POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Exploration of logic-in-memory architectures for hybrid CMOS/Emerging Technologies circuits

Supervisors
prof. Marco VACCA
prof. Mariagrazia GRAZIANO
prof. Maurizio ZAMBONI

Candidate
Michele Bianco

Academic year 2019 – 2020

Summary

Today, Von Neumann's bottleneck and reaching physical limits in transistor scaling put a stop to the rapid improvement of integrated circuits. The Logic in Memory approach is a promising way to redefine the exchange of information between CPU and memory, memory is no longer seen as a single data storage unit but is designed to be able to perform logic and arithmetic functions. From a more technological point of view, new devices are being studied to replace or complement CMOS technology. In the first part of this thesis a brief introduction of the state of the art of Logic in Memory will be given, the MTJ and pNML devices will also be introduced. In the second part, useful methods will be proposed to create a 3D hybrid system using the two new technologies and CMOS. Through the proposed methodologies, a new memory architecture has been developed that allows the processing of data within it. The thesis work ends by proposing a new type of processor capable of operating a high number of operations in parallel, this is done by exploiting the new memory architecture created.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The first integrated circuits date back to the 1960s, the materials used were mainly germanium and silicon, the latter subsequently becoming the most widely used material. The technological improvements, in terms of the number of transistors per unit of area, since the mid-1960s have been well foreseen by moore's law for about 50 years. This law, assumed that the number of transistors would double every 18 months, this hypothesis was made in realization of the ability of man to reduce the size of transistors. However, in recent years the ability to reduce size has decreased due to physical and economic constraints. Looking at the performance, it grew faster than an exponential because simultaneously increased the density of transistors and the frequency of clock. Today, the power dissipated places a limit on this ,therefore, it is no longer possible to maintain the clock's frequency increase that Moore's law provided. Another aspect that has characterized the last few years is the ever growing gap that has formed between the performance of the processors and those of the memories. The lower rate of growth of the latter worsens the performance of the overall system, the potential of the processors cannot be exploited to the maximum in structures where they have to wait for memory. In the widely used architecture of Von Neumann, this problