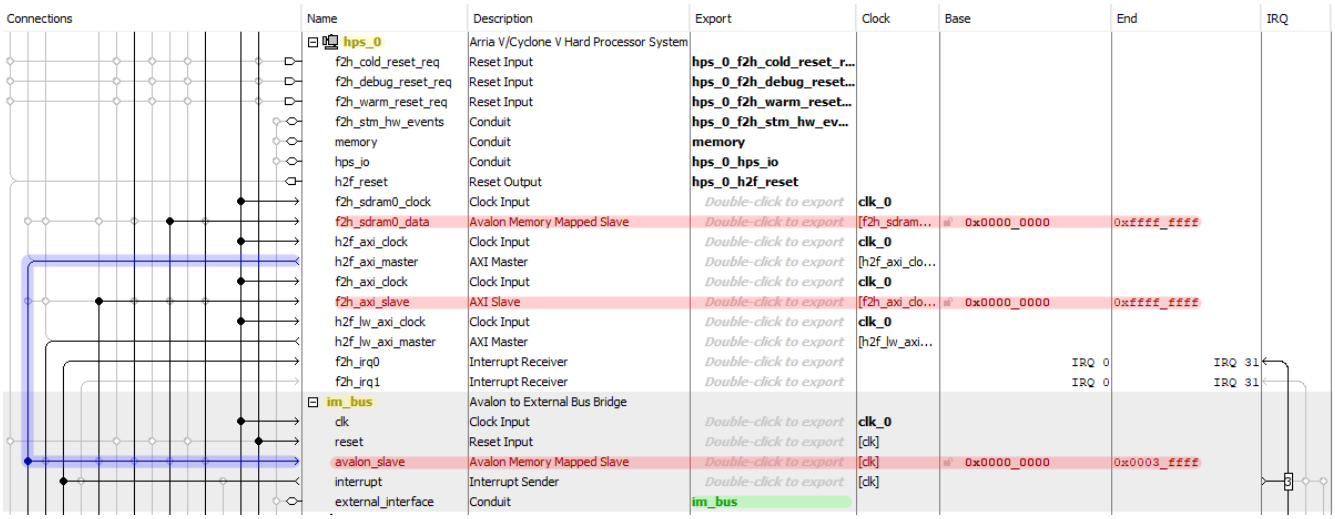

Figure 2.4: QSys window with an HPS named hps_0 and a slave module named mm_bus.

Figure 2.4 highlights different elements. The two entities present in the system are represented in yellow, there is thus the HPS named hps_0 which is an external representation of the ARM processor and an Avalon to External Bus Bridge called im_bus which makes it possible to make the link between a bus of the interconnection and a bus defined by the user on the FPGA. This module is discussed again later. In each module, some lines are in red, to denote the different slaves of the system. In the base and end columns, it can be seen that these slaves have a defined address space. They all start at 0 because they are relative to the starting address of the master. In the im_bus an exported signal is highlighted in green: it is accessible from the FPGA. In fact, most of the lines shown in QSys represent several signals describing entities, bus, ... Finally, a master-slave connection is shown in blue.

2.1.4 Simulation

Once the different modules of the system have been defined, it can be interesting to simulate some of them. This step is not necessarily mandatory but it can save a lot of time if the system is large. Indeed, compiling a large system can quickly take several hours, or days. It is thus beneficial to conduct verifications before compilation using simulation rather that later on by measurements on the working system. Sometimes, the simulation is also more practical because it allows to easily generate detailed scenarios. To simulating tool used by quartus is Modelsim. In this tool, signals can be fixed and tests can be performed. All signals are then displayed so that the user can verify the behavior.

2.1.5 Synthesis and Fitter

These two steps are the firsts of the compilation. The compiler transforms the descriptions made in Verilog into a logic circuit and then try to fit everything on the target FPGA. It checks that the resources are sufficient for the given design. The whole fitting process is time-constrained. This

means that it places the different blocks in such a way that the delays are minimal. After the Synthesis, the user can display the circuit to verify parts of its design visually. The tool used for this is the RTL net viewer. An example is given in Figure 2.5.

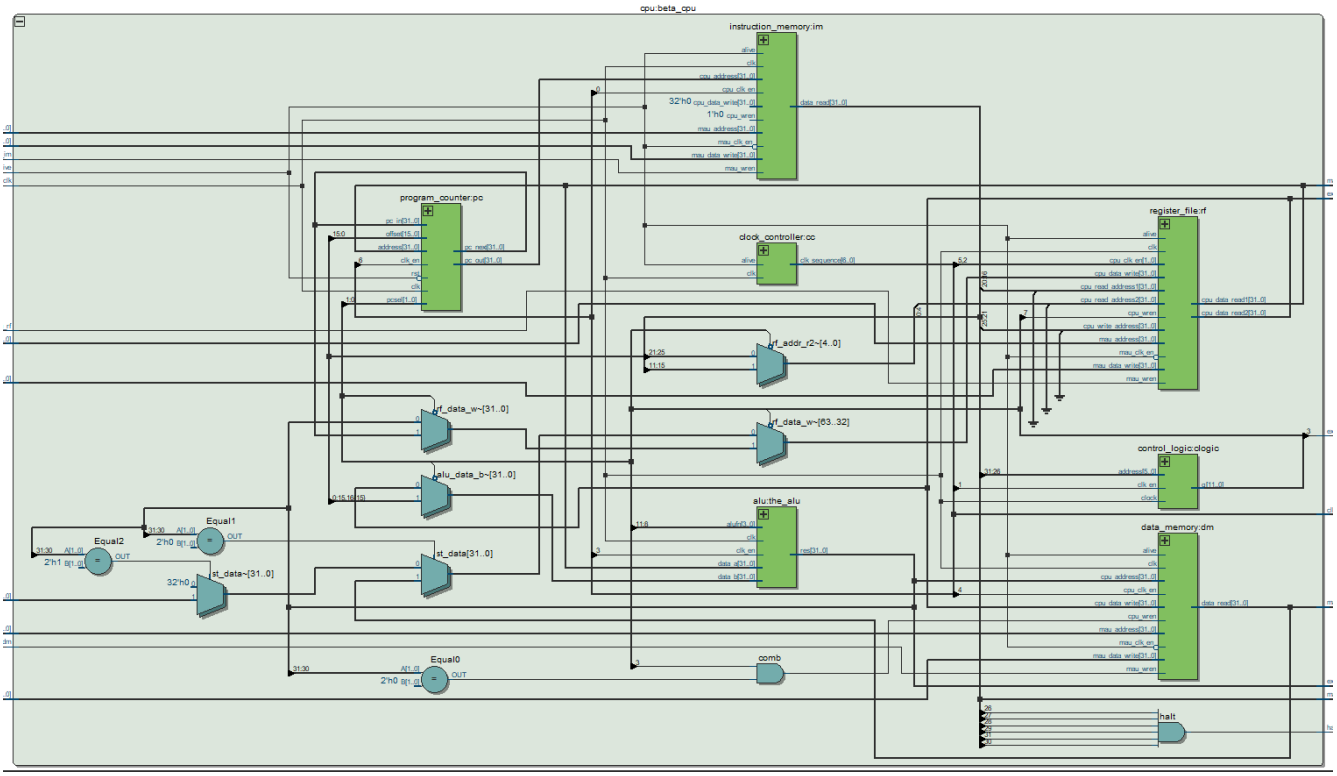

Figure 2.5: The beta machine CPU high level view in RTL viewer.

2.1.6 Timing Analysis and Design Optimal

FPGA designs are strongly dictated by time constraints. The purpose of this step is to check if the constraints are well respected by the design, after fitting. All this is done by simulation by testing the signal propagation delays at different temperatures. If one of the tests fails, the design is not validated and it has to be modified. For a timing to be validated, it is necessary for example that all the stages of a circuit have finished their work in the time existing between two rising edges of the clock. The software responsible for these verifications (TimeQuest) conducts many tests and measurements but these are not discussed here since this is beyond the scope of this work. What is important to remember though is that the user sets the constraints, e.g., by imposing the clock frequencies, and the software then takes care of checking them in the worst possible scenarios.

2.1.7 On-Chip Debug

The last step is to program the FPGA with the just compiled and timing-validated design. This is done using the Quartus Programmer, it is a straight forward step for the user, so there is not much to say about it here. Nevertheless, one still has to perform a last verification, the one in real environment. Most of the time, it is errors made during the Design Entry that are seen here. For

example, the user has connected components incorrectly, in a way that is not structurally wrong but is functionally wrong. Other errors that are not totally due to the user can also occur but are very specific (e.g. bugs due to operations on clock signals, ...).

To help the user in this debugging task, a very handy tool is provided in the Intel suite. This tool is the Signal Tap Logic Analyzer. Indeed, it allows to add a logic analyzer to the design which is connected to signals of the design that the user can choose. If the design and the analyzer are present on the FPGA, it is possible to display the signals in real time in the tool window when the board is connected to the computer. This tool also allows more advanced measurements such as triggered measurement which launches the recording of the data when a given trigger occurs (when a signal goes high for example). In short, it is the perfect tool for debugging.

Chapter 3

The Beta machine CPU and IO unit design

As explained in the introduction, the goal of this work is to design a Beta machine. This machine is in fact a minimal CPU based on the Harvard architecture, i.e., data and instructions are stored in two different memories. The machine uses words of 32 bits, stored in memory according to the little-endian convention, and only implements operations on 2's complement signed 32 bits integers.

In this section the different modules making up the Beta machine are studied one by one, from their functionality to their design. Simulation results are provided for some of them. Once the different elements are explained, the desired instruction set for the Beta machine is described. And finally, the different modules are connected to each other to form the Beta machine with the desired instruction set.

3.1 Beta machine components

Github link to the verilog files of the CPU.

3.1.1 Program counter

The program counter is a module with seven inputs (clk, clk_en, rst, pcsel, offset, data, pc_in) and two outputs (pc_out and pc_next), as shown in Figure 3.1. Its functions are to provide the current counter on pc_out (PC) and its next value ($PC + 4$) on pc_next, and to react to a control signal named pcsel. This control allows to choose the program counter's operating mode. The different modes and their functions are listed in Table 3.1. Note that if one connects pc_new to pc_in and that pcsel is set to 0b00, the program counter acts like a simple 32bits 4-by-4 counter.

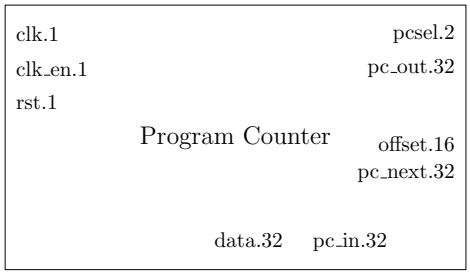

Figure 3.1: Program Counter.

Mode	pcsel	Details
NORMAL	0b00	The current value of the counter program pc_out becomes the value taken from the input pc_in.
BEQ	0b01	The current value of the counter program pc_out becomes the value taken on the input pc_in if the value on data is not 0. Otherwise, the current value becomes pc_in + 4 × offset.
JMP	0b10	The current value of the counter program pc_out becomes the value on data.
BNE	0b11	The current value of the counter program pc_out becomes the value taken on the input pc_in if the value on data is 0. Otherwise, the current value becomes pc_in + 4 × offset.

Table 3.1: Program counter modes.

Concerning the internal circuit (that is shown in Figure 3.2), the pc_out values is evaluated for the different modes and is put on a multiplexer which chooses the right output according to the pcsel value. For the BNE and BEQ modes, another multiplexer is used to select between the two possible results, the comparison $data == 0$ naturally controls these two multiplexers.

For the offset, it should be noted that it first undergoes a conversion from signed 16 bits to signed 32 bits before actually being used in the circuit.

The result selected by the multiplexer controlled by pcsel then goes to the register where it is stored at the next rising edge of the clock if the clk_enable is high. The pc_out thus has the desired value and the pc_next simply corresponds to pc_out + 4. When the reset (rst) is high, the register is synchronously reset to 0.

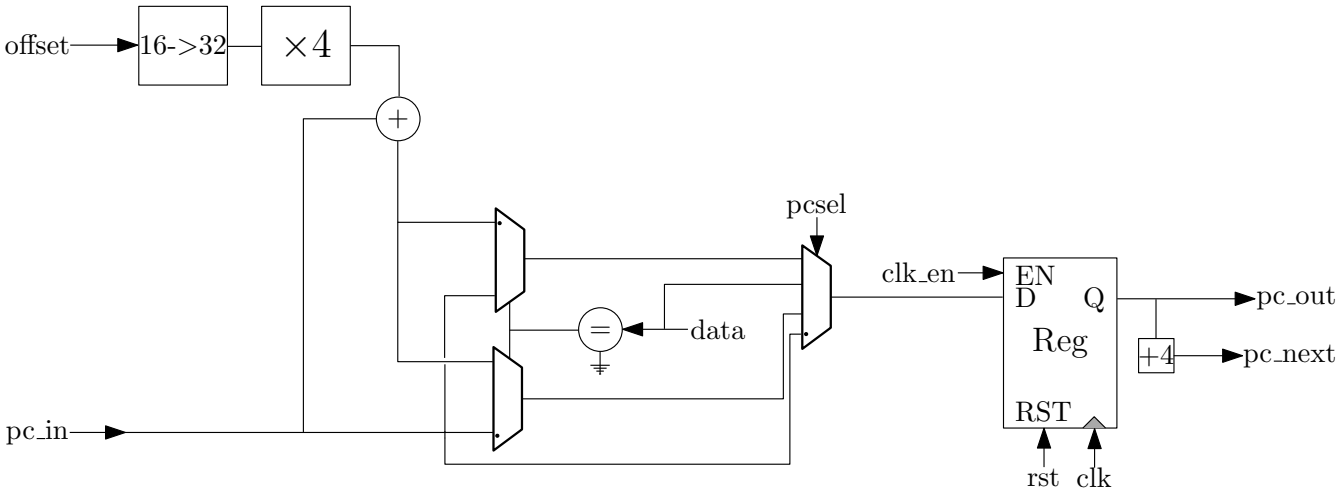

Figure 3.2: Program Counter internal circuit.

The Verilog implementation of the program counter is relatively straight-forward.

Program counter simulations

The different modes of the program counter have been verified by simulation. In the first mode (NORMAL mode, pcsel = 0b00), it was checked that with a pc_in at 4 294 967 292 (the maximum value with the last two bits at 0 that holds on 32 bits), the program counter returned 0 for pc_next and did not give a random value. Other random values for pc_in were checked to be sure that it worked correctly. Results of this first simulation are shown in Figure 3.3. Note that the input values are changed only once per two clock cycles but the module is totally able to manage input changes once per cycle.

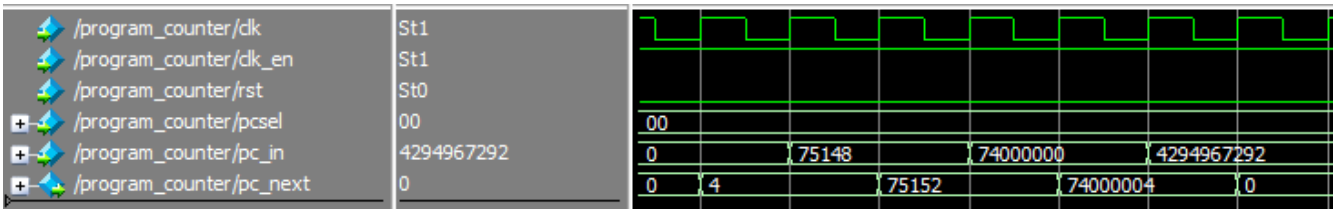

Figure 3.3: Program Counter in normal mode simulations for various pc_in values.

To test the JMP mode (pcsel = 0b10), it was simply tried to set data to several values to check if pc_out became this value. Results are provided in Figure 3.4.

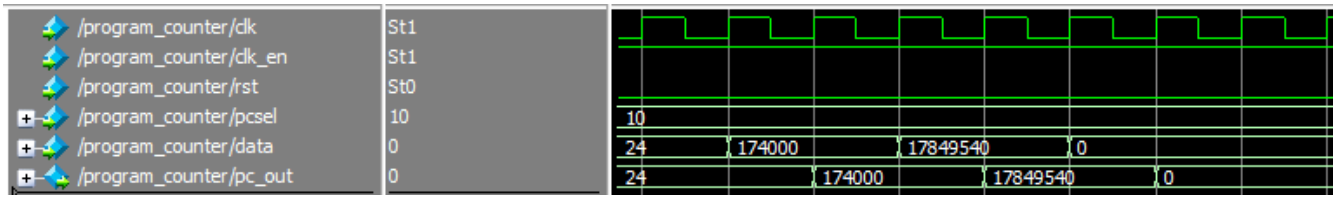

Figure 3.4: Program Counter in jmp mode simulations for various data values.

The last simulation was for the BEQ (pcsel = 01) and BNE (pcsel = 11) modes in order to check different offsets (positive and negative) with data fixed at a certain value and then at 0. Results are given in Figure 3.5 for BEQ and 3.6 for BNE.

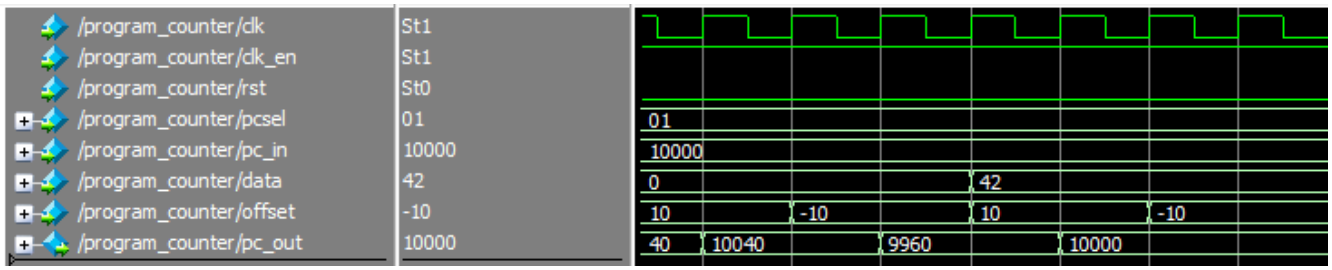

Figure 3.5: Program Counter in BEQ mode simulations for various data and offset values.

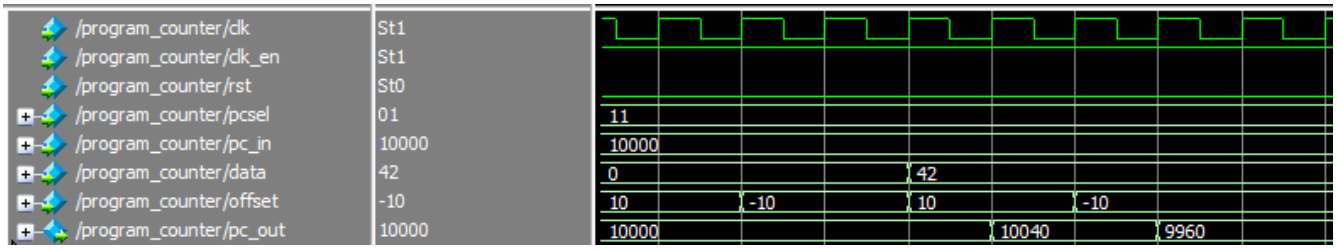

Figure 3.6: Program Counter in BNE mode simulations for various data and offset values.

In the different simulations the results are all valid, which adds confidence to this design before moving on to the practical implementation that comes later.

3.1.2 ALU

The Arithmetic Logic Unit (ALU), as its name suggests, is used for arithmetic, logic and bit shifting operations. These operations are selected using the alufn signal. The chosen operation is then applied on the two operands data_a and data_b and the result is sent on res. The various operations are listed in Table 3.2[1] and the module is pictured in Figure 3.7.

The internal design of the ALU, given in Figure 3.8, is very simple. Indeed, all operations are executed in parallel and a multiplexer is in charge of selecting the result that appears on the output. This value is then recorded in the register on the rising edge of the clock if the clk_en is in the high state. For the ALU, there is no reset, that's why the register reset is always 0.

For the implementation of the ALU in Verilog, the operators are directly used in the code and the compiler selects the right circuits to perform them. The arithmetic operations are performed by DSPs while the logic operations are performed directly by logic gates. Concerning the shifts, they are basically free on an FPGA. Indeed, this essentially correspond to shifting the connections of the signals.

[1] unsigned(data_b[4:0]) in the table means that only the five last bits of data_b are used and that they are interpreted as an unsigned number

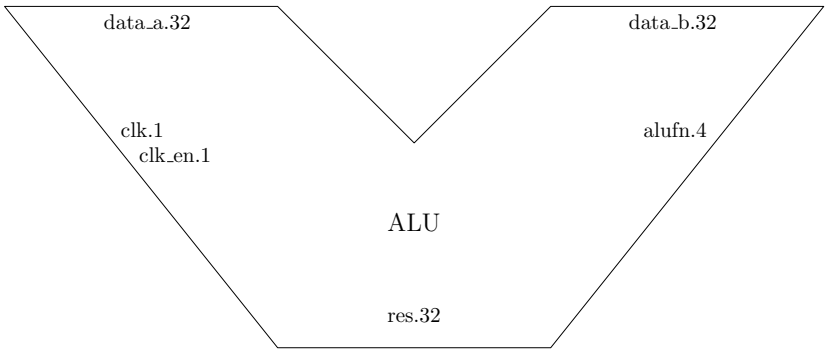

Figure 3.7: Arithmetic Logic Unit (ALU).

Operation	alufn	Result
No operation	0b0000	res = 0
Addition	0b0001	res = data_a + data_b
Substraction	0b0010	res = data_a - data_b
Multiplication	0b0011	res = data_a × data_b
No operation	0b0100	res = 0
Bitwise-and	0b0101	res = data_a & data_b
Bitwise-or	0b0110	res = data_a | data_b
Bitwise-xor	0b0111	res = data_a ⊕ data_b
Compare equal	0b1000	res = (data_a == data_b) ? 1 : 0
Compare less	0b1001	res = (data_a <data_b) ? 1 : 0
Compare less or equal	0b1010	res = (data_a ≤ data_b) ? 1 : 0
Logical left shift	0b1011	res = data_a << unsigned(data_b[4:0])
Logical right shift	0b1100	res = data_a >> unsigned(data_b[4:0])
Arithmetic right shift	0b1101	res = data_a >>> unsigned(data_b[4:0])
No operation	0b1110	res = 0
No operation	0b1111	res = 0

Table 3.2: ALU operations.

ALU simulations

In order to check the behavior of the ALU, all the operations have been tested with different operands : the tested combinations data_a, data_b are listed in Table 3.3. All the results are correct for these test cases.

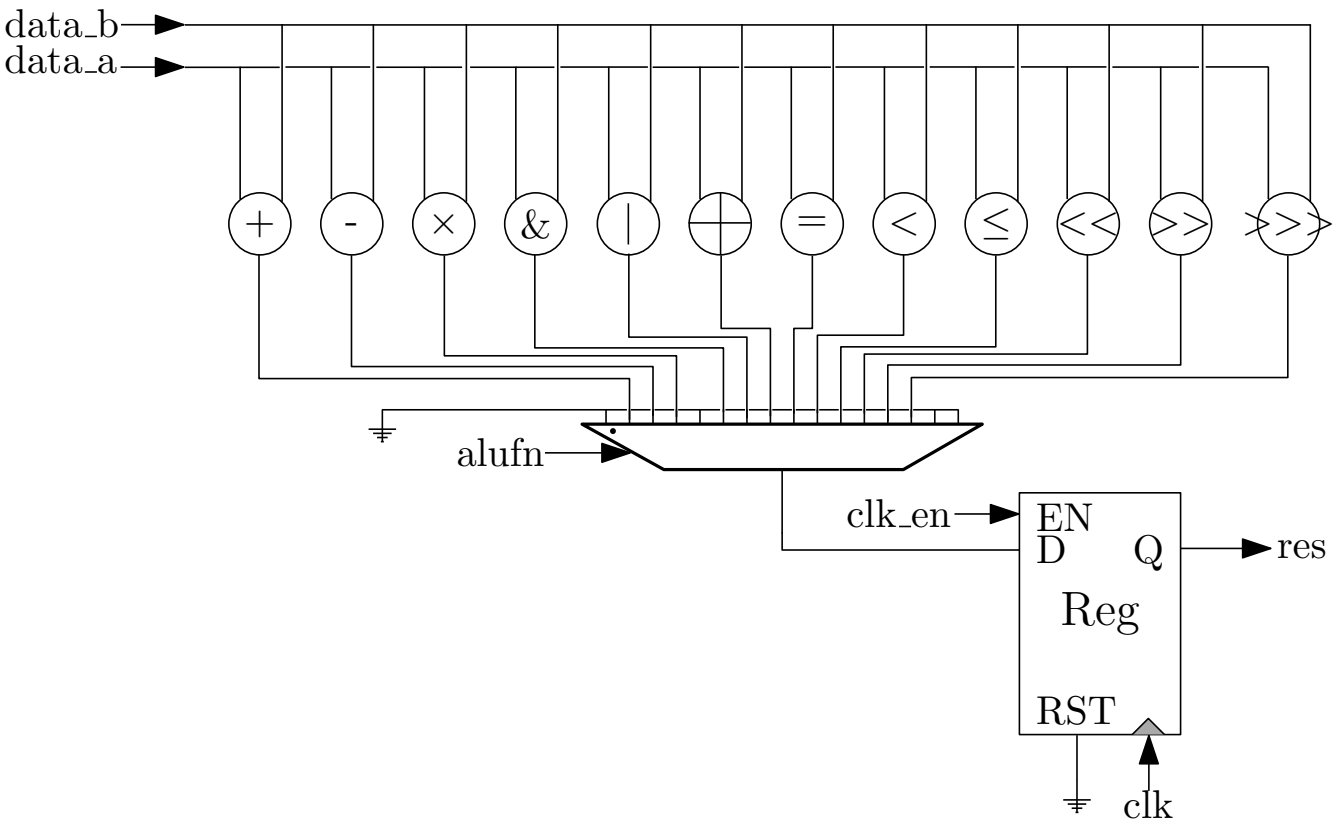

Figure 3.8: Arithmetic Logic Unit (ALU) internal circuit.

data_a	data_b
800	8
0	0
0	8
8	0
-1	8
8	-1
-8	8
8	-8

Table 3.3: Operand combinations used in the ALU simulations.

The results of the simulations are shown in Figures 3.9 to 3.16.

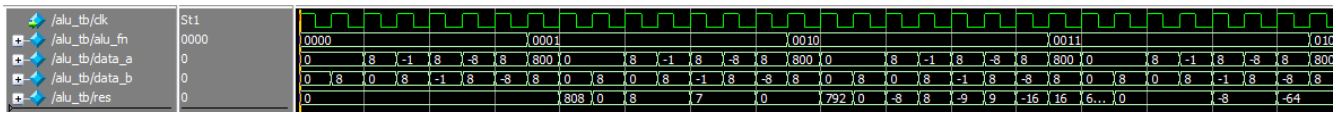

Figure 3.9: ALU simulations for alufn = 0b0000 (No operation), 0b0001 (Addition), 0b0010 (Substraction) and 0b0011 (Multiplication). data_a, data_b and res values are shown in signed decimal representation.

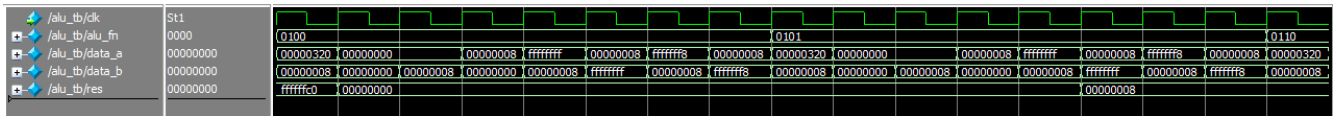

Figure 3.10: ALU simulations for alufn = 0b0100 (No op) and 0b0101 (Bitwise-and). data_a, data_b and res values are shown in hexadecimal representation.

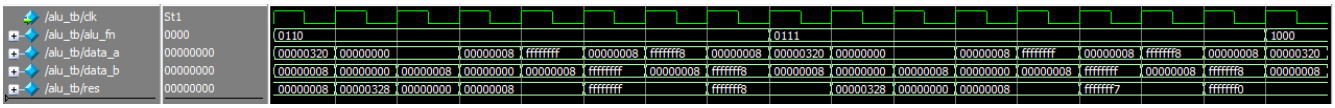

Figure 3.11: ALU simulations for alufn = 0b0110 (Bitwise-or) and 0b0111 (Bitwise-xor). data_a, data_b and res values are shown in hexadecimal representation.

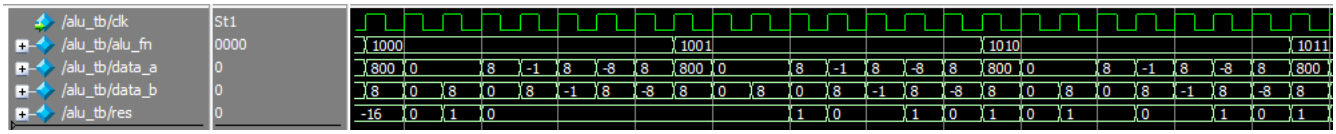

Figure 3.12: ALU simulations for alufn = 0b1000 (Compare equal), 0b1001 (Compare less) and 0b1010 (Compare less equal). data_a, data_b and res values are shown in signed decimal representation.

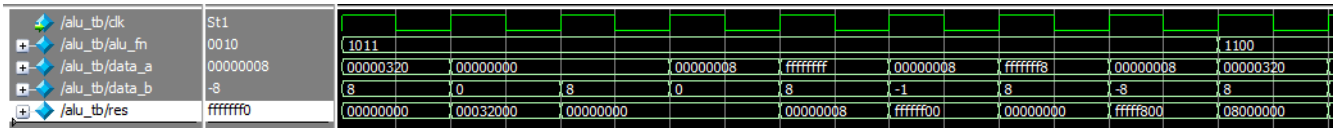

Figure 3.13: ALU simulations for alufn = 0b1011 (Logical left shift). data_a and res values are shown in hexadecimal representation. data_b is shown in signed decimal representation.

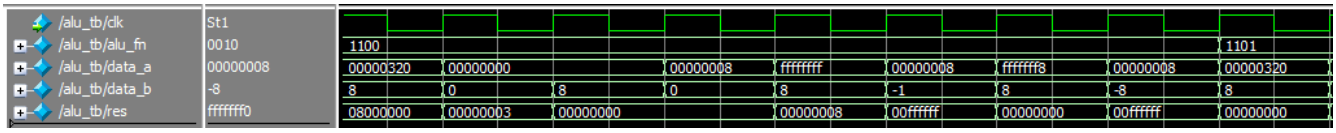

Figure 3.14: ALU simulations for alufn = 0b1100 (Logical right shift). data_a and res values are shown in hexadecimal representation. data_b is shown in signed decimal representation.

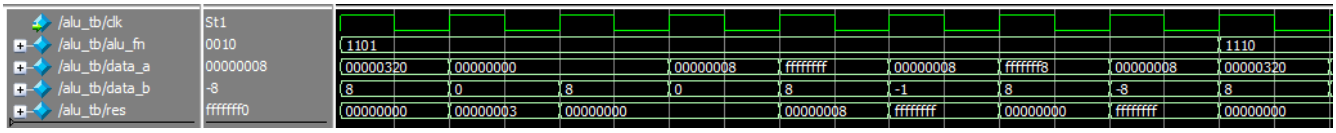

Figure 3.15: ALU simulations for alufn = 0b1101 (Arithmetic right shift). data_a and res values are shown in hexadecimal representation. data_b is shown in signed decimal representation.

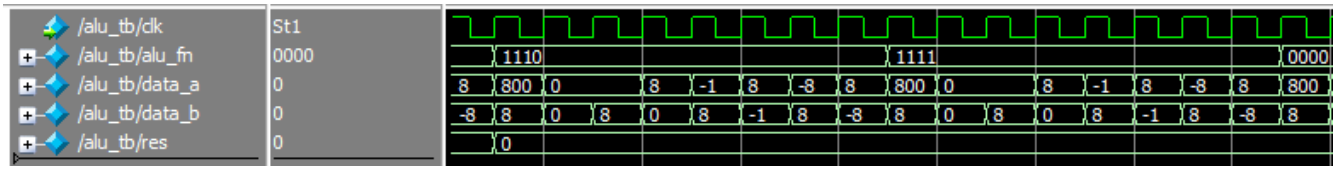

Figure 3.16: ALU simulations for alufn = 0b1110 (No operation) and 0b1111 (No operation). data_a, data_b and res values are shown in signed decimal representation.

3.1.3 Memories, generalities

On the Cyclone V, all memory definitions are specified through the same interface: the altsyncrams. Whether it is for RAM, ROM, one port, two ports, two real ports (independent clocks), ... this module is used. Also, both MLAB and M10K are represented by this Verilog module. Obviously, for all these different functions, the module must be configured. This is done on the one hand by a parameterization (for example to specify the size of the memory, the size of the words, the choice of the type MLAB or M10K, the number of ports, of clocks, the inputs / outputs which are registered, if there is a clock enable or not, ...) and on the other hand by the inputs and outputs which are used or not (for example for the ROM all the ports linked to the writing are unused and the write enable is put at the ground, in the low state). In Figure 3.17 the most general interface with two true ports is shown.

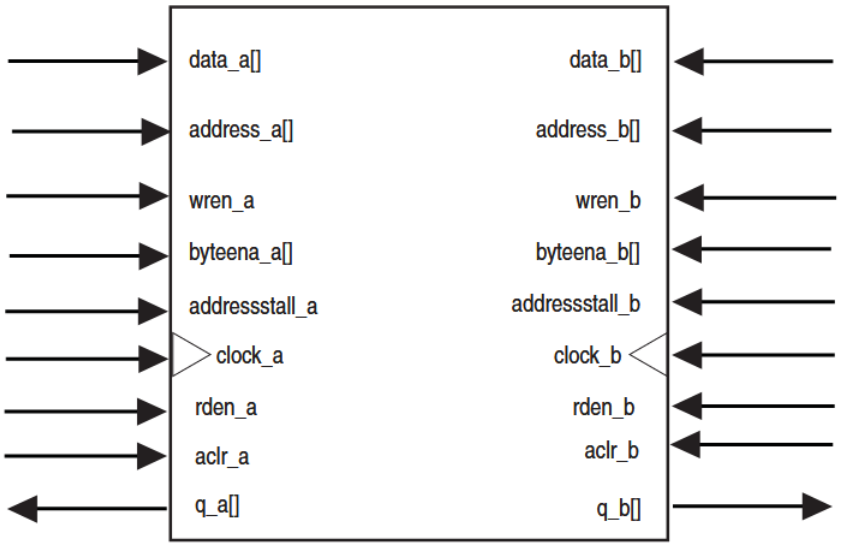

Figure 3.17: Altsyncram - True Dual-port mode [4].

Fortunately for the user, all this can be done through a graphical interface: with the Megawizard, seen earlier in this report. This tool is therefore used intensively for the generation of different memories, which is the subject of the following sections.

3.1.4 Instruction memory

The purpose of the instruction memory is, as its name indicates, to contain instructions. These, as detailed later, are coded on 32 bits. It is thus necessary to have words of 32bits. The access to the memory only needs to be done in read mode, to read the instructions, so it would be tempting to use a ROM. However, it is also necessary to have a write access in order to be able to modify the program dynamically, without having to pass by a recompilation and reprogramming of the FPGA. For the configuration of the Altsyncram, the memory only has one port, the outputs are not registered in order to have the results of the reads and writes after one clock cycle (the inputs are obligatory registered), only one clock is used and clock enable signals are added. The only thing left to do is to choose the size of the memory. For the instruction memory, it is set to 32 768 words (thus 32 768 operations coded on 32 bits), which corresponds to an address aligned on words of 15 bits. This memory described with the Megawizard is called ram_32768 in the project, its ports are shown in Figure 3.19.[2]

Now that the module based on the altsyncram is created, an interface to enable access from the rest of the beta machine is required. One also needs to add the programming ports. The module that represents this interface is actually the instruction memory itself, the module ports are shown in Figure 3.18.

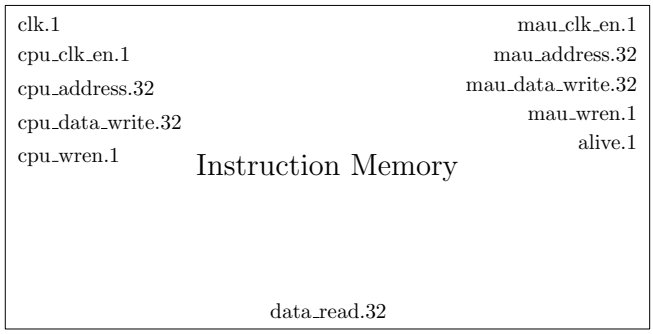

Figure 3.18: Instruction memory.

In fact, two different buses can be chosen, with the bit named alive: one that is used for accesses from outside the CPU (from the Memory Access Unit, this is discussed in another section) and one for access from the CPU. There is therefore a multiplexer on each input which chooses one of the two buses according to the value of alive. These signals are then redirected to the ram_32768 to access the memory.

[2]The notation cpu_address[16:2] in the figure means that bits 2 to 16 of cpu_address are selected. The other bits are unused.

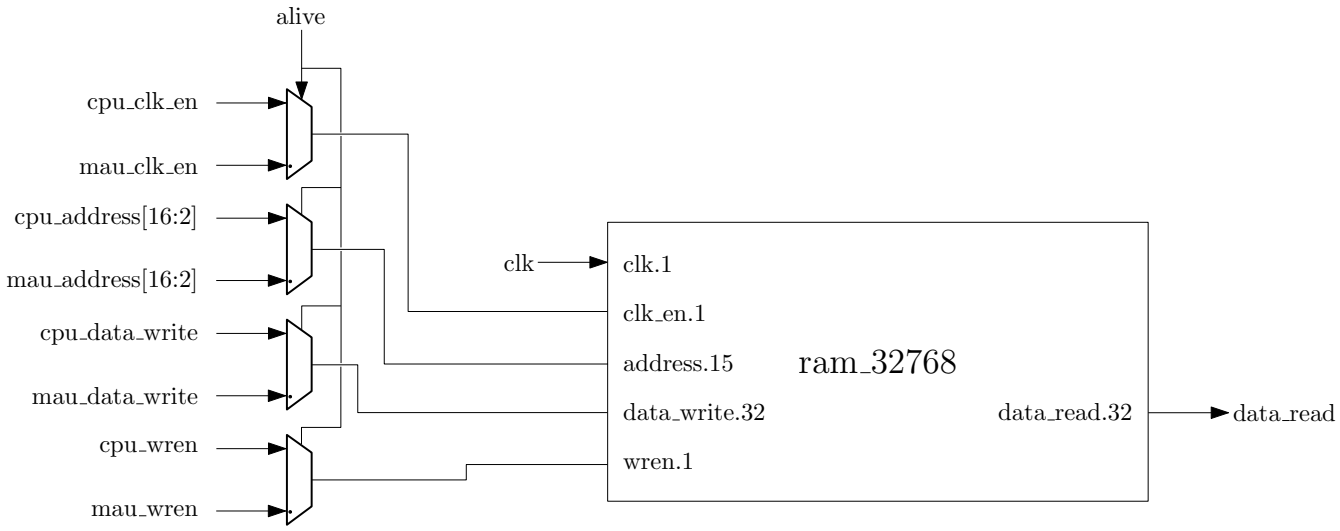

Figure 3.19: Instruction memory internal circuit.

3.1.5 Data memory

The data memory is used to store data from the CPU. It corresponds to the main memory of the CPU.

This memory is similar to the instruction memory. The only thing that changes is the size of the internal RAM. Indeed, this one is 16384 words of 32 bits. This value has been chosen to be twice as small as the instruction memory, so that both fit together in the FPGA (in addition to the other memories that are presented later). The interface of this module is displayed in Figure 3.20 and its internal circuits in Figure 3.21.

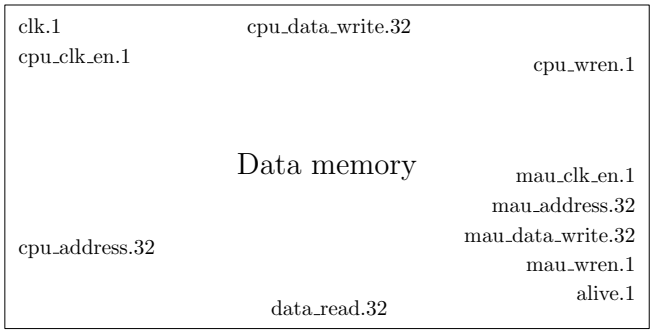

Figure 3.20: Data memory.

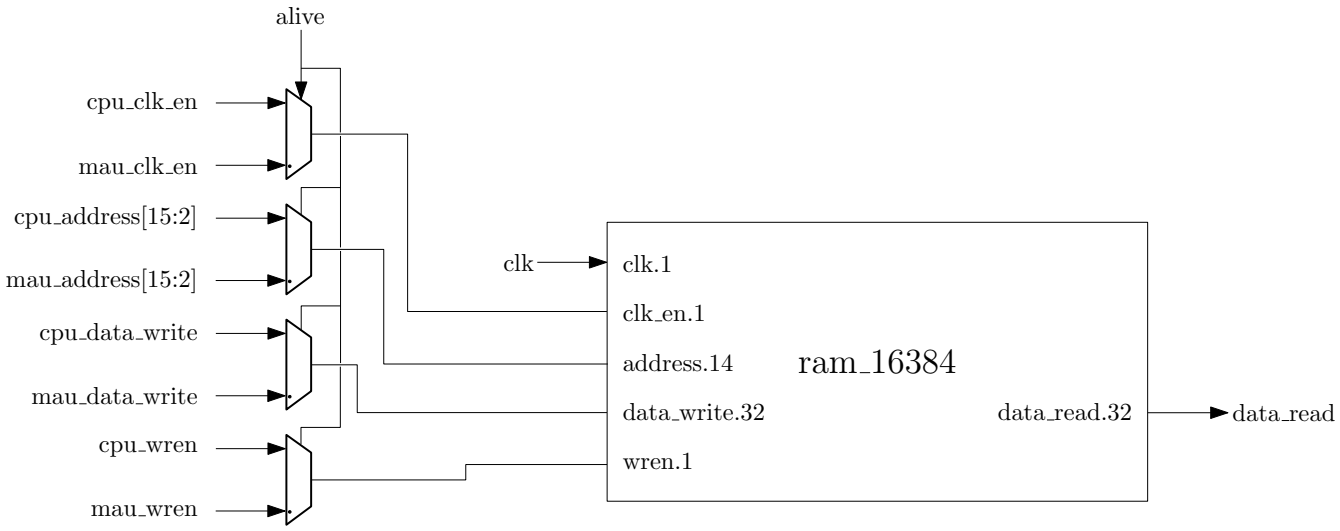

Figure 3.21: Data memory internal circuit.

3.1.6 Register file

The purpose of the register file is to provide access to 32 registers to the CPU. Two simultaneous reads and one write operations may occur. Register 31 has a special role. A read to this register always results in a 0 value written on the data_read output.

In the register file, two memories of 32 words of 32 bits are used, this memory then represents the 32 registers. As the register file needs two simultaneous read and write accesses, it is mandatory to use two separate memories because Altsyncrams do not support tri-port accesses. By having two rams with two ports, one port of each can be used to perform mirrored writes and the other port is used to have a read access. This gives two independent reads and a single write. In fact, the memories are copies of each other in this configuration.

Although writing and reading are never enabled at the same time in this work (the reason is explained later in this chapter), the register file implements all three ports anyway in order to provide the correct interface for future works,

Four ports are used to control the operation of the register file. First of all, the two clock enable signals (cpu_clk_en) allow reading for cpu_clk_en[0] and writing for cpu_clk_en[1]. Then, a cpu_wren signal indicates whether writing operations are allowed. However, in the case where the writes and reads of the register file are executed in two different steps, cpu_wren and cpu_clk_en[1] must be in the high state for a write to take place. In case a user would like to use all the ports simultaneously, it would be required to set both cpu_clk_en signals to the high state. In this configuration, writing is then totally controlled by the cpu_wren signal. Finally, the alive signal allows, as for the instruction and data memories, to make the register file listen to the CPU or MAU buses. The different ports of the register file are shown in Figure 3.22.

For access from the MAU, the register file exposes only one port which is used either for reading or writing (depending on the state of mau_wren). That is why the mau bus only has one address signal, a read signal and a write signal. When writing, the value is stored on both memories, while reading only operates on one memory.

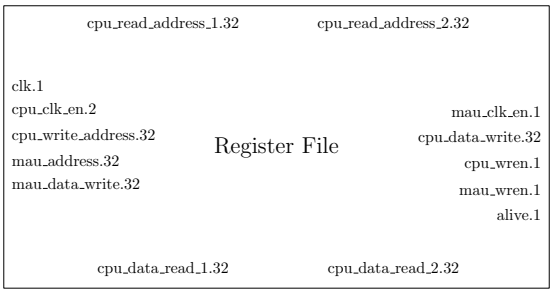

Figure 3.22: Register file.

As shown in Figure 3.23, although two memories are now present, the internal circuit remains very similar to that of the memories previously seen. The notable differences are at the level of the cpu_clk_en and of the cpu_wren where the two clock enable signals must now be managed. At the output, a multiplexer is positioned on each of the two data-read signals in order to make a selection between the value in memory and 0. The value 0 is put on the output when the input address points to register 31.

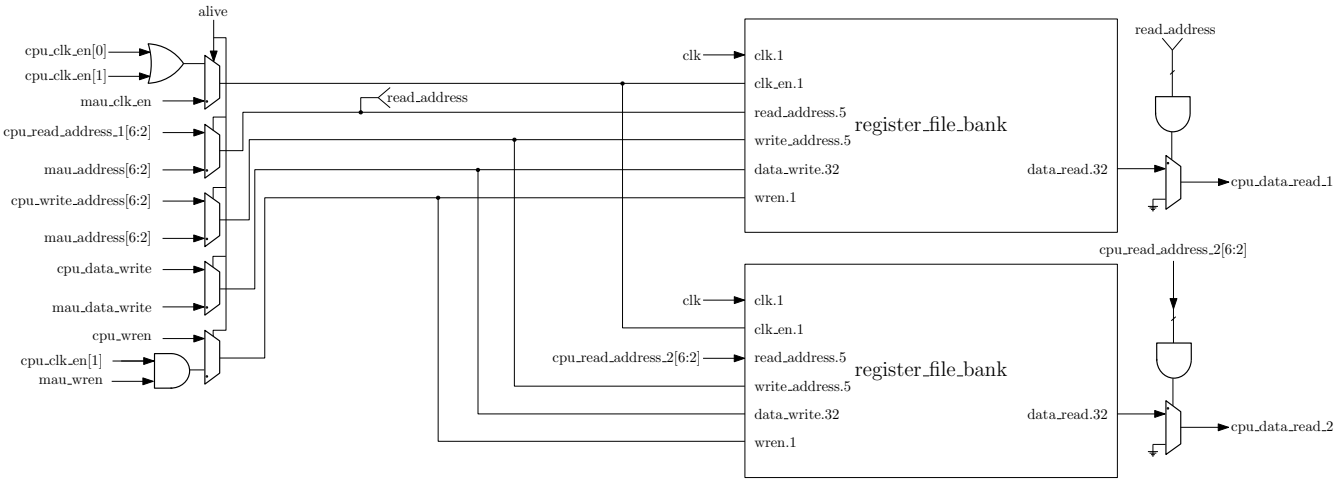

Figure 3.23: Register file internal circuit.

3.1.7 Control logic

The control logic is the core element of the cpu. Its purpose is to convert the opcode of an instruction into a set of control signals that adequately configure the different modules of the system so that the instruction executes properly. The control logic used in this work is simply a ROM (thus an Altsyncram configured in read-only mode) whose address is the opcode. The read value contains all control bits. As seen in the following section, the opcodes of the Beta machine are represented on 6 bits. The addressing of the control logic is therefore done on 6 bits. The words contained in the memory are represented on 12 bits.

The ROMs can be written at the compilation of the project on Quartus. This is done by associating a hexadecimal file (.hex) to the ROM. The configuration of this file and the meaning of the different bits of the words in this memory are explained later in the report.

This module being a wrapper of the Altsyncram configured in ROM, only the interface of this module is given, in Figure 3.24.

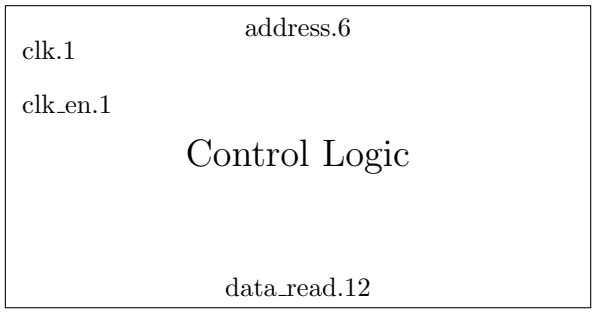

Figure 3.24: Control logic.

3.1.8 Clock controller

As mentioned earlier, the different modules making up the machine are executed in sequence and not in parallel. This allows to avoid pipelining in the machine and to simplify its realization. The execution of an instruction takes seven clock cycles. Implementing pipelining could be a very interesting and useful subject for future works.

For this sequence to be realized, the modules must be executed in the right order. It is therefore important to have a module that generates the sequence. This module is the clock_controller. It generates a signal named clk_sequence which contains the seven clk_enable signals which orchestrate the machine. The interface of the clock_controller is given in Figure 3.25.

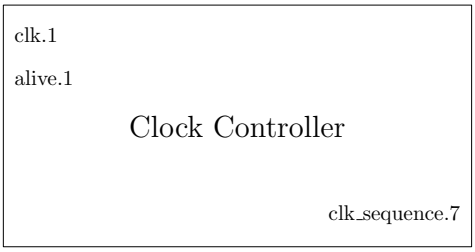

Figure 3.25: Clock controller.

At the level of the realization, this module is very simple, it initializes a register to $0b0000001$ when the alive signal is low, then when alive is high, it carries out a circular logical left shift at each clock cycle. Thus, only one bit of the clk_sequence signal is high at any time, and thus only one module is activated. The internal circuits of this module are shown in Figure 3.26. It is then sufficient to associate the generated signal with the clk_en signals of the different modules in the right order to obtain the expected execution sequence.

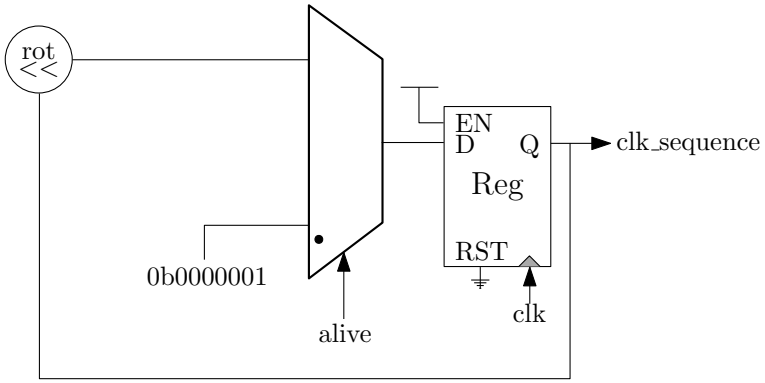

Figure 3.26: Clock controller internal circuit.

3.2 Instruction Set Architecture

3.2.1 Instructions formats

Two instruction formats are used in the instruction set. The first one is used for all instructions requiring two operands (R_a and R_b in Figure 3.27) which is for example the case for simple arithmetic instructions. The second format is used for all instructions requiring a single operand and a constant. This is the case for example for conditional jump operations. As can be seen in Figure 3.27, the R_x contain register addresses. As they are on 5 bits and no bank change operation exists, 32 registers are addressable. R_c stores the result of the instruction when a result exists. It should be noted that the constants are on 16 bits and signed.

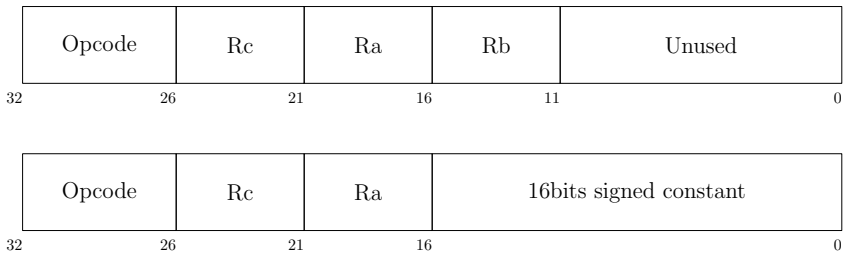

Figure 3.27: Instructions format.

3.2.2 Instruction set

Arithmetic and logic instructions

The different arithmetic and logic instructions are managed by the Arithmetic Logic Unit (ALU). The different instructions listed in Table 3.4 are supported.

Name	Operation	Opcode	Const Operation	Const Opcode
Addition	ADD	0x20	ADDC	0x30
Substraction	SUB	0x21	SUBC	0x31
Multiplication	MUL	0x22	MULC	0x32
Bitwise and	AND	0x28	ANDC	0x38
Bitwise or	OR	0x29	ORC	0x39
Bitwise xor	XOR	0x2A	XORC	0x3A
Compare equal	CMPEQ	0x24	CMPEQC	0x3A
Compare less	CMPLT	0x25	CMPLTC	0x35
Compare less or equal	CMPLE	0x26	CMPLEC	0x36
Logical left shift	SHL	0x2C	SHLC	0x3C
Logical right shift	SHR	0x2D	SHRC	0x3D
Arithmetic rifht shift	SRA	0x2E	SRAC	0x3E

Table 3.4: Arithmetic and logic instructions.

All these instructions simply perform the operation associated with them on the operands R_a and R_b in the normal case and on the operands R_a and the constant in the case of instructions with constants. R_a is always used as the first operand. For example, the instruction given in Figure 3.28 results in $[R_{10}] = [R_{14}] - 17$. The notation $[R_x]$ means the content of register R_x.

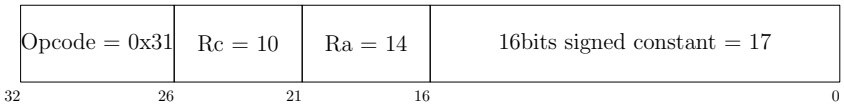

Figure 3.28: SUBC instruction example.

The comparison operations simply put a 0 in R_c when the comparison is false, and 1 otherwise.

Memory instructions

Here, only two instructions are present. One is to load a value from the data memory and the other is to store a value in it. The notation $< x >$ means at address x in memory. Note that here R_c doesn't necessarily contains a result at the end of an operation and is used as an operand for ST. Also, addresses must be byte aligned here and offsets contained in Constant must also be byte aligned. Both instructions are detailed in Table 3.5.

Name	Operation	Opcode	Action
Load	LD	0x18	[Rc] = <[Ra] + Constant>
Store	ST	0x19	<[Ra] + Constant> = [Rc]

Table 3.5: Memory instructions.

Program counter instructions

The purpose of these instructions is to modify the value of the program counter conditionally or not. The different instructions are listed in Table 3.6. The values of column Next PC is written to PC and the address of the next instruction before the jump is stored in Rc. Note that [Ra] should be byte aligned for JMP but the constant should be 32 bits aligned for BEQ and BNE (as they already multiply the constant by 4).

Name	Operation	Opcode	Condition	Next PC
Jump	JMP	0x1B	None	[Ra] & 0xFFFFFFC
Branch Equal	BEQ	0x1D	[Ra] == 0	PC + 4 × (Constant + 1)
Branch Not Equal	BNE	0x1E	[Ra] != 0	PC + 4 × (Constant + 1)

Table 3.6: Program counter instructions.

Control instructions

Only one control instruction exists. This instruction bascally stops the machine and has no operand or result. The instruction is detailed in Table 3.7

Name	Operation	Opcode
Exit	EXIT	0x3F

Table 3.7: Exit instruction.

3.3 Beta machine

The final circuit of the beta machine is shown in Figure 3.29. The details of this circuit and how the modules have been interconnected are explained in this section.

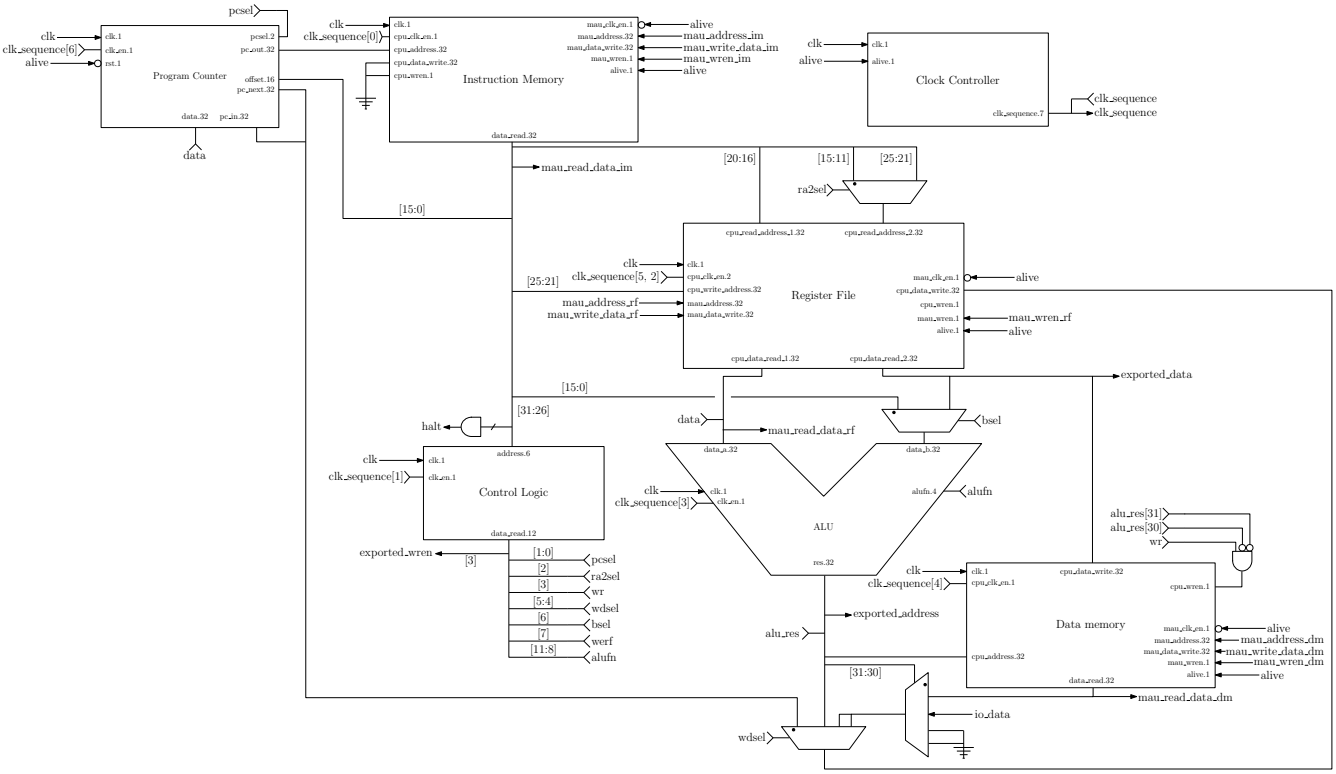

Figure 3.29: Beta machine internal circuit.

3.3.1 Module sequence

The first thing to establish is the execution sequence. The first step is to fetch an instruction from the memory instruction, so it is executed first. Then, the opcode of this instruction must be decoded by the control logic. The control logic has the second position. The register file can now be operated in read mode, in order to obtain the values of the registers useful in the current instruction. This step could not have been carried out at the same time as the previous one because a multiplexer controlled by a signal from the control logic is present at the cpu_read_address_2 input of the Register File. This allows it to choose between Rb and Rc, Ra always being the address of the first port.

The values of the registers are now present at the output of the Register file, the ALU is then executed. In case the result of the ALU should be stored in Data memory, it is then executed. Also, when executing the Data memory, it could provide data that must be stored in a register. That is why the next step is the execution of the Register file in write mod. Finally, the Program counter is enabled so that the PC is updated. The execution sequence is displayed in Figure 3.30.

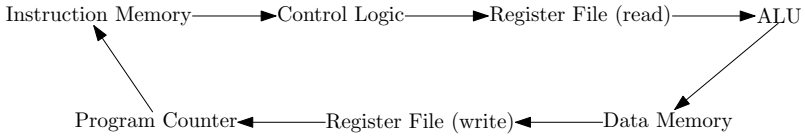

Figure 3.30: Execution sequence.

3.3.2 Instructions and control logic

In this section, the content of the control logic is defined. At the same time, the connections between the different modules forming the Beta machine are also explained. The complete circuit of the Beta machine is shown in Figure 3.29. The reader can refer to it to follow the developments of this section.

First of all, the output pc_next of the program counter is connected to its input pc_in to form a counter by 4. The output pc_out of the program counter can be connected to the CPU address of the memory instruction in order to read the current instruction. The first six bits of this instruction are connected to the address of the control logic so that the opcode is decoded. Of course, the clk and clk_enable signals of all modules are also set to follow the sequence described in the previous section.

Arithmetic and logic instructions

In order for an ALU instruction to perform correctly, two things are required. First, the necessary registers must be read from the register file and second the ALU must execute the operation. To do this, the instruction coming from the instruction memory is split. The bits corresponding to Ra go to the address of the first port and the bits corresponding to Rb to the second port. As the results are stored in the register Rc, the bits corresponding to Rc in the instruction are used as the write address for the register file. The two operand registers can then be loaded.

For the ALU, it needs on its first input the value read in the register Ra, this is why the reading of the first port directly goes to the first input of the ALU. As for the second operand of the ALU, it can either be the value of Rb or a constant. This is why a multiplexer is placed on the input corresponding to the second operand. Finally, the result of the ALU is redirected to the register file so that it can store this value in Rc.

At the control logic level, three signals are necessary. The first is used to control the operation of the ALU, it is alufn. The second is bsel and allows to choose if the second operand of the ALU is the constant or the value read in register Rb. And last, the signal werf allows to authorize the write operation in the register file. The current content of the control logic is shown in Table 3.8.

Instruction	alufn	werf	bsel
ALU instruction	opcode	1	1
ALU constant instruction	opcode	1	0

Table 3.8: Control logic - ALU instructions.

Memory instructions

As shown in Table 3.5, the load operation (LD) executes [Rc] = <[Ra] + constant>. To make this operation possible, a sum with constant must be performed and its result used as an address for the data memory. This is why the output of the ALU is set as an address for the data memory. Then the value read from the data memory must be used as a write value to the register file. As

a signal has already been connected to this register file entry, a multiplexer must be added which chooses between the ALU output and the value read from the data memory. This also means the control logic must provide a signal, named wdsel, to drive the newly added multiplexer.

For the storage operation in data memory (ST), <[Ra] + constant> = [Rc] must be executed. Everything remains the same as for the load at the ALU and the first port of the register file. However, the value in Rc must be read in the register file and nothing allows it until now. As Ra is already using the first port, Rc is put on the second port for reading. A multiplexer must be added to it as Rb was already associated with this port. The output of the second port of the register file must be connected to the cpu_data_write input of the data memory so that [Rc] is written there.

Two new signals are added to the control logic. The first, wr, allows the data memory to be written and the second, ra2sel, drives the multiplier which has just been added. The current content of the logical control is displayed in Table 3.9.

Instruction	alufn	werf	bsel	wdsel	wr	ra2sel
ALU instruction	opcode	1	1	01	0	0
ALU constant instruction	opcode	1	0	01	0	x
Load instruction	add	1	0	10	0	x
Store instruction	add	0	0	xx	1	1

Table 3.9: Control logic - ALU and memory instructions.

Program counter instructions

For the JMP instruction, it is first necessary to save the value of PC + 4 in Rc. This means that the pc_next output of the program counter must be usable as a write value for the register file. pc_next is therefore added to the multiplexer driven by wdsel. Then, the value of PC must be set to the value given in Ra. The reading of the first port of the register file is therefore connected to the data port of the register file. The register file must also be set to JMP mode.

Only one signal, pcsel, must be added to the control logic: it will allow the program counter mode to be set.

For the branch instructions, everything happens as for JMP except that the value of pc is different in the control logic and that the constant contained in the instruction is connected to the offset input of the program counter. The final content of the control logic is given in Table 3.10.

Instruction	alufn	werf	bsel	wdsel	wr	ra2sel	pc
ALU instruction	opcode	1	1	01	0	0	00
ALU constant instruction	opcode	1	0	01	0	x	00
Load instruction	add	1	0	10	0	x	00
Store instruction	add	0	0	xx	1	1	00
JMP instruction	xxxx	1	x	00	0	x	10
BEQ instruction	xxxx	1	x	00	0	x	01
BNE instruction	xxxx	1	x	00	0	x	11

Table 3.10: Control logic - ALU, memory and program counter instructions.

In the control logic, all values corresponding to don't care (x), are set to 0.

Control instructions

The last instruction to be handled is EXIT which is intended to generate a halt signal. As the EXIT instruction corresponds to opcode 0x3F, it is sufficient to put all the bits of the opcode at the input of an AND logic gate in order to have at the output the halt signal which will be high when EXIT is called.

3.3.3 Other signals and ports

The Beta machine is only one part of the system designed during this work. In fact, it consists of the CPU of this system. It therefore has other ports which allow its connection with other super-modules.

First there are the four ports exported_address, exported_data, exported_wren and io_data, the first three allow to add listeners to the ST instruction. io_data allows, with the three signals mentioned above, to add the IO unit (IOU) as a listener to the LD instruction. In the context of this work, the first two bits of the address are used to designate the listener which is concerned by the ST and LD operations. For the LD part, it is therefore necessary to add a multiplexer before putting the result of a data memory load on the multiplexer controlled by wdsel. The purpose of this multiplexer is to select where the load comes from. io_data therefore constitutes an input of this multiplexer. The addressing table for the different units is described in Table 3.11.

Super unit	Address
CPU	00
IOU	01
GPU	10

Table 3.11: Super unit addresses.

Then, there are the halt and alive signals. These two signals allow communication with the control unit (CTRLU), their use is discussed later.

Finally, there are all the signals starting with mau. These are connected to the Memory Access Unit (MAU). Again, this unit is seen later in the report.

In the end, this gives the CPU a more complex interface than usual. This interface is that of Figure 3.31. In order to simplify the CPU interface, not all signals are put in the CPU and a bus representation is used. However, all signals are available in the internal circuit.

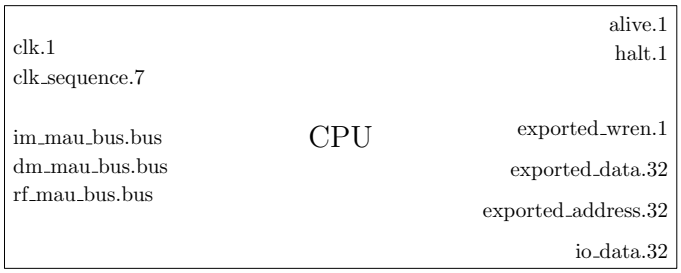

Figure 3.31: Beta machine, the CPU.

3.4 IO Unit (IOU)

Github link to the verilog files of the IOU.

The IO unit was very briefly discussed in the previous section, especially in terms of its addressing by the store (ST) and load (LD) operations. It should be noted that this module is very experimental in this work. It was mainly added to add interaction to the programs that are loaded on the beta machine. Its implementation is therefore very straightforward, naive and incomplete. Incomplete means that only three IOs of the DE10 Nano have controllers in the IOU. These three IOs are the 8 LEDs, the 2 buttons and the 4 switches respectively shown in pink, green and red in Figure 3.32.

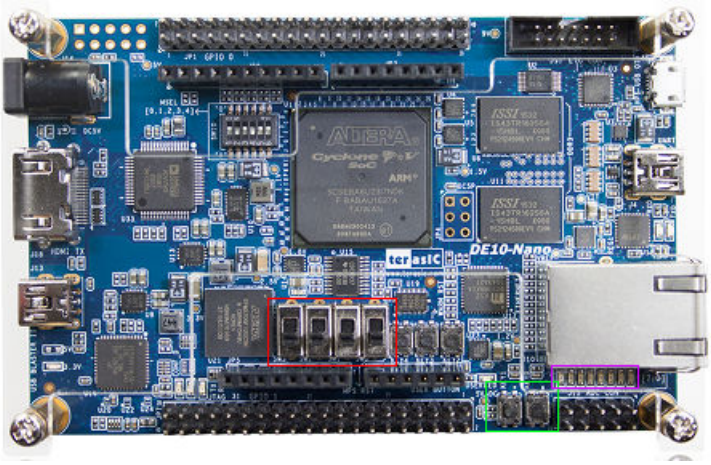

Figure 3.32: Used IOs of the DE10 Nano board.

The interface that the IOU exposes is the one in Figure 3.33. It is identical to the memory instruction except that the IOU has the three ports, i.e., buttons, leds and switches, that allow to connect this module to the physical IOs on the board.

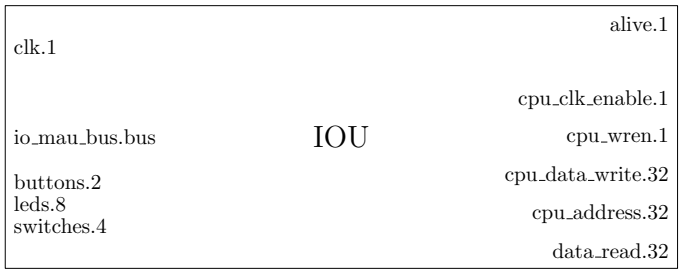

Figure 3.33: IO Unit (IOU).

The internal circuit, shown in Figure 3.34, looks again very much like a memory with the multiplexing between the two ports (MAU and CPU). Inside, a register represents an IO. The buttons and switches are read-only, the multiplexer added for writing thus has only one output port, the one of the LEDs. On the output side, the values of each register are multiplexed to data_read. As it can be deduced from the multiplexers, the addresses of the IOs are 0x0 for the buttons, 0x1 for the LEDs and 0x2 for the switches.

When it comes to reads and writes, the data word contains the value of all the elements of the addressed IO on its last bits. For example, when writing the LEDs, the last 8 bits of data_write contain the new state of each LED. To know to which bit corresponds a precise LED, the user can look at the DE10 Nano board. The LEDs, buttons and switches are identified on the board.

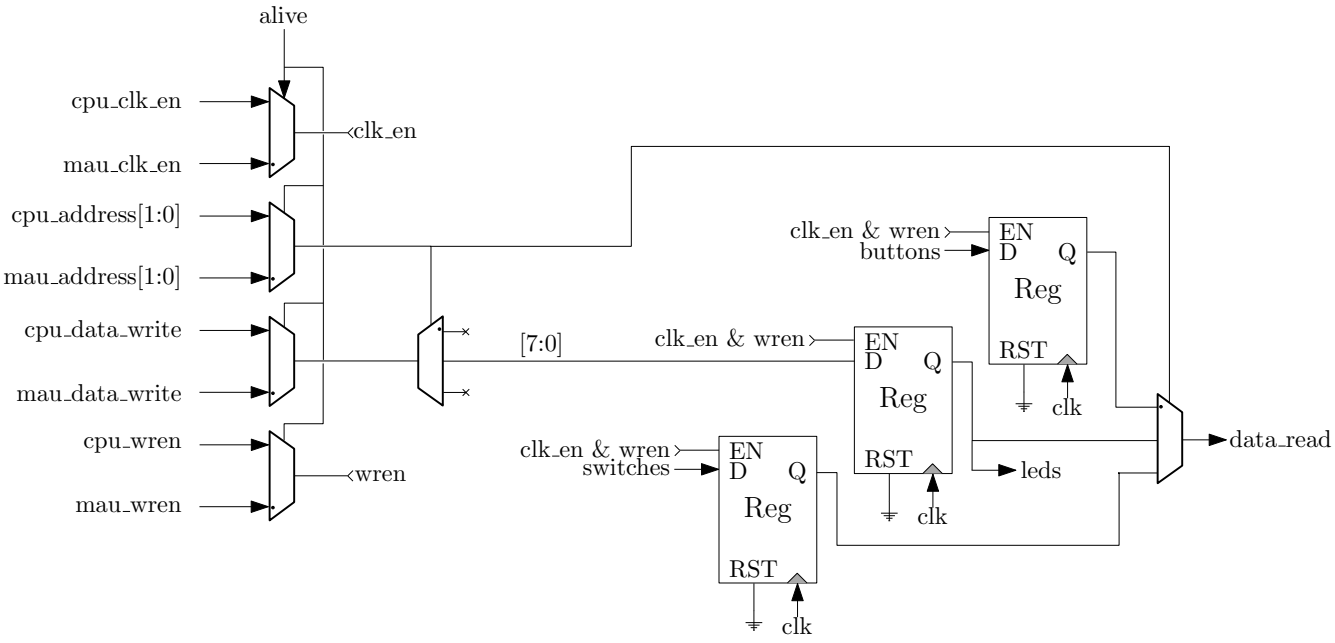

Figure 3.34: IO Unit (IOU) internal circuit.

Chapter 4

GPU and Clock Unit design

4.1 GPU

Github link to the verilog files of the GPU.

The GPU bears its name rather badly because it is not really programmable. But this is the only name that was found for it and therefore kept. This unit provides a graphic memory and the tools to draw patterns of 8x8 pixels, called masks. The writing in this memory is done from the beta machine (the CPU) through the store instruction (ST). The specific way of addressing the memory and the structure of the commands are described later. As far as the colors are concerned, it has been decided to work with 12 bit RGB colors, that is to say 4 bits per primary color.

In addition to this, the GPU also provides the HDMI controller. This controller simply reads the graphic memory in loop and draws its content on the screen. The controller handles a 16:9 screen using a resolution of 848x480 pixels and a refresh rate of 60Hz. The protocol used is the VESA 848x480, it is described later.

Before moving on to the description of the different modules making up the GPU, a description of how the screen and the masks are interpreted by the GPU is done. The VESA protocol is also detailed.

4.1.1 Screen and tile representation

As said before, the masks are 8x8 pixels and the goal is to be able to apply them anywhere on the screen. The screen is therefore naturally divided into a set of 8x8 pixels squares which are called tiles. In memory, the idea is that there are 3 sub-memories whose words contain a color of the three primary colors (red, green or blue) of a whole tile. A tile corresponding to 64 pixels and a primary color being coded on 4 bits, it gives words of 256 bits. Then, by dividing the dimensions of the screen (848x480 pixels) by 8, one finds that 106x60 tiles, so 6360, are enough to represent the screen. In order not to use too much memory, it is decided to use 72x54 tiles. This corresponds to the largest integer size allowing a 4:3[1] ratio with a maximum use of 4096 tiles. It is decided to limit

[1]This ratio corresponds to the ration between the width and the height, in terms of tiles, of the useful screen. This has nothing to do with the resolution ratio of the screen that is 16:9.

to 4096 tiles because to use 6360 tiles it would be necessary to have a memory with 8192 words, which would use too much memory unnecessarily. The useful screen is centered in the physical screen. Figure 4.1 summarizes all this.

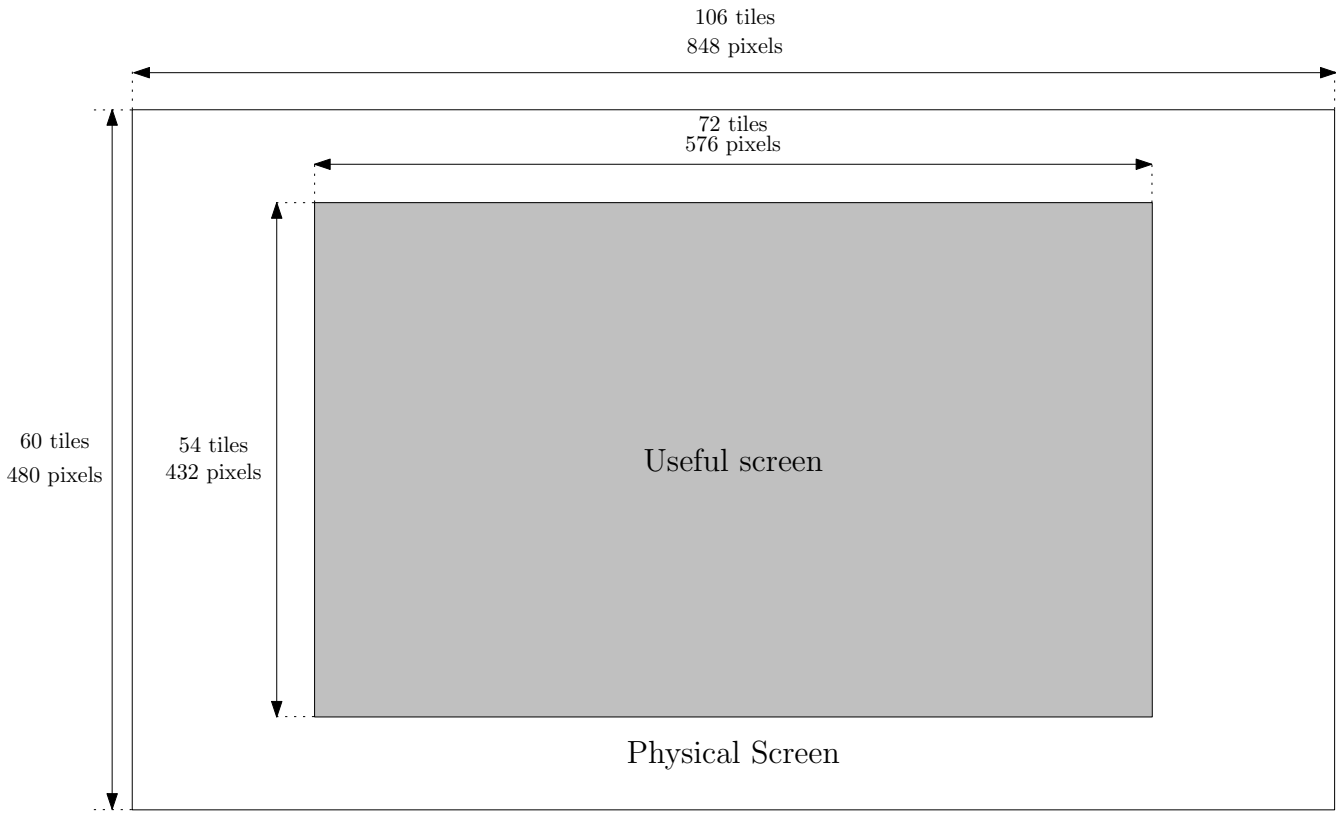

Figure 4.1: Useful screen area in the physical screen.

The problem with putting all tiles in a single memory like this is that only one tile is accessible at a time. This means that a mask can only be applied to one tile. This is a pity because it prevents the mask from being drawn anywhere on the screen. Indeed, the mask cannot for example be drawn between two tiles as shown in Figure 4.2.

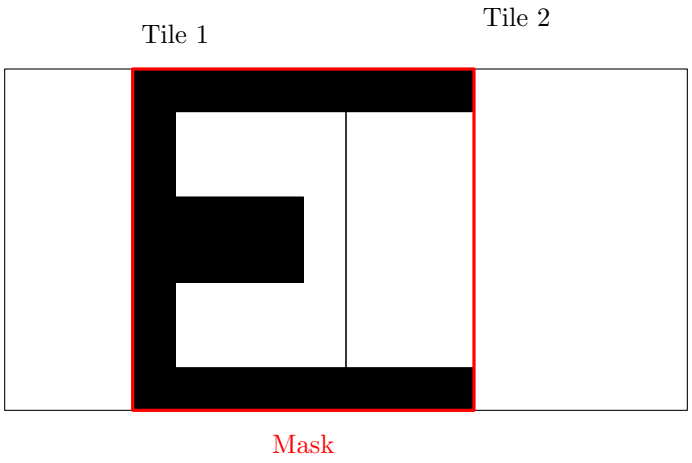

Figure 4.2: Mask overlapping two tiles.

When such a placement of the mask takes place, what actually happens is a tile is chosen first. And then an offset in pixels is added to the mask. In the example in Figure 4.2, the mask has a positive offset of N pixels with respect to Tile 1. The idea is then to allow a mask to have an abscissa and an ordinate offset ranging from 0 to 7. It is not useful to go further than 7 as this corresponds to placing the mask at the next tile. To be able to apply the mask, it is therefore necessary to be able to load four tiles. The first one being the one chosen to draw the mask in (x, y), the others being those in (x + 1, y), (x, y + 1) and (x + 1, y + 1). A naive solution to load these four tiles would be to have a memory for each tile. However, this has two problems. The first one is that this is impossible for the Cyclone V. Indeed, it was seen earlier that the Cyclone V has just over 500 M10K memory blocks. However, to represent all the tiles, one would need 3888 memories because the definition of a memory in the code must use at least one whole M10K, which is simply impossible. Secondly, this would generate way to many connections in the circuits and a complex multiplexing stage in the memory circuits. This is not tractable for the Cylone V neither.

Another way is to try to divide the set of tiles into four groups. These four groups should be distributed on the screen in such a way that the placement of a mask loads 4 tiles of different groups each time. If one thinks about it, it is possible to convince itself that a simple paving of 4 colors allows this (a color represents a group). Figure 4.3 shows such a tiling.

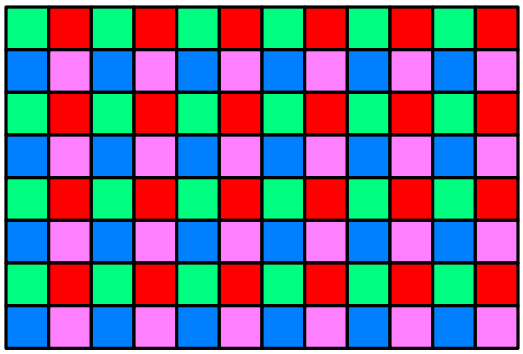

Figure 4.3: Screen tiling.

And the different cases possible by choosing four tiles in the way explained earlier, i.e. one tile, the one to the right of it, the one below it and the one at the bottom right are shown in Figure 4.4. It can be seen that in each case, a group is present only once, which validates the possibility of representing all tiles using four distinct memories.

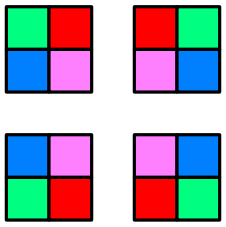

Figure 4.4: Screen tiling selection cases.

4.1.2 Mask representation

A mask has the same dimensions as a tile but does not have the same content. Indeed, a mask can contain four different values. The first value is *keep*, corresponding to a 0 in the mask. This value means that no modification is made to the pixel targeted by this position in the mask. Then there is *set primary*, corresponding to 1, which sets the pixel at this location to the primary color, this is better explained later. When the value is 2, for *set secondary*, it is then the secondary color that is applied. And finally, for the value 3 corresponding to a *reset*, the pixel becomes black. Each of these values can therefore be represented on 2 bits. Table 4.1 summarizes the possible operations of a mask.

Mask Operation	Mask Value
Keep	0b00
Set primary	0b01
Set secondary	0b10
Reset	0b11

Table 4.1: Mask operations.

In terms of representation, a mask is a simple vector of 8x8x2 (128) bits. Its LSB corresponds to the upper left corner of the mask and its MSB corresponds to the lower right corner of the mask. This correspondence is shown in Figure 4.5.

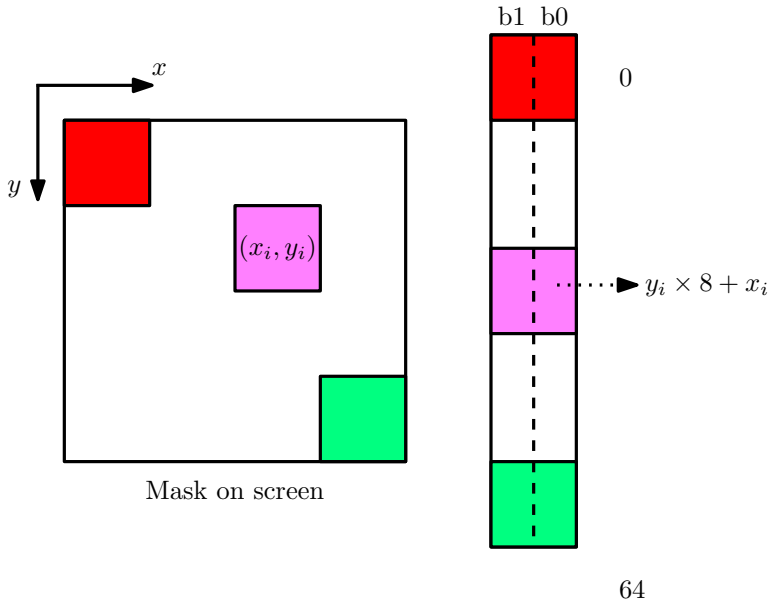

Figure 4.5: Mask vector representation.

4.1.3 Using the GPU

As introduced, to use the GPU, it is necessary to use the store instruction from the CPU. But the address and data must be correctly formatted. For the address, it must start with 0b10 as seen in the chapter on the CPU (so that the addressed unit is the GPU). Then, it was decided to

make the address natural, that is to say that it is readable without any conversion. The address therefore contains the exact pixel position where a mask should be applied. A pixel being located by four variables: block_x, block_y (corresponding to the tile), off_x and off_y (corresponding to the offset from the tile), these four variables are directly present in the address. The format of the address is detailed in Figure 4.6.

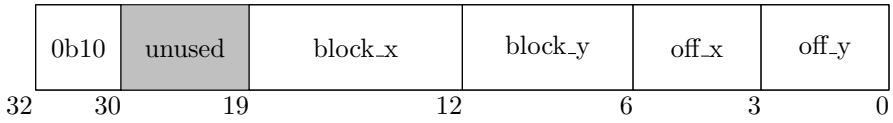

Figure 4.6: Store instruction address.

Concerning the data writen by the store instruction, it must contain the mask identifier and the values of the two colors. As a color is represented on 12 bits and a word on the CPU is represented on 32 bits, 8 bits remain for the mask. The GPU can therefore support 256 masks. The data format is detailed in Figure 4.7.

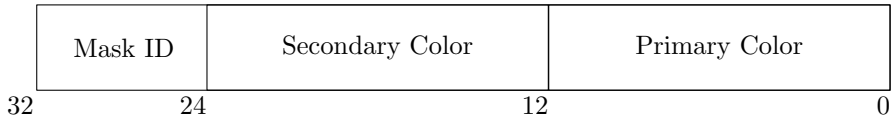

Figure 4.7: Store instruction data.

4.1.4 VESA protocol

The VESA protocol [5] works exactly like the VGA protocol. That is, four signals are present. On the one hand there is the 24 bit color signal (8 bits for each primary color, only 4 are truly used here, the least significant bits are replaced by zeros), the vertical and horizontal synchronization signals and the display enable signal. The synchronization signals inform the screen of the end of a line for horizontal synchronization and the end of a frame (all the lines on a screen) for vertical synchronization. Between two horizontal synchronization signals, several things happen. First, there is a pause just after the signal. This time is called horizontal back porch. Then, after this pause the colors are sent, the value must be maintained for a certain time to fix a pixel. All the pixels of the line are assigned one after the other, in a burst. Of course there are specific timing criteria that must be met for this to work, this is discussed next. After sending all the colors, a new pause takes place, called horizontal front porch. And finally, after this pause, a new horizontal synchronization signal is sent. For the frames, it is similar. Before the frame, there is a pause (vertical front porch) followed by the vertical synchronization followed by the vertical back porch. Note that the horizontal and vertical synchronizations are independent, so both must be evaluated at each moment. When no pause or synchronization occurs, either horizontal or vertical, the display is said to be enable. The display enable signal is then high and the colors are actually used for the pixels. Outside this condition, the color signal can be set to any value and is not listened to.

The operation of the protocol is shown in Figure 4.8. The legend and length of the signals are available in Table 4.2. In this figure, the protocol is described in relation to the pixels, and therefore the position on the screen. As the screen only includes the pixels of the region where

display enable is high, the frame shown is a virtual screen, not the physical one, which is smaller. The description is made in relation to the pixels because it is easier to understand but also to implement. Indeed, as shown later in this report, pixel counters are sufficient to implement this protocol. Then, it is enough to fix the right clock frequency so that the timings of each pixel are respected. For VESA 848x480, a 33.750MHz clock is used. Figure 4.9 shows the value of the signals according to the position on the screen.

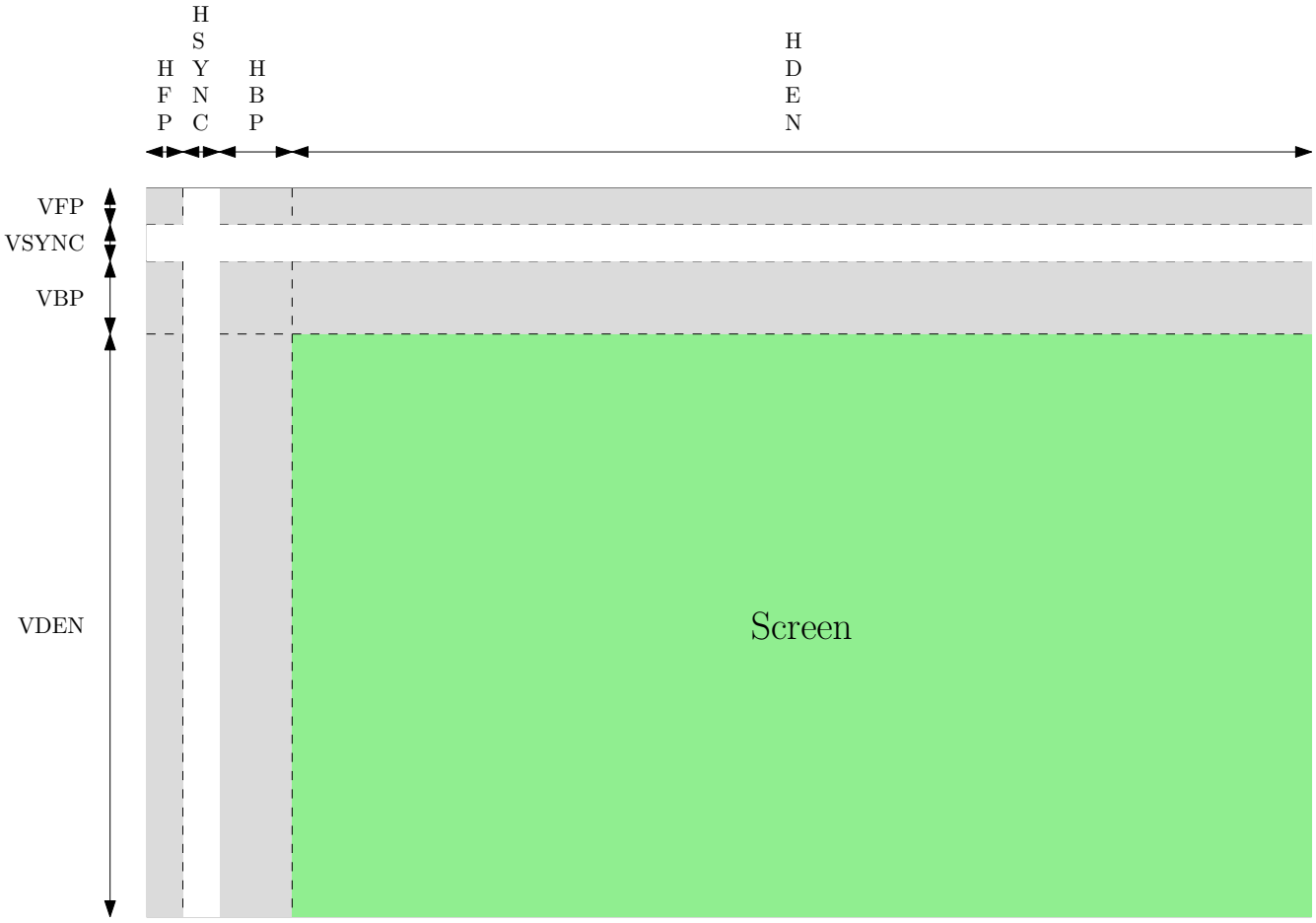

Figure 4.8: VESA virtual screen schematic.

Signal	Complete name	Length (pixels)
HFP	Horizontal Front Porch	16
HSYNC	Horizontal Sync	112
HBP	Horizontal Back Porch	112
HDEN	Horizontal Display Enable	848
VFP	Vertical Front Porch	6
VSYNC	Vertical Sync	8
VBP	Vertical Back Porch	23
VDEN	Vertical Display Enable	480

Table 4.2: VESA 848x480 counts.

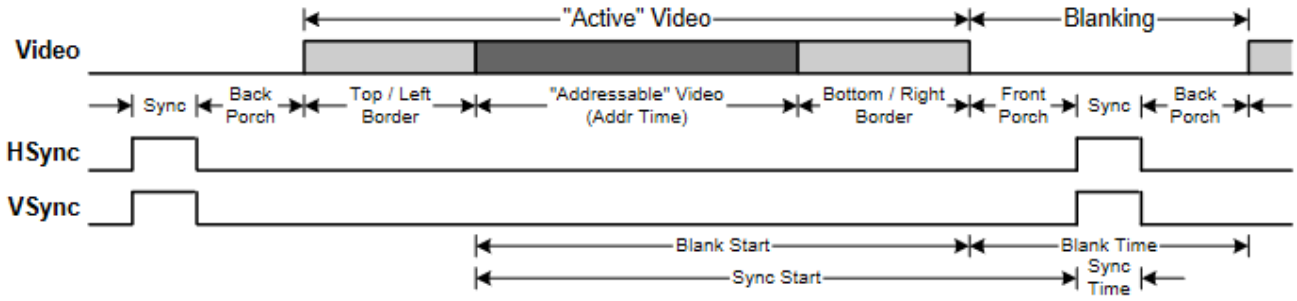

Figure 4.9: VESA signals - video signal represents both display enable signals [5].

4.2 GPU components

4.2.1 Graphic memory

The graphic memory has two functions. The first one is of course to store the state of all the pixels of the screen. This is done as discussed above by splitting the memory into four parts. One for each group of the Figure 4.3 tiling. Each of these sub-memories is then divided into three memories, one for each primary color. This was done this way because it is more optimal to have three memories whose word size is a power of two. By dividing by three, each word has 4 bits while it would have had 12 bits in the case of full colors, which is not a power of 2. A fifth port with a different clock from the first 4 ports is also exposed by the graphic memory. This one allows the reading of the memory tile by tile for the HDMI controller part of the GPU. This operation is completely independent of the CPU accesses of this memory, that is why another port is used. The tile loaded from this port is referred to as Tile b.

The second function is to correctly direct the values in memory to the outputs when reading, and the values in input to the memories when writing. Indeed, depending on the editing position (address), the current tile belongs to one of the memories. And the memories of the other three tiles to be selected also depends on this position. See Figure 4.4 for the different possible configurations.

From now on, the current tile (at the target position) will be tile 0. The one to the right of it will be tile 1, the one below it will be tile 2 and the one below it on the right will be tile 3. Figure 4.10 summarizes this. Note that the colors are set by tiles. A memory word is therefore $4 \times 64 = 256$ bits.

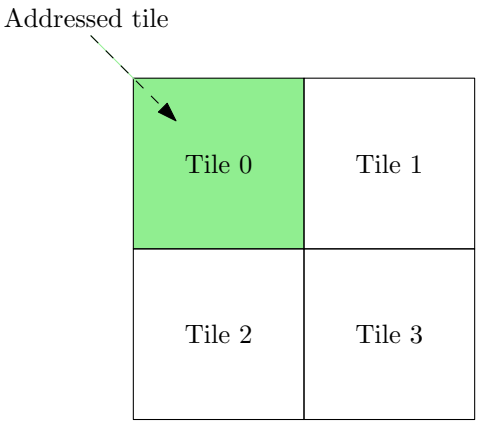

Figure 4.10: Tile identifiers.

Memory unit

The memory units are the modules representing the groups of tiles, so there are 4 of them in the graphic memory. Each memory unit is composed of three Altsyncram configured in true-dualport RAM representing the three primary colors. True dualport means that they come with two completely independent ports whose clocks are different. The module simply interfaces these three Altsyncrams. The internal circuit is given in Figure 4.12 and its interface in Figure 4.11. As can be seen, the wren_b and data_b signals of each memory are grounded since the second port is read-only, that is the one corresponding to Tile b. The HDMI controller doesn't need to write in memory. For the rest, the signals are simply retransmitted. The addresses in the three internal memories are the same since it is the same tile that is targeted each time. The words are all 256 bits long as each word contains 64 pixels and one of their color components on 4 bits.

Now that the memory units are described, it is possible to establish the complete memory circuit. The complete circuit is shown in Figure 4.13. It consists of a parallel connection of four memory units.

Each color input of each memory unit is multiplexed between the four color inputs of the same color. The four inputs of these multiplexers are in fact the colors of each of the tiles used (tile 0, tile 1, tile 2 and tile 3). Then, each of the outputs of the memory are multiplexed from the values in memory in the different memory units. All these multiplexers ensure that the tiles are correctly routed, as explained above. These multiplexers are controlled by five signals: sw0, sw1, sw2, sw3 and swb which are respectively linked to tile 0, tile 1, tile 2, tile 3 and tile b. The determination of the values of these switches as well as those of the addresses of the memory units are not displayed in the circuit for simplicity. But their value are simply based on the selected tile location, Tile 0. Depending on its location, it belongs to one of the four tile groups. The groups of the neighbor tiles are also computed based on the locations of these tiles and their respective switches are set according to the group of the tile.

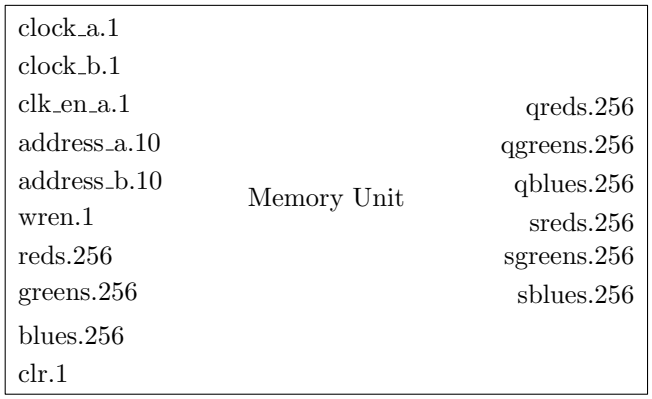

Figure 4.11: Memory Unit.

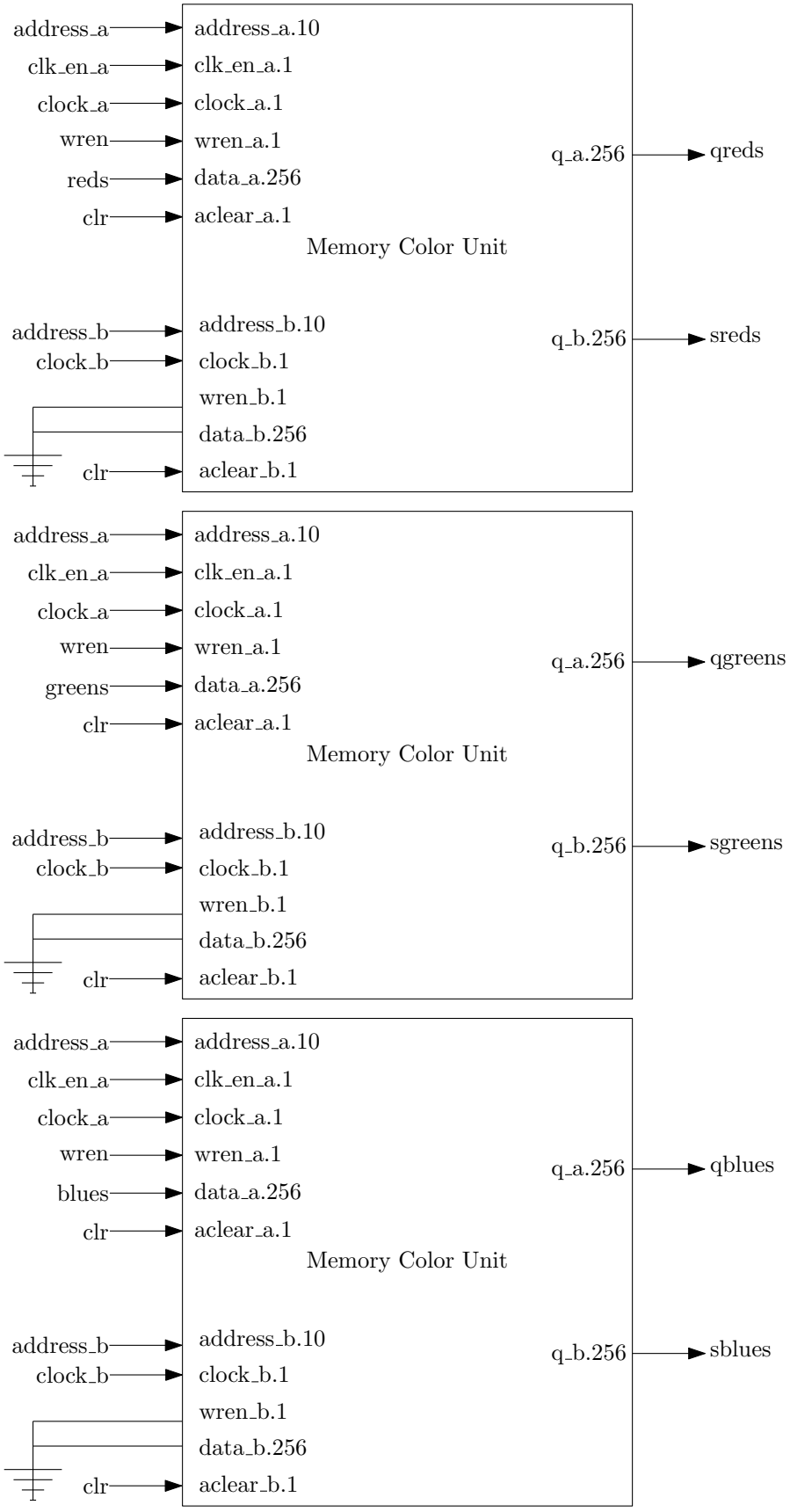

Figure 4.12: Memory Unit internal circuit.

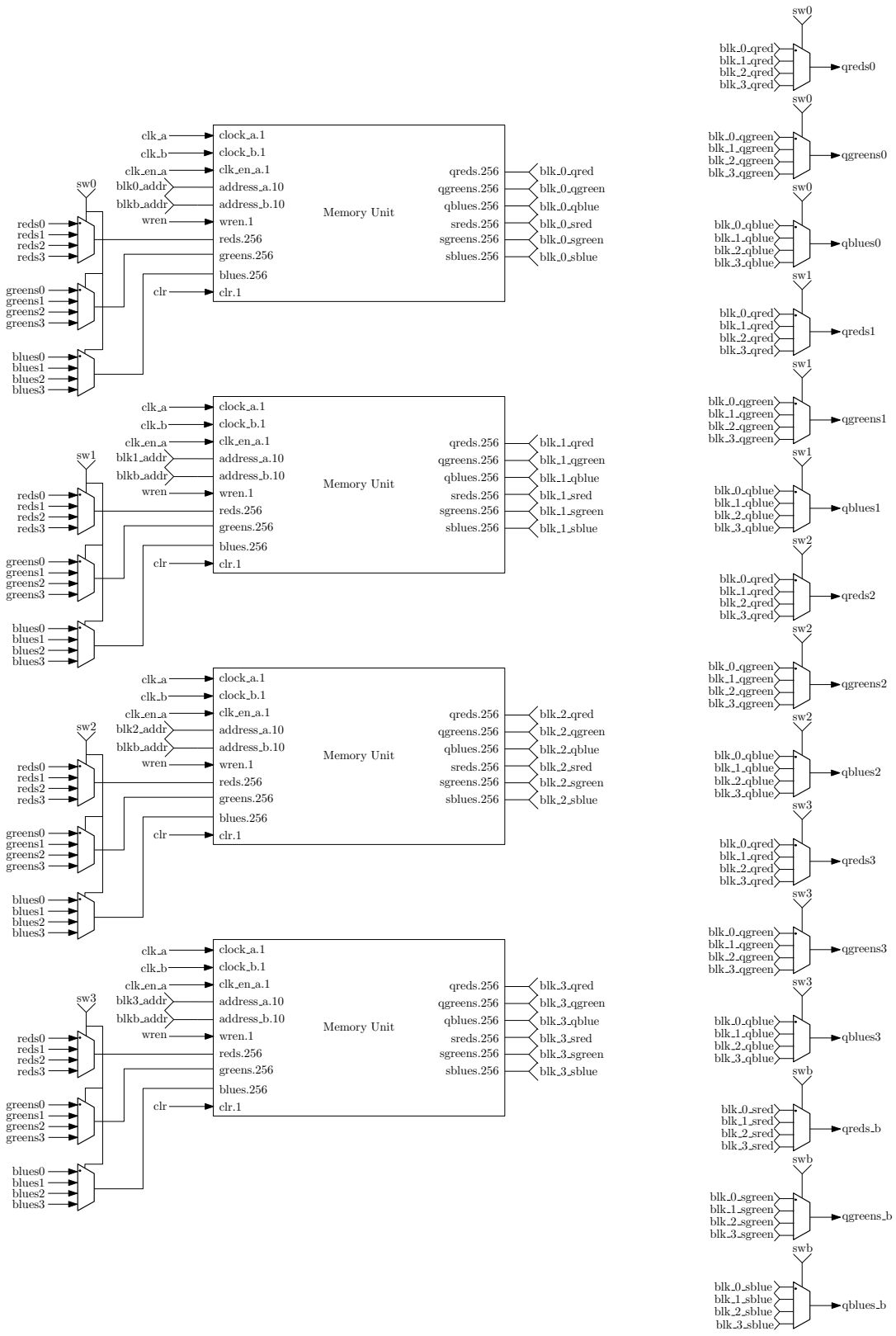

Figure 4.13: Memory internal circuit.

The interface of the memory can be found in Figure 4.14. The red, green and blue signals of each tile are put together in a bus to simplify the representation of the module.

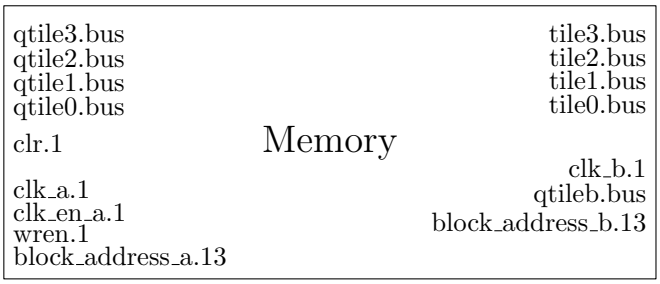

Figure 4.14: Memory.

4.2.2 Mask memory

The mask memory is identical to the CPU instruction memory except that the Altsyncram used contains only 256 words of 128 bits. This allows to store 256 masks. The circuit is given in Figure 4.15 and 4.16 provides its interface. Note that if the address is equal to 255, then the mask sent to the data_read output is a constant mask (named clear mask) which clears all pixels.

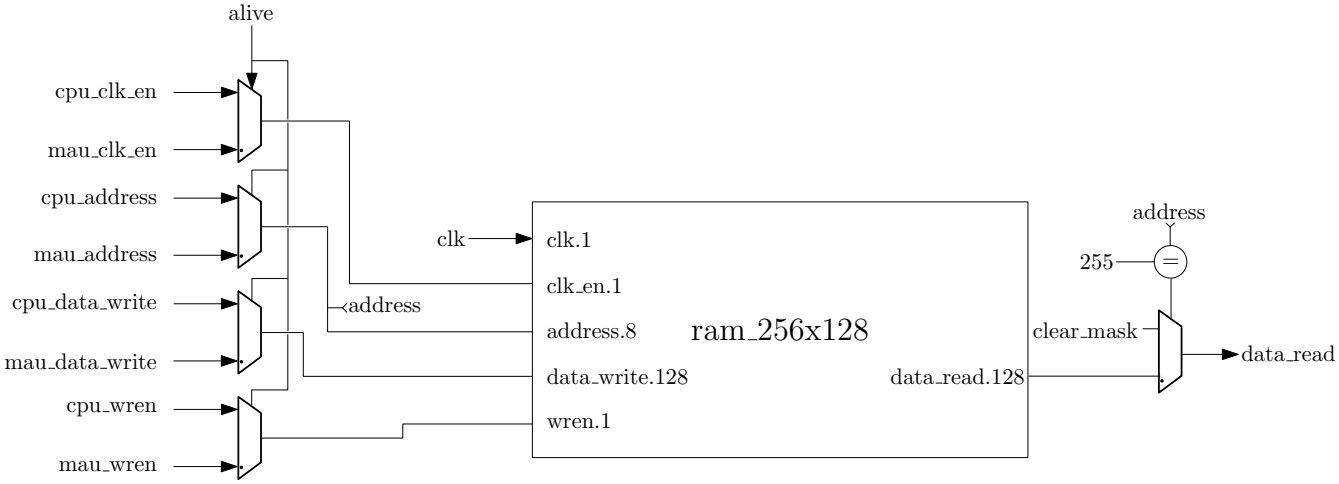

Figure 4.15: Mask memory internal circuit.

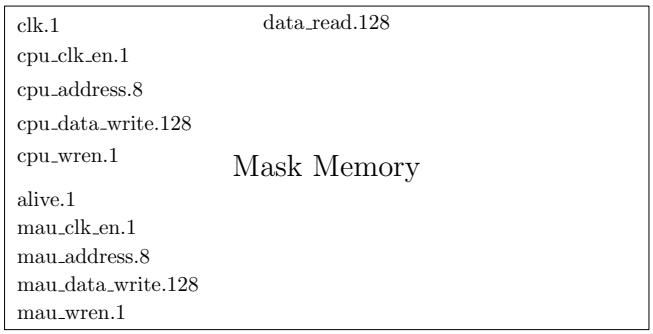

Figure 4.16: Mask memory.

4.2.3 Shifter

The objective of this module is to replicate the mask four times and to shift it in each of these copies so that it corresponds to the mask offset given by the user. After that, each of the copies can be associated with one of the four tiles loaded out of memory. This association is done in another module.

Shifter Unit

This module is responsible for shifting. To do this it simply takes the signed offsets given in input and shifts the bits of the mask. The offsets being given in pixels and the masks having two bits per pixel, it is necessary to multiply by two each offset. To achieve an horizontal offset, each line of the mask is shifted $2 \times$ offset_x while for a vertical offset, the whole mask is shifted of $16 \times$ offset_y given that there are 8 pixels by line. The circuit and the interface of the shifter unit are respectively given in Figures 4.17 and 4.18.

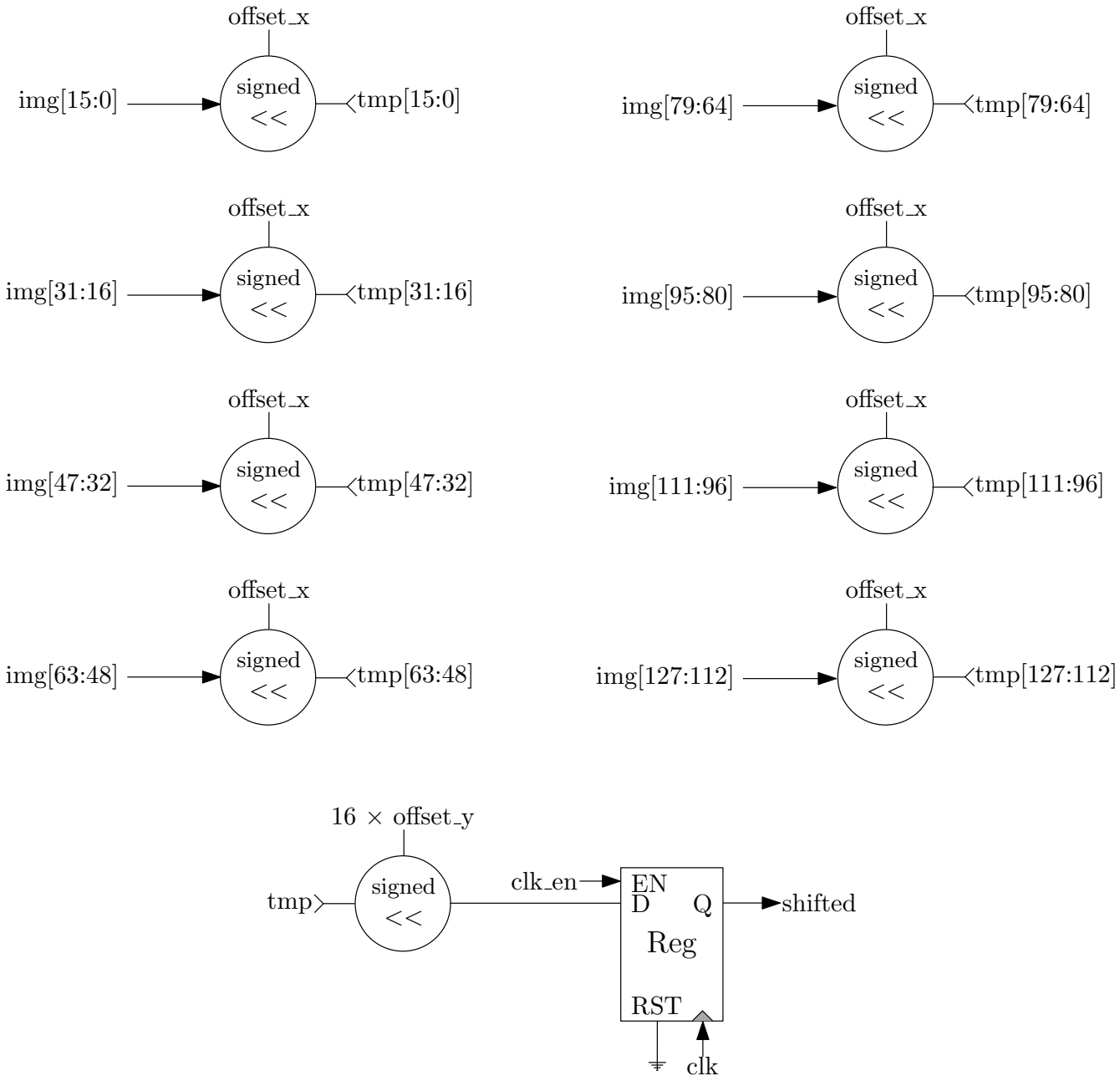

Figure 4.17: Shifter unit internal circuit.

Figure 4.18: Shifter unit.

The complete shifter is just a simple parallelization of four shifters but one has to choose the right offsets for each shifter unit. In Figure 4.19 are shown the different offsets needed to obtain the desired masks. Each offset is taken with respect to the upper left corner of the mask to which

it is associated. off_0 is of course equal to (off_x, off_y) since the offset is always given from the selected tile and the selected tile is identified by id 0. For the others, the offsets are off_1 = (off_x - 8, off_y), off_2 = (off_x, off_y - 8) and off_3 = (off_x - 8, off_y - 8).

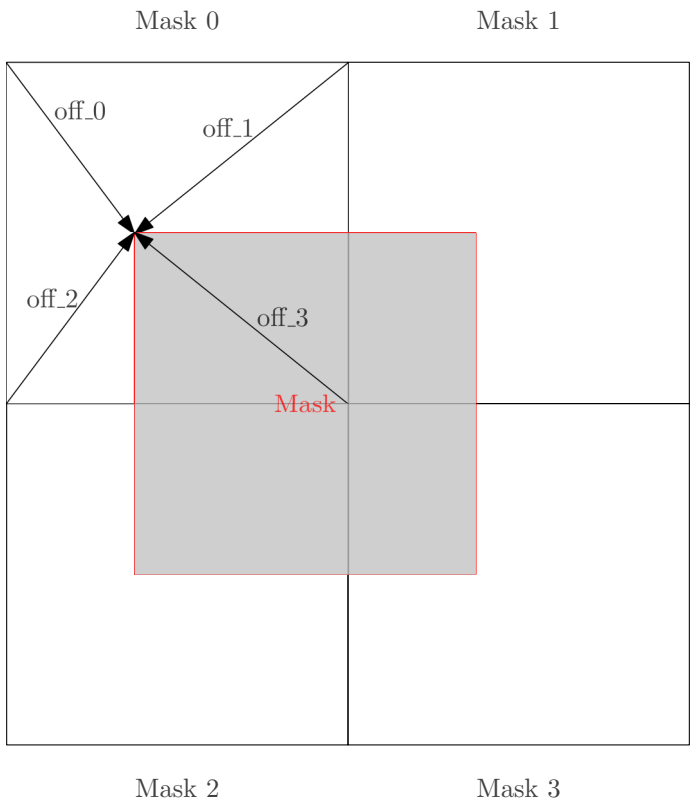

Figure 4.19: Masks offsets.

The complete shifter circuit can then be established and is given in Figure 4.20. Figure 4.21 contains the shifter interface.

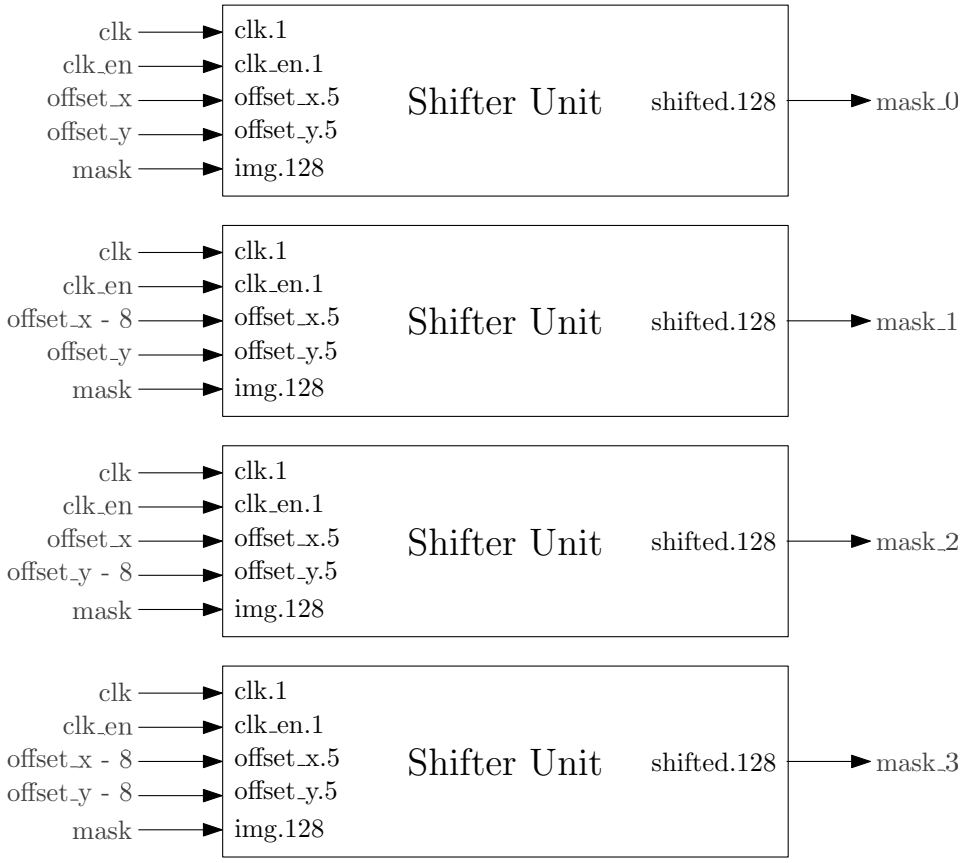

Figure 4.20: Shifter internal circuit.

Figure 4.21: Shifter.

4.2.4 Mask Logic Unit (MLU)

The MLU is simply used to modify the current value of a pixel in a tile according to the operation described for this pixel in the corresponding mask. This module is a kind of giant multiplexer. In fact, it contains a multiplexer for each primary color of each pixel and will select an input (the current value of the pixel, the primary color, the secondary color or the black color) according to the value of the mask at that position. Its operation is very simple to understand and its circuit impossible to draw on a page of this report, so the circuit is not detailed here. The interface is however given in Figure 4.22. Note that this circuit is completely combinatorial. Concerning the tile buses, they are simply made up of the three color signals of the corresponding tile.

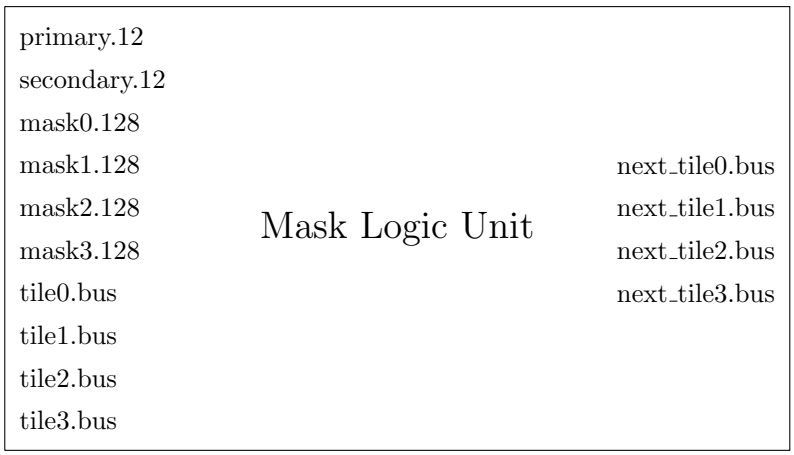

Figure 4.22: Mask Logic Unit.

4.2.5 Graphic Counter (GC)

The graphics counter is the first element fully included in the HDMI controller part of the GPU. It simply counts and provides several counts. First, it provides the current position of the pixel taken into account, so cnt_h and cnt_v representing the horizontal and vertical position of the pixel. These go through all the virtual screen of the VESA, that is to say from 0 to 1087 for cnt_h and from 0 to 516 for cnt_v. These counts are then used to correctly generate the synchronization signals of the VESA in another module.

Then, it provides four other counters blk_x, blk_y, off_x, off_y which again give the position of a pixel but using this time the GPU coordinate system. It should be noted that these counters only count from the start of the useful screen. In other words, blk_x and blk_y are both at 0 when cnt_h and cnt_v correspond to the location of the first pixel of the first tile in memory, the one at the top left corner of the useful screen. And their values continue to increase until the end of the VESA virtual screen.

Notice also that the useful screen does not correspond to the entire active area of the VESA. As mentioned earlier, not all the space is used because of memory constraints. The useful screen is centered in this active region. When the counter is in the useful region of the screen, the in_mem signal is high to warn that the position given by the counter corresponds to a tile present in memory. The range of the counters and a schematic of the positioning of the useful region are given in Figure 4.23.

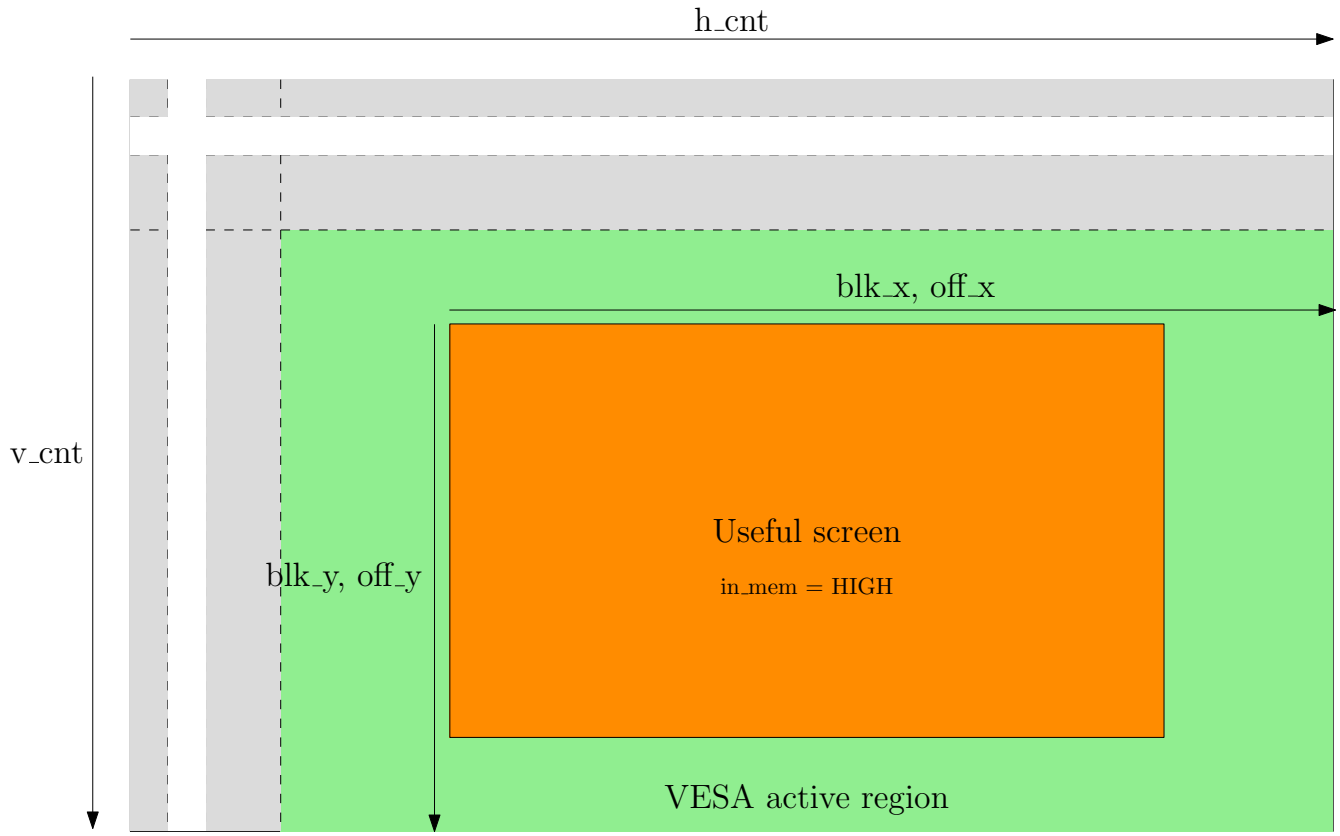

Figure 4.23: Counters range and useful screen.

All these counters are implemented with the help of registers by choosing correctly the clk_enable signals and the reset signals. The inner circuit without the in_memory which is only a condition on the values of h_cnt and v_cnt is given in Figure 4.25. The interface is also shown in Figure 4.24.

Figure 4.24: Graphic counter.

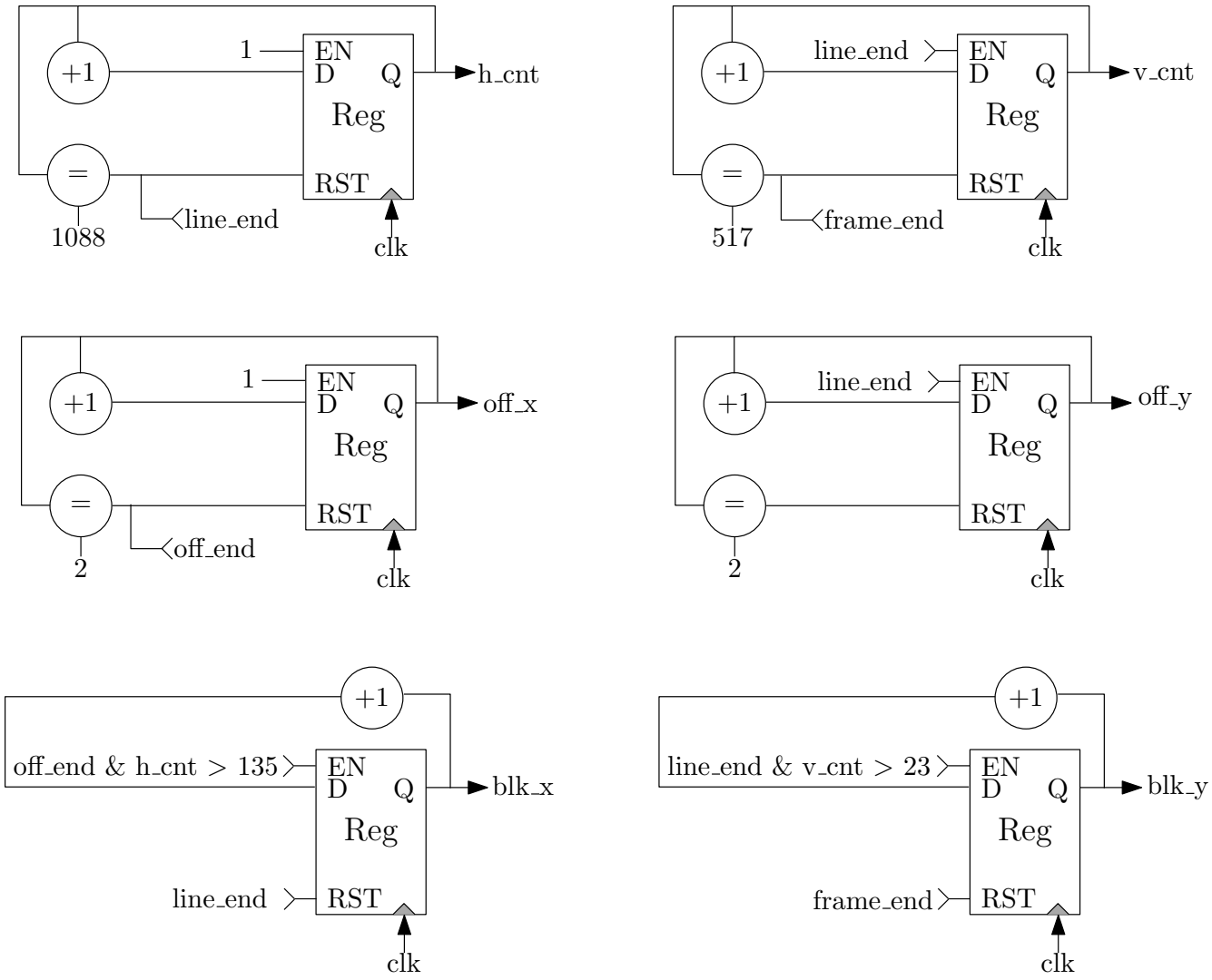

Figure 4.25: Graphic counter internal circuit.

4.2.6 Synchronizer

The synchronizer is in charge of generating the synchronization and display enable signals of the VESA protocol. It simply looks at the values of the h_cnt and v_cnt signals provided by the graphic counter. If the values are within the ranges associated with the synchronizations or display enable, the synchronizer sets the signal to high. As a reminder, the ranges were described in Figure 4.8 and Table 4.2. As the internal circuit is only a condition, it is not shown. The interface is shown in Figure 4.26.

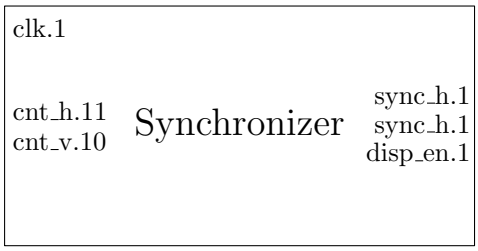

Figure 4.26: Synchronizer.

4.2.7 I2C HDMI Config

This module is provided by Terasic to configure the HDMI controller present on the DE10 nano board. In this work, it is used as a library, i.e., its description is not detailed here since it has been used as a black box. In the GPU, it is simply connected to the pins of the physical HDMI controller as Terasic suggests in its tutorial. The interface is provided in Figure 4.27.

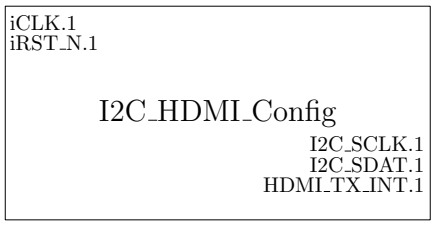

Figure 4.27: I2C HDMI Config.

4.2.8 Complete circuit

The GPU whose internal circuit is given in Figure 4.28, like the CPU, runs in sequence. It starts by reading the mask in the mask memory. Then it reads the tiles to be edited in the memory and shifts the masks at the same time. The results of these two operations arrive in the mask logic unit where they are processed in a combinatorial way. Each mask operates on the tile associated to it, taking into account the primary and secondary colors. Once this is done, the memory is enabled again, in writing mode this time.

Similarly to some CPU memories, the mask memory has a MAU access that allows reading and writing of this memory from the ARM processor.

This constitutes the computational part of the GPU, the one that the CPU controls through its ST and LD operations.

Then, there is the controller that fetches each tile in memory one by one and draws all the pixels on the physical screen. It also manages the synchronization signals of the VESA protocol. The graphic counter is used to provide the address of the memory. To do this, the blk_x and blk_y outputs are concatenated to form the address. Then, the offsets are added together to form an index in the tile and select the current pixel. The selected pixel is then put on the output if the signal in_memory of the graphic counter is high, otherwise the color is black. It is to this effect that the two multiplexers are used on the hdmi_tx_d output. It should be noted that the index

and the im_mem signal must be delayed so that they arrive at the multiplexers at the same time as the tile requested from the memory. This is why two registers are left on their way. The cnt_h and cnt_v signals go to the synchronizer which is responsible for generating the synchronization signals. And finally, there is also the I2C_HDMI_Config to configure the physical HDMI controller.

It should be noted that two clocks are present. On the computational side, it is the same clock as the CPU that is used to ensure that the two modules are synchronized. On the HDMI side, it is the VESA protocol clock that is used. It is called gpu_clk. The GPU interface is available in Figure 4.29.

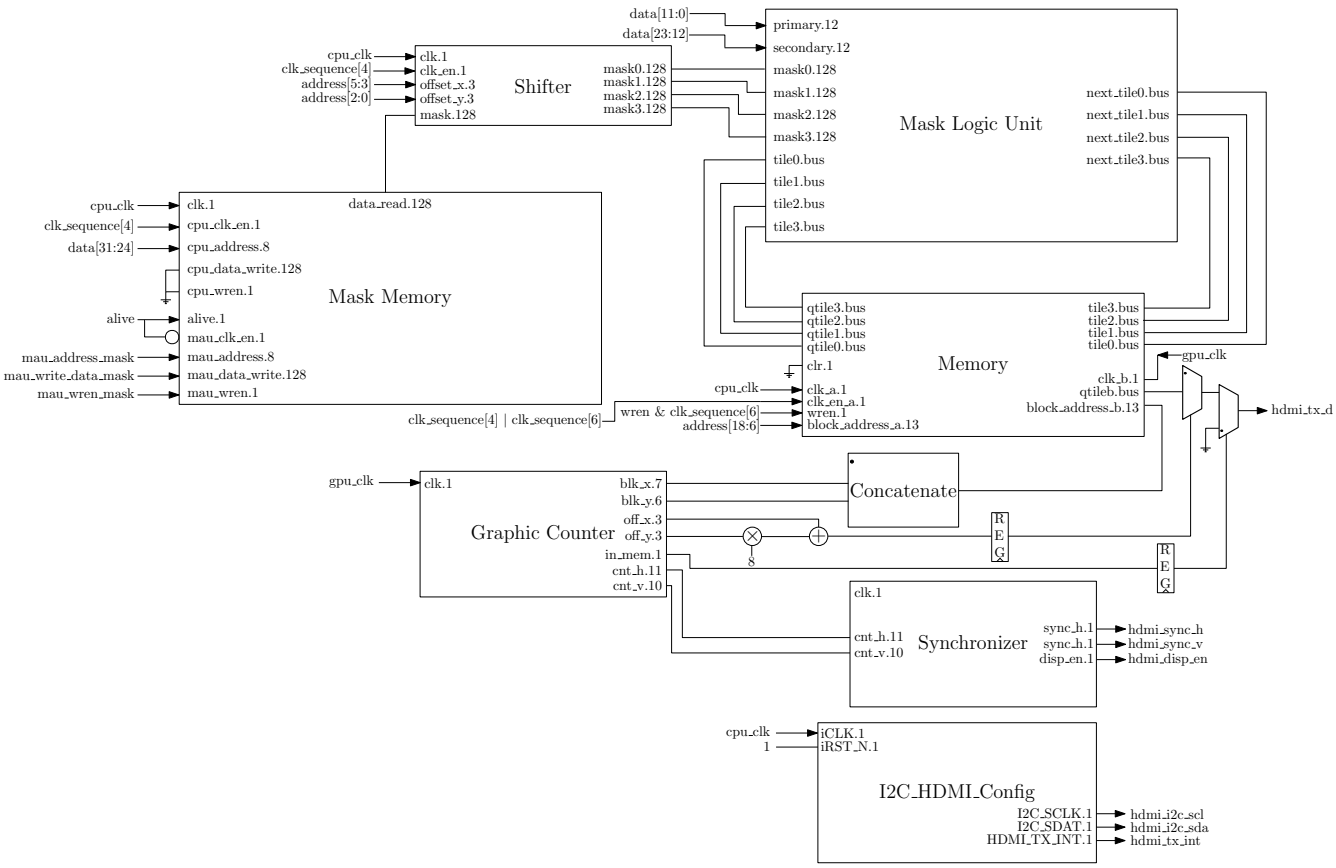

Figure 4.28: GPU internal circuit.

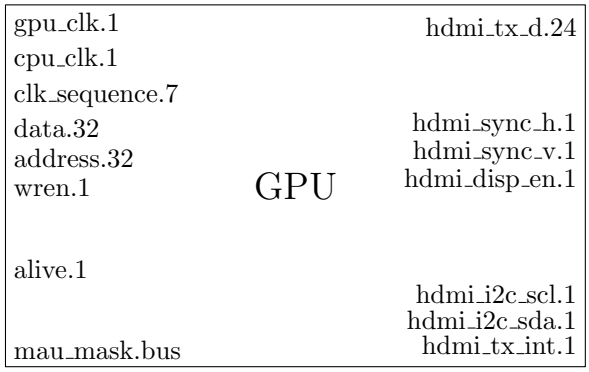

Figure 4.29: GPU.

4.3 Clock Unit (CLKU)

Github link to the verilog files of the CTRLU.

As seen previously, two clocks are needed. One as high as possible for the CPU and the modules working synchronously with it and one for the 33.750MHz HDMI controller. The CPU clock cannot exceed 50MHz. Indeed, other higher frequencies have been tested but do not meet the timing constraints at compile time. For the CPU, the clock unit only passes the FPGA 50MHz physical clock from the input to the output (the CPU clock circuit is a simple wire). For the GPU clock, the frequency has to be modified. This is done using a PLL (Phased-locked loop) which allows to generate a clock with a given frequency from another clock with a fixed frequency. This PLL is easily configured in a GUI using the Quartus Mega Wizard by setting the input and output frequencies. The CLKU then consists in a PLL. Its interface is given in Figure 4.30.

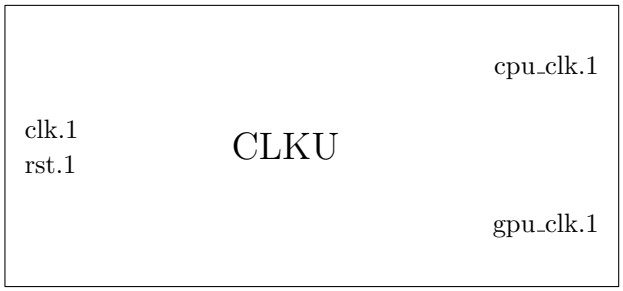

Figure 4.30: CLKU.

Chapter 5

Memory Access Unit and Control Unit design

This section describes the Memory Access Unit, which gives the ARM part read and write access to several memories of the system. Another super-unit, the Control Unit is also described here. The latter keeps the state of the machine (on or off) and gives access to this register to the ARM part. The ARM processor can therefore switch the machine on and off as it wishes from this access. Before introducing these two super-units, the Avalon protocol is first described. This protocol allows the communications through the bridges existing between the ARM and FPGA parts of the Cyclone V.

5.1 Communication between ARM and FPGA sides

The communication between the two parts takes place via one of the previously mentioned bridges using the Avalon protocol [6] described by Intel. This protocol allows high-speed communication. The Avalon interface has different sub-interfaces for different types of communication. Each type is adapted to certain applications. In this project where the protocol is used to communicate between master and slave devices, the Avalon MM is used. The Avalon MM protocol is address-based which is very suitable for the needs of this project. We here describe the read and write operations of this protocol.

5.1.1 Read operation

The sequence diagram of the protocol for the reading operation is visible in Figure 5.1. In the diagram, two masters are represented but only the unique master case is described. In the first place, the master simply has to give the start address to which it wants to read, the length of the burst (the number of words desired). It also has to switch the read and beginbursttransfer signals to high. The slave then responds to this with a waitrequest that goes up. The master must imperatively keep all signals unchanged — except beginbursttransfer which returns to the low state — during the waitrequest to give the slave time to capture them. Once ready, the slave removes the waitrequest and informs that data is available by switching readdatavalid to high and sending the first word. The other words are then sent to the master in the order of their addresses.

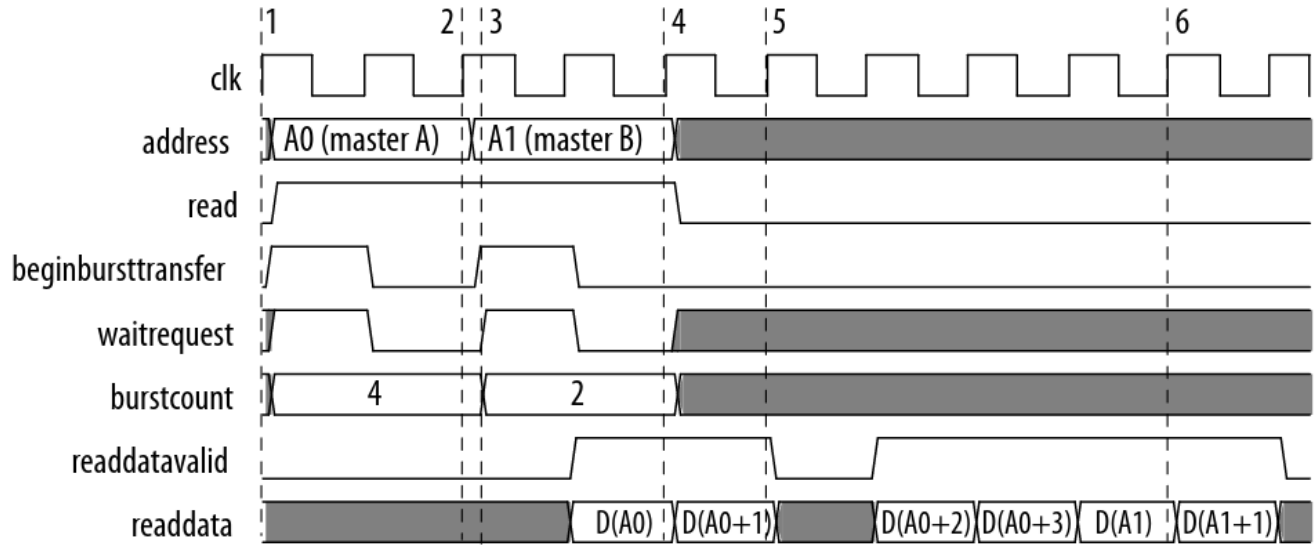

Figure 5.1: Read operation using Avalon MM [6].

5.1.2 Write operation

For writing, it's even simpler. As can be seen in Figure 5.2, the beginning is similar to reading. The master has to provide the first address, the length of the burst and also to switch beginbursttransfer to high. In addition to that, the first word is also provided. Here again, the slave responds with a waitrequest. As soon as the waitrequest is switched off, the master sets the beginbursttransfer to low and sends the words one after the other, starting at the next clock cycle. If the write signal goes down, the burst is paused and resumes as soon as it goes up again.

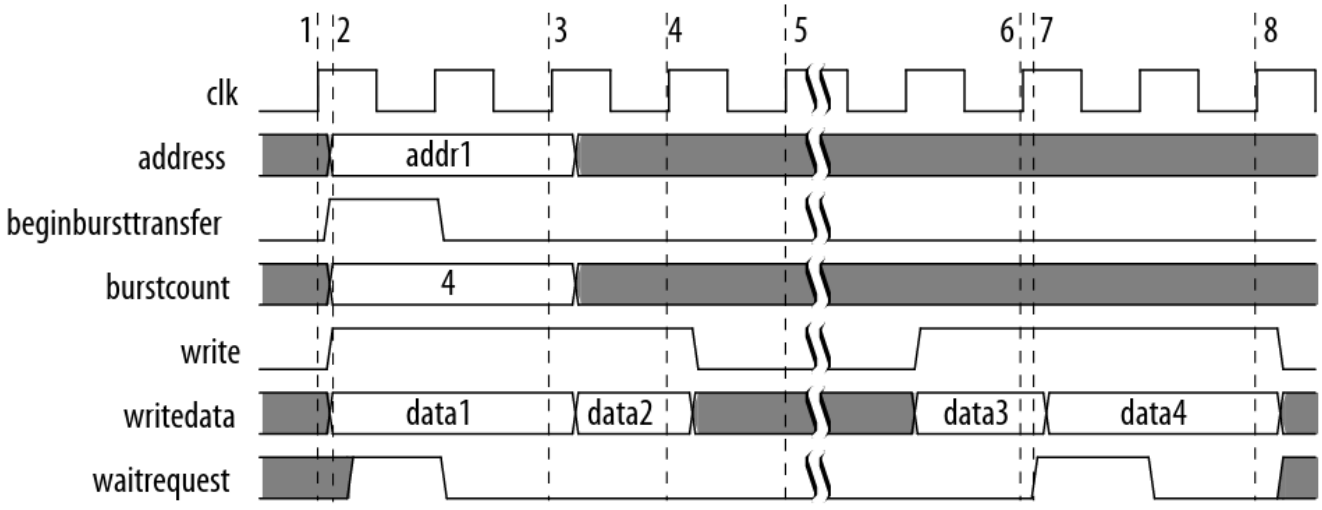

Figure 5.2: Write operation using Avalon MM [6].

This protocol is not very complicated. However, it has a lot of signals and is not necessarily practical to use directly in user-developed slaves and masters. This is why interface modules are used. These modules are described in the sections where they are needed.

5.2 Memory Access Unit (MAU)

Github link to the verilog files of the MAU.

Like previously mentioned several times the MAU allows access to several memories of the machine. In order to simplify the protocol in the machine, a module called Avalon to External Bus Bridge is used. This module manages the Avalon protocol and exposes an interface (the external bus) using a simpler protocol. This module is used as many times as there are memories to connect to the interconnect. The various signals used by this module are shown in Figure 5.3. Note that this module has an integrated timer that is reset each time the Avalon to external bus bridge receives an acknowledgement from the external slave. When this timer reaches 0, the request from the master (denoted by Avalon Switch Fabric in Figure 5.3) is deleted.

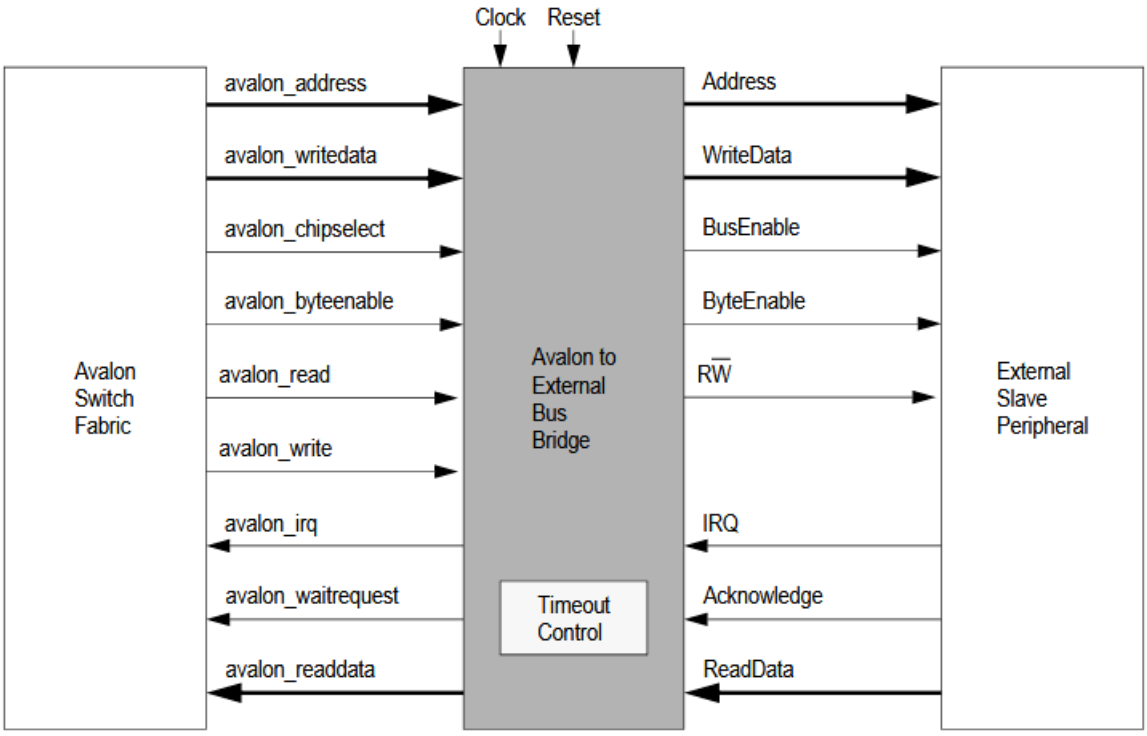

Figure 5.3: Avalon to external bus bridge interface and signals [7].

Compared to the Avalon protocol, this protocol is considered simpler because it offers an interface closer to that used by the memories. Indeed, the usual signals address, write data, read data and R$\overline{W}$ are present. Moreover, this protocol is very simple to use. Indeed, as can be seen in Figure 5.4. The master places all the necessary elements on the bus and then switches the bus enable signal to high. This means that the slave can act. The slave then does what it has to do. Either take the data and write it to the address given by the master or send the requested data that is present at the address given by the master. Once the slave has finished its job, it switches the acknowledgement signal to high during a clock cycle to validate the transaction.

It is possible to configure the address range and the word size used by these bus. For this work, it is decided that the addressable space is 256 Kbytes long and that the words are on 32 bits for each access except for the access to the GPU mask memory which is on 4 Kbytes with 128 bits words. Each of these accesses are put on the HPS-FPGA bridge, their offset is given in Table

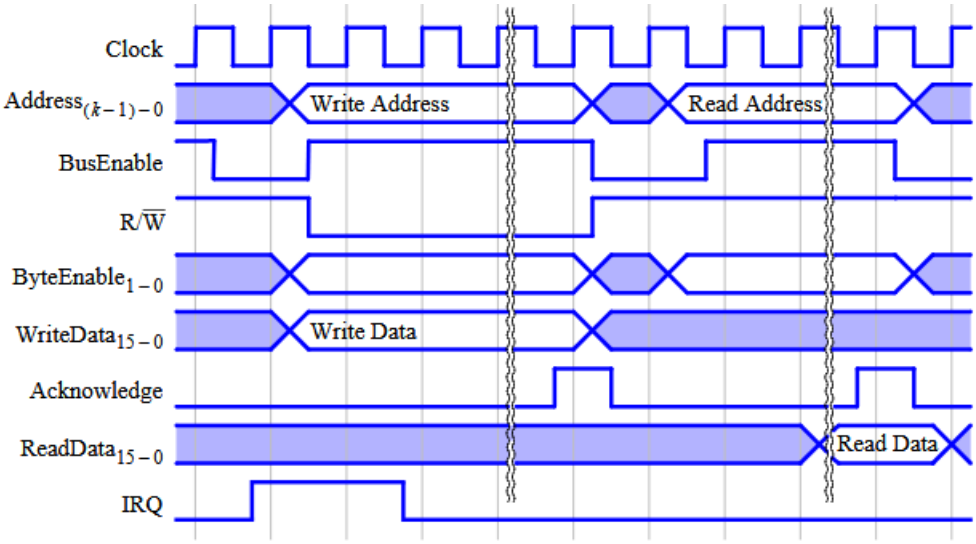

Figure 5.4: External bus protocol timings [7].

5.1. These dimensions were chosen to make the offset easy to remember. Indeed, only the sixth number changes in the hexadecimal address. Moreover, this gives future users a large margin if they want to change the dimensions of the memories. As far as the mask memory is concerned, the address range is simply the size of the memory. This has been done especially to discourage any modification at this level. Indeed, this would lead to a large modification of the GPU which could degrade its performance or make it impossible to include in the design (due to lack of space on the FPGA). More over, the size of this memory is already the maximum addressable size possible for the store operation, using the GPU format.

Access	Base	End	Address range	Word size
Instruction Memory (CPU)	0x0000 0000	0x0003 FFFF	256kB	32
Data Memory (CPU)	0x0010 0000	0x0013 FFFF	256kB	32
Register File (CPU)	0x0020 0000	0x0023 FFFF	256kB	32
I/Os (IOU)	0x0030 0000	0x0033 FFFF	256kB	32
Mask Memory (GPU)	0x0040 0000	0x0040 0FFF	4096B	128

Table 5.1: Memory bus description.

The connections in QSys are shown in Figure 5.5. The different modules and the HPS are highlighted in yellow. The different Avalon slaves and the master are shown in red. It can be noticed by following the connection highlighted in blue that it is indeed the HPS-FPGA bridge that is used (h2f_axi_master in Qsys) and that the offsets of the slaves are indeed those previously given. The signals corresponding to the buses are displayed in green. Each of them is exported, that is to say that they can be used from the FPGA. They are the ones that are connected to the different memories.

Now that the descriptions of the external bus bridge and its implementation are done, the design of the MAU can be considered.

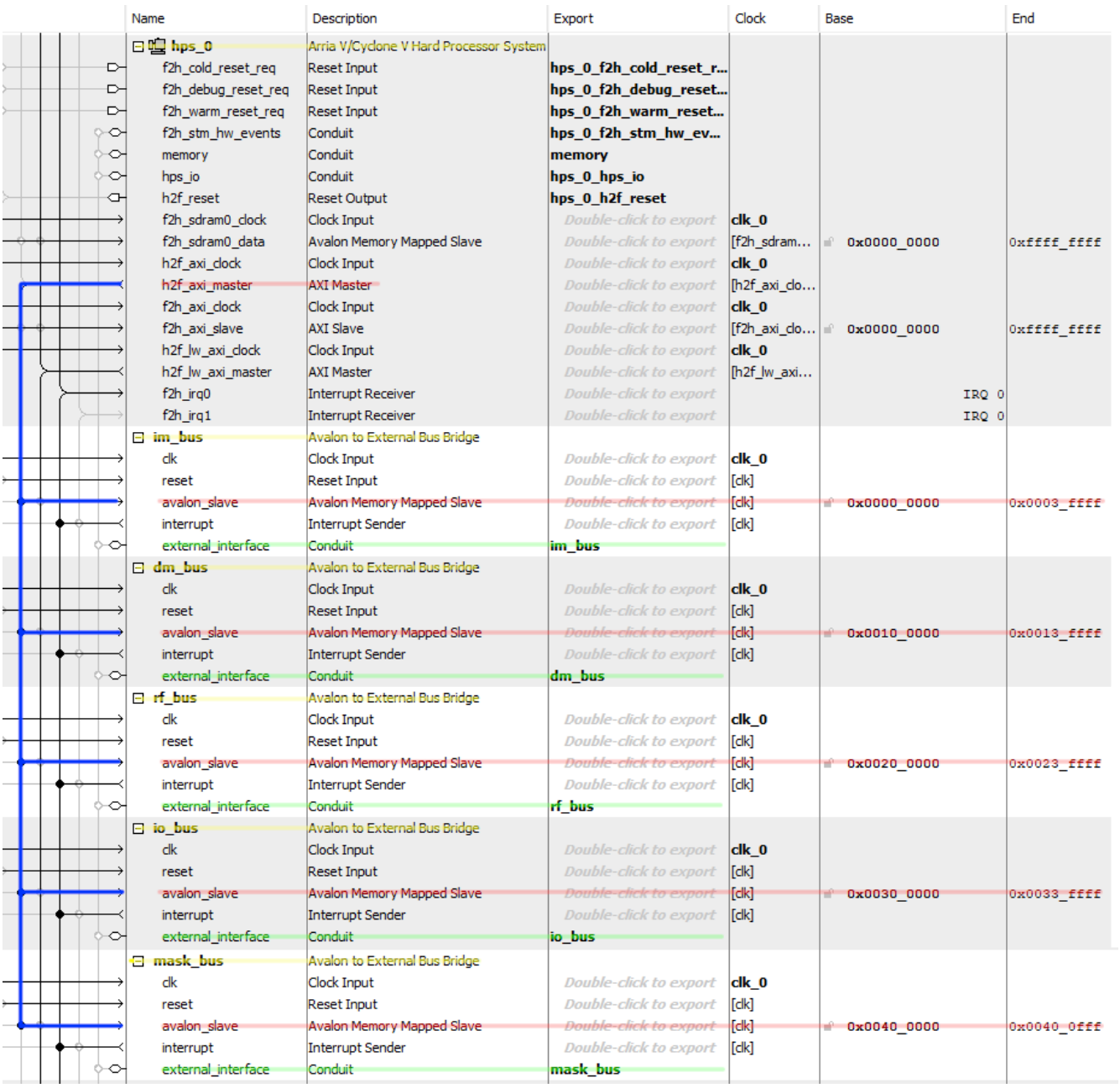

Figure 5.5: Avalon to external bus bridges connections in Qsys.

5.2.1 Memory Access

In fact, the Memory Access Unit is composed of only two types of modules. The Memory Access 32, and the Memory Access 128 both allowing the interfacing between an external bus and a memory (these two modules respectively manage words of 32 and 128 bits). To do this, they both implement a finite state machine. That of Memory Access 32 is given in Figure 5.6. The only difference that the 128 bits version has is that the address is not multiplied by 4 in the Idle state.

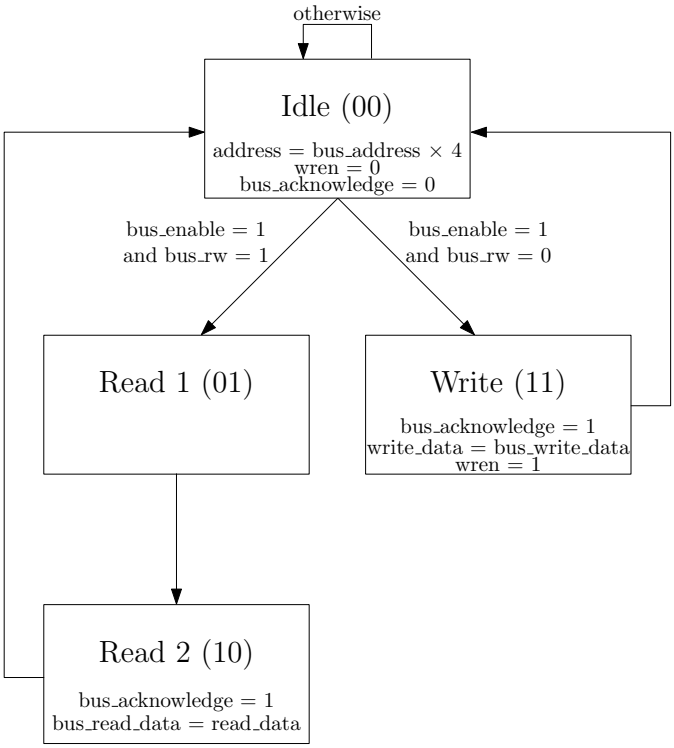

Figure 5.6: Memory Access 32 finite state machine.

This state machine simply follows the protocol described earlier. It only acts when the bus is high and sends an acknowledge during a clock cycle once the request is executed (the finite state machine is reevaluated at each clock cycle). On the read side, a state seems to be useless (Read 1). This one is in fact very important, it gives the memory time to fetch the value present at the requested address. The interfaces of the memory access are given in Figures 5.7 and 5.8.

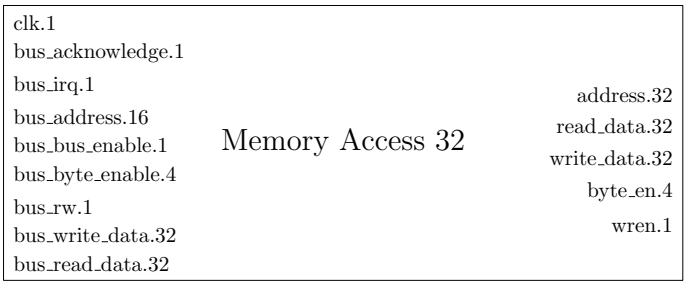

Figure 5.7: Memory Access 32.

5.2.2 Memory Access Unit circuit

The Memory Access Unit is therefore a simple parallelization of four Memory Access 32 modules and one Memory Access 128 module. The MAU is later connected to the different signals exported in QSys. The circuit of the MAU is given in Figure 5.9 and its interface is shown in Figure 5.10. In order to simplify the interface of the MAU, not all signals are put in it and a bus representation is used. However, all signals are shown in the internal circuit.

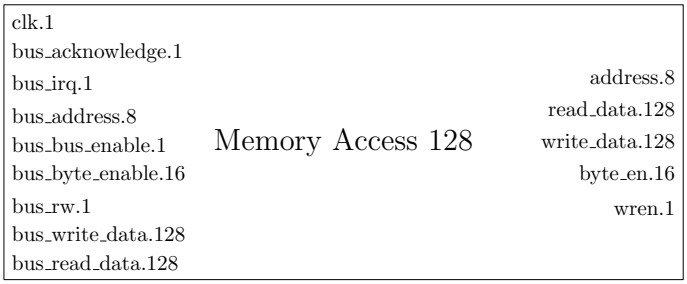

Figure 5.8: Memory Access 128.

5.3 Control Unit (CTRLU)

Github link to the verilog files of the CTRLU.

The control unit is a very simple unit. It manages the state of the machine. This means that it decides whether the machine is switched on or off. Initially, the machine is switched off (alive is in the low state). When alive is low, the memories of the machine can be manipulated by the ARM processor. It is thus at this moment that the machine is programmed. Once this is done, the ARM processor can start the machine by switching alive to the high state, the means by which this is achieved are seen just after. During the execution of a program, it can be stopped in two ways. First, the ARM processor can stop it by passing alive to the low state. Second, the beta machine itself can order the stop by executing the EXIT instruction which changes the halt signal to high. The control unit then responds to this signal by switching alive to low.

As for the MAU, it is necessary to connect a module to the interconnect to make this unit accessible from the ARM processor. Once again, the Avalon protocol is not used directly. Instead, a module called Parallel IO (PIO) is used. This one gives direct access to a register from the interconnect (and thus from the ARM processor). This module also uses Avalon MM, so the register has an address from the point of view of the ARM processor. A difference with the Avalon to external bus bridge of the MAU is that the PIO is put on the lightweight HPS-FPGA bridge. This choice is made because the PIO is not frequently used and does not require large data transfers. For this kind of modules, the HPS documentation advises to use the lightweight bridge. The connection of the parallel IO in QSys is shown in Figure 5.11. The usual color code is used.

One of the ports is exported. In fact this one exposes two signals. One input and one output. The output allows to read what the ARM processor proposes as new value of the register while the input allows to set the value of the register. A module must therefore be added between the input and the output in order to listen to the ARM processor port and the beta machine port and to set the register accordingly. This module is in fact the control unit itself.

5.3.1 Control Unit circuit

The Control Unit, like the memory access, simply implements a finite state machine. This one is described in Figure 5.12. The hps_cmd signal represents the signal coming from the PIO (thus from the HPS) and the halt is the stop signal of the beta machine. According to these signals, the value of alive is fixed. The on/off transitions initiated by the ARM processor are done in two steps. A transition is completed when the ARM processor sends a 1 followed by a 0. This has been

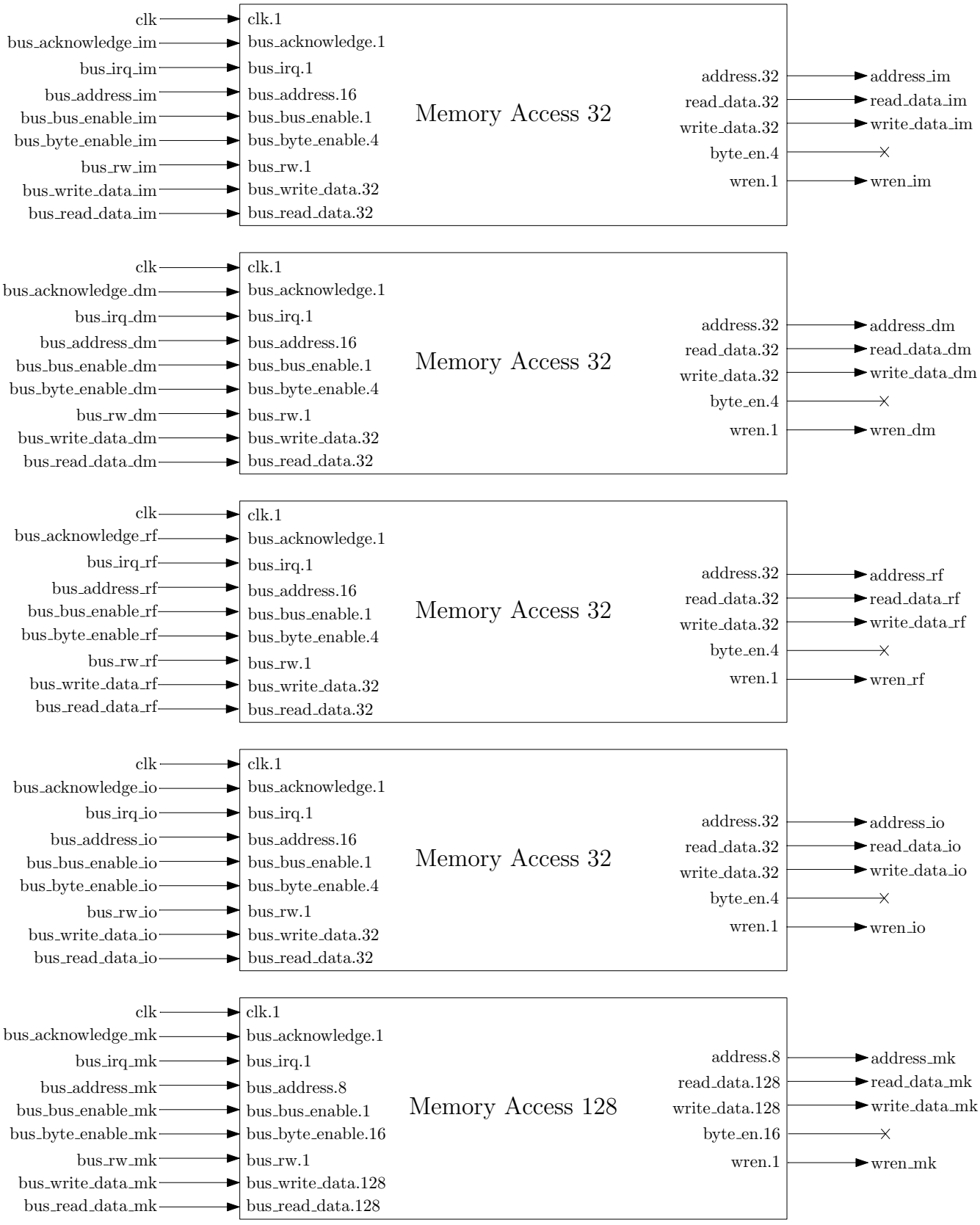

Figure 5.9: Memory Access Unit internal circuit.

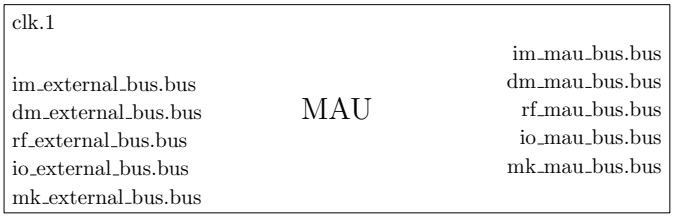

Figure 5.10: Memory Access Unit.

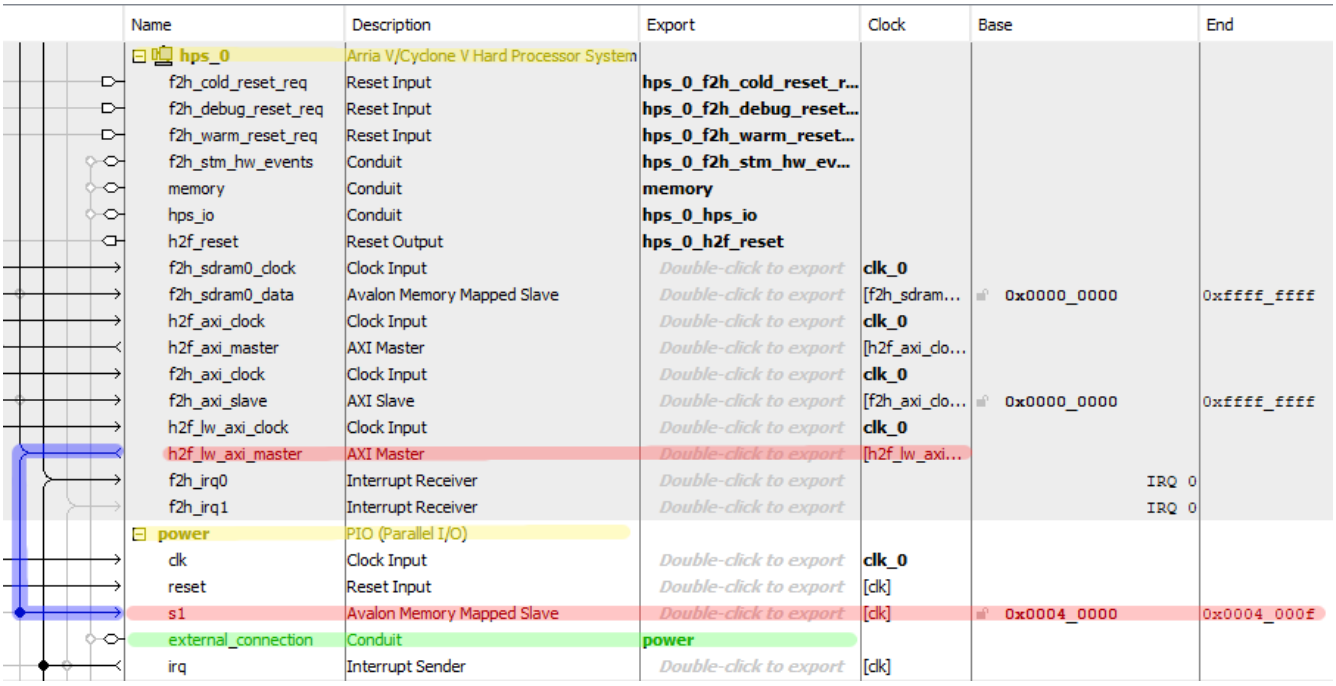

Figure 5.11: Parallel IO connections in QSys.

done as so to have only one bit for the command. If the change of state was done directly when a 1 appeared, the final state would have been uncertain. Indeed, it is impossible to know for how many clock cycles the 1 is maintained by the ARM. Depending on the number of cycles it would take to reset its signal to 0, the state would switch several times between on and off. With the intermediate state, any bounce between states is avoided. Furthermore, simply using 1 for alive and 0 for stop is not a good idea either since the ARM processor would have to always listen to state changes coming from the beta machine to ensure that theirs values are coherent.

The control unit then exposes its state and the alive signal. The two bits of the state are passed into a logic AND gate whose two inputs are inverted and the output of this gate is used to set the PIO register (the corresponding circuit is shown when the entire system is connected later in this report). The PIO register is therefore high when the machine is stopped and low at any other time. This is done because when the register goes from low to high, an interrupt is sent to the ARM processor. Since it is interesting to receive the interrupt when the machine is shut down, it has been done this way. However, this interrupt is not used in this work but it could trigger an interaction between the ARM processor and beta machine. A visual summary is displayed in Figure 5.13 and the interface of the Control Unit is therefore the one given in Figure 5.14.

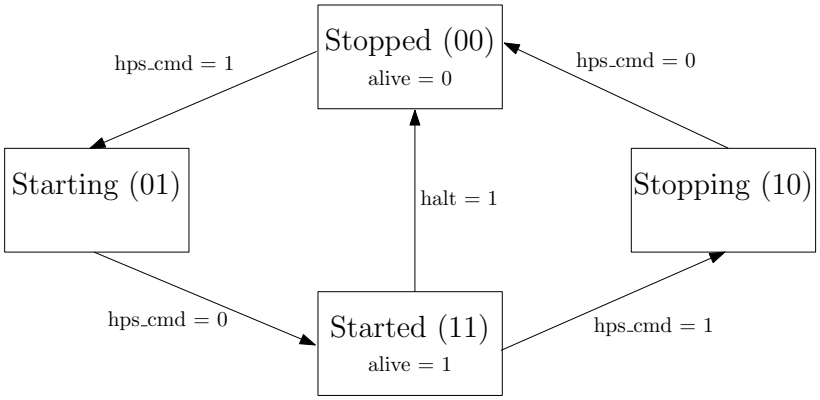

Figure 5.12: Finite state machine of the Control Unit.

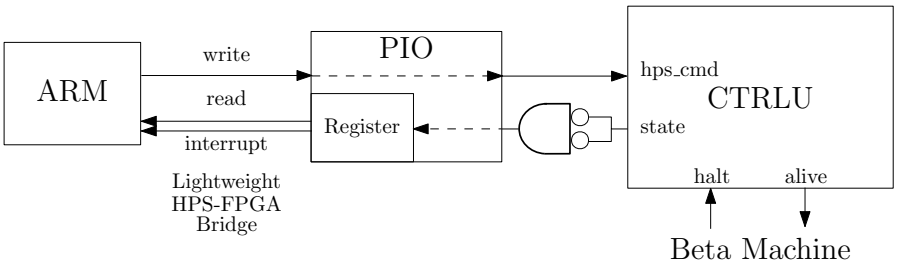

Figure 5.13: ARM, Beta Machine and CTRLU interaction summary.

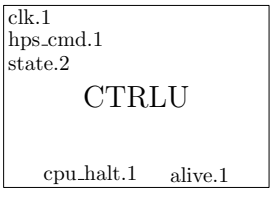

Figure 5.14: Control Unit.

Chapter 6

System design

Github link to the GHRD file.

In this chapter the final steps of the system design are described. The generation of the HPS system allowing the connection between the FPGA and the ARM processor, the interconnection of the different super units as well as the installation of the system on the FPGA and the OS on the ARM processor are detailed.

6.1 Generating the HPS module

In the previous sections, the ARM-FPGA interconnect was described using QSys. Now that all the required elements have been defined, all that remains is to generate the HPS system from QSys. This is done simply by pressing the generate button and choosing Verilog as output language in QSys. This operation generates a module that can be used in the project. In fact, the module gives access to all the signals exported to QSys (as input or output depending on the signal). In addition to that, some signals such as reset, clock, etc. are present in the module interface. Inside the HPS module are the different modules that have been added in QSys (PIO and Avalon to external bus bridge) as well as other modules necessary for the operation of the interconnection. The HPS is also connected to the different FPGA-ARM bridges inside. The interface of the HPS is given in Figure 6.1.

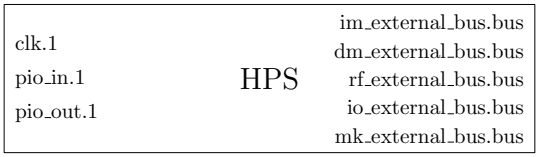

Figure 6.1: HPS system.

6.2 Golden Hardware Reference Design (GHRD) and system circuit

The GHRD has already been mentioned in this report. It is a project provided by Terasic, the manufacturer of the board used for this work, the DE10 Nano. This project includes the definition of the different inputs and outputs of the system as previously discussed. A Verilog file is also provided. This file describes a module whose inputs and outputs are redirected to physical inputs and outputs, this file is called the top-level module. It is then a question of linking all the units defined in this project within the top-level module and of correctly redirecting the IOs of the system to those of the top-level. The HDMI outputs, buttons and leds are part of the IOs of the top-level module.

The system circuit is fairly straight-forward. Indeed, all the units have already been described and their interraction briefly detailed. It is now a matter of connecting everything together. The HPS provides on the one hand the access to the PIO for the startup control to the CTRLU and on the other hand the different external buses for the CPU, GPU and the IOU. These signals are first directed to the MAU that links the external buses and the memories. Then the CLKU provides the clocks to all units, including the HPS. Finally, connections are made between the CPU, the GPU and the IOU to ensure the control of the latter two by the ST and LD instructions.

The circuit of the system is displayed in Figure 6.2. In this circuit, the inputs and outputs are the ones of the top-level module. These are thus directly connected to the physical IOs of the FPGA.

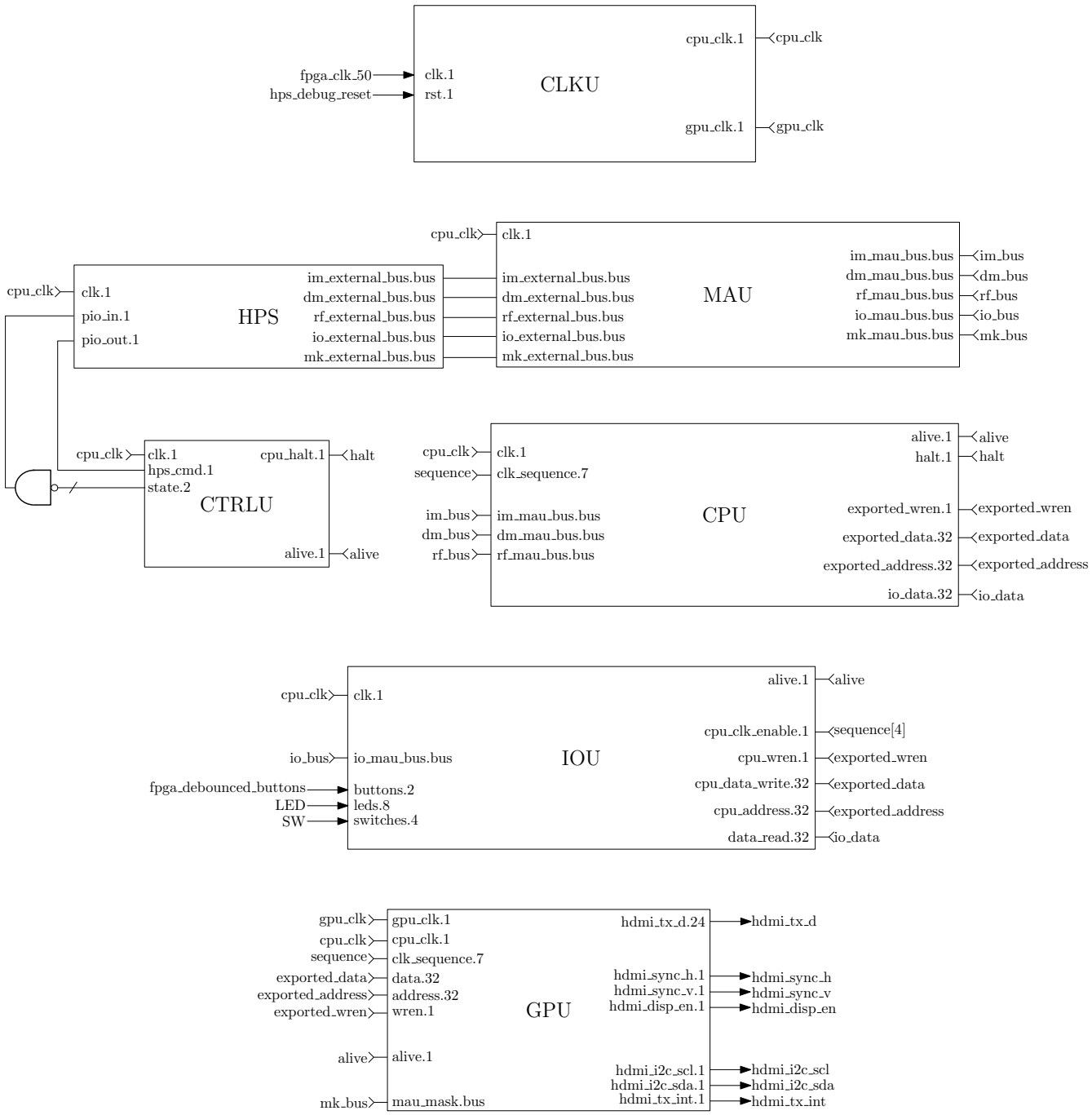

Figure 6.2: Whole system.

6.3 ARM side Operating System

The ARM processor can be used in two ways. Either in bare metal where the programming is done directly in assembly on the processor. This method has the disadvantage of not being very accessible and of being expensive. The bare metal programming tools for the processor are not free and are professional licensed softwares. The second method is to install an operating system. This can be done by flashing an SD card that serves as storage for the ARM processor. On this card

are created several partitions containing the bootloader (u-boot generally), the OS (Angström on the images of Terasic but any Linux can be adapted) and the programming file of the FPGA part. In this configuration, the bootloader configures the FPGA before launching the operating system.

The image used to flash the SD card in this work is one provided by Intel using u-boot and ubuntu. The Terasic image containing Angström was also tested but this distribution is no longer maintained, so the image was not selected for use. The big disadvantage of the Intel image is that it takes up a lot more space, Ubuntu being much larger than an embedded Linux like Angström. It could be the work of another master thesis to improve the choice of the OS. Once the OS is installed and booted, one just has to program on it to be able to interact with the system on FPGA. This is the subject of the next section.

Chapter 7

Accessivity tools and demonstrations

With the entire system designed and programmed on the FPGA, it is time to test it. But before that, one needs tools to allow to write in the memories of the machine from Linux, on the ARM processor. In addition to that, tools simplifying the development on the system are presented. After the presentation of the different tools, the various assembly demonstrations are detailed.

7.1 Tools

Github link to the files of the tools.

7.1.1 Beta assembler

The first tool was developed by Romain Mormont. It compiles assembly code using the ISA described earlier in the report into machine code for beta machine. This tool had already been created before this work, so it was decided to take advantage of it and not reinvent the wheel. Some minor modifications were made to add a binary file backup that could be used for programming the beta machine of this work, to add the exit instruction and to remove the div instruction.

7.1.2 Beta utils

Beta utils is a utility that allows access to memory and system control from the ARM processor. In fact, to do this it is enough to write and read in the space address of the bridge Avalon on which a slave is connected in QSys. To have access to a given slave, it is necessary to add its offset (specified in QSys) to the starting address of the bridge in the physical memory. The start and end addresses of each bridge are constant and specified in Figure 1.10.

To keep things simple, it was decided to implement everything in the user space of the OS. Beta utils is therefore only an application. This application does not have direct access to the physical addresses. In order for it to be able to read and write to specific locations in physical memory, a mapping must first be created between the physical memory and the virtual memory of the Beta utils process. To do this, the system call mmap is used. This creates a mapping and returns a pointer allowing access to the specified physical memory in a contiguous manner from

the virtual memory. This means that $pointer + n$ corresponds to the nth element of the mapped physical region. This solution is not the fastest but it is more than sufficient for the operations performed by Beta utils and has the advantage of being simple.

For the communication with the memories, the tool allows two operations. Both can only be done when the system on FPGA is stable, in other words when the alive signal is low. The first one is writing. This one allows to write the content of a binary file in the chosen memory, with a certain offset in it.

Program command.

```
-p, --program <memory_id> <file_path> <size> <offset>
Programs a memory with a binary file - only works if machine is off,
parameters are
    .memory_id: the id of the memory to be programmed
    .file_path: path to the binary file
    .size: number of bytes in the binary file
    .offset: offset where to start writing
```

Reading is done from a starting offset to an ending offset. The bytes are simply displayed, there is no saving to a file. The memory identifiers are IM for instruction memory, DM for data memory, RF for register file, IO for io memory and MK for mask memory.

Read command.

```
-r, --read <memory_id> <start> <end>
Reads and displays the content of a memory from byte start to byte end
    .memory_id: the id of the memory to be programmed
    .start: the first byte to read
    .end: the last byte to read
```

To switch on and off the system, two commands are provided: start and shutdown.

Start and shutdown commands.

```
-st, --start
Starts the machine

-sd, --shutdown
Stops the machine
```

7.1.3 Mask Drawer

The mask drawer is a small tool coded in Python developed for this work. It provides a graphical interface, shown in Figure 7.1, that allows editing of masks. In the window there are five buttons. Four are used to select the state to be applied: keep, primary color, secondary color and clear. Once a state has been selected, simply click on the grid to change the state of the pixel. A square on the grid represents a pixel. If the mouse is held down on the left button, all the cells through which the cursor passes have their state changed. It is also possible to simply click one cell by one

cell. The right click is a shortcut to the keep state. Once the mask is drawn, it can be saved as a binary file by clicking on generate. Note that the mask drawer takes as argument the name of the file, without its extension. If this file already exists, it is opened and its edition can be resumed. If it does not exist, it is created.

Figure 7.1: Mask drawer

7.2 Assembly demonstrations

Assembly codes and demonstrations.

Once the implementation of the system was completed, I was fortunate to be put in contact and have a discussion with two Intel employees, Alexander Nadel research scientist at Intel Israel and Gyuszi Suto principal engineer at Intel Oregon. During this discussion, they provided us with guidance on how it was appropriate to verify the system. The easiest way, for them, was to perform several demonstrations using the different parts of the system and verify experimentally that everything worked. This is far from the verification methods presented in an article [8] explaining how verification is done at Intel that Nadel Alexander suggested reading. But doing

such verification (formal verification and so on) would be the work of one or more master theses. Therefore, after discussion and reading of the article, it was decided to use the experimental approach, in addition to the simulations done on the ALU to prove that some confidence can be given to this system. The demonstrations are explained in this section.

7.2.1 Assembly libraries

Before moving on to the demos, this section briefly describes the libraries written for this job to simplify programming for users. The first one gpu_utils provides all the functions to use the gpu. io_utils provides exactly the same thing but to access the IOs of the IOU. And finally, there is the stack library which provides an implementation of stack. The different functions of these libraries are all documented in their files.

7.2.2 Hanoï towers

The first demonstration is to solve the Towers of Hanoi problem graphically. The Hanoi Towers problem consists of moving a stack of N elements of different dimensions that are ordered from the widest at the base to the narrowest at the top of the tower. To do this, the elements can be moved one by one and placed on a stack. There are three stacks. The tower is initially placed on one of them. For a move to be valid, the element at the top of the stack on which the moved element is to be placed must be larger than this one. In other words, an element must always be placed on top of a larger element. The goal is then to move the whole tower to another stack. When the demo is launched on the machine, a tower of height 6 is created on the first of the three piles and then the program solves the problem step by step, until the tower is placed in the center. Between each movement, there is a wait so that the resolution is possible to follow. This demonstration in action on the system developed in this work is available on YouTube.

7.2.3 Stacker game

Solving the Towers of Hanoi does not include the IOs part, so a small game with interactions has been added. However, here the task was delegated to a friend, Gauderic Schnackers who is a graduated physicist engineer. This allowed two things. First this validated the whole system, but it also validated the accessibility of the system. This is important as the system is to be used as laboratory material in an introductory course. Since Gauderic had never worked so low on a machine and never programmed in assembly, he was the perfect tester.

This game is very simple, the goal is to stack sticks of a certain length to the top of the screen. If two sticks don't stack perfectly, the protruding part of the stick is subtracted and the next stick to be placed is shorter. If the stick is completely consumed before reaching the top, the player loses the game. Again, this demonstration performed on the system of this work is available on YouTube.

Chapter 8

Conclusion

The goal of this work was to design a harvard 32bits CPU, the beta machine, for the laboratories of the Computer Structures [INFO0012] course at the University of Liège. As described in this document, this has been done completely. The machine is implemented with arithmetic, shift, conditional branch and jump operations. As it is a Harvard architecture, it has two memories: one for data (65kB) and one for instructions (131kB). In addition to all this, a register file provides 32 registers of 32 bits to the machine. This is more than enough for many applications.

Access to some IOs as well as a graphical accelerator called GPU in the work has been implemented. They allow to add a lot of interraction for the students. The IO access unit is easily accessible through the CPU Load and Store operations and allows the reading and/or writing of three IOs: two push buttons, four switches and eight LEDs. Concerning the GPU, it allows rendering on a 16:9 60Hz screen with a resolution of 848x480 pixels (of which 576x432 can be written). The writing is done using masks that the user can program. The masks allow to write 8x8 pixels per operation with two different colors (it can also be chosen to clear a pixel or to keep its current value). This allows for a more efficient GPU, in fact in the best case it accelerates the writing by 64, which is a great saving in terms of clock cycles.

In addition, a facility to write and read the values in different memories from the ARM processor has been provided. The ARM processor was chosen as the access to the system developed in this work in order to simplify the access. Thanks to this, it is enough to know how to navigate in a Linux terminal and to launch a program to be able to use the system and the beta machine, which is very simple. The whole work was done with simplicity in mind. Finally, several demonstrations were made to show that everything worked as it should.

This work has therefore fully met its objectives. However, several things could be improved in future work. First, it would be possible to greatly increase the performance of the processor by pipelining its different paths and modules. Just by removing the execution sequence, the troughput would be multiplied by 7. Concerning pipelining, if it is done at all levels (especially for complex operations of the alu: multiplication and division), it would increase the clock frequency of the CPU. With this done correctly, it would certainly be possible to go up to 100 or 200 MHz, compared to the current 50MHz. Then, it is possible to add more IOs. At least managing all the ones on the card and adding audio management on the HDMI would be a nice addition. Finally, modifications to the software on the ARM side and memory access on the FPGA side could allow debugging with breakpoints in the code etc., which could be really interesting for the labs.

Although the system is already functional and easy to use, there are still many possibilities for modifications. The only limit is the capacity of the FPGA, which could be changed anyway if needed.

Bibliography

[1] Cyclone v FPGAs features. https://www.intel.fr/content/www/fr/fr/products/details/fpga/cyclone/v/features.html. Accessed: 2020-2021.

[2] Intel Altera. Cyclone v hard processor system technical reference manual, 2018.

[3] Intel quartus prime design software - support center. https://www.intel.com/content/www/us/en/programmable/support/support-resources/support-centers/quartus-support.html. Accessed: 2020-2021.

[4] Altera. Internal memory (RAM and ROM) user guide, 2014.

[5] Video Electronics Standards Association. VESA and industry standards and guidelines for computer display monitor timing (dmt), 2008.

[6] Intel. Avalon interface specifications, 2021.

[7] Altera. Avalon to external bus bridge, 2015.

[8] Roope Kaivola et al. Replacing testing with formal verification in intel core i7 processor execution engine validation. *Springer*, 2009.

[9] Tom Martin Morris Mano, Charles R. Kime. *Logic and Computer Design Fundamentals*. Pearson, fifth edition, 2016.

[10] Robert H. Halstead Stephen A. Ward. *Computation Structures*. The MIT press, 1989.

[11] John L. Hennessy David A. Patterson. *Computer Organization and Design*. Elsevier, 2014.

[12] Niklaus Wirth. The design of a RISC architecture and its implementation with an FPGA. 2015.

[13] Jikku Jeemon. Pipelined 8-bit RISC processor design using verilog HDL on FPGA. In *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pages 2023–2027, 2016.

[14] Yamin Li and Wanming Chu. Using FPGA for computer architecture/organization education.

[15] Intel Altera. Cyclone v device overview, 2018.

[16] Intel Altera. Cyclone v device handbook: Volume 1: Device interfaces and integration, 2020.

[17] Intel Altera. Cyclone v device handbook: Volume 2: Transceivers, 2018.

[18] Intel. Embedded peripherals IP user guide, 2021.

[19] Intel Altera. AN 709: HPS SoC Boot Guide - Cyclone V SoC Development Kit, 2016.

[20] ARM. Cortex-A9 technical reference manual, 2008.

[21] Chris Terman. MIT course, 6.004 computation structures, 2019.

[22] Hunter Adams. Cornell university course, ECE4760 designing with microcontrollers, 2012.

[23] Mathy Laurent Fontaine Pascal. University of liège, INFO0012-2 computation structures, 2019.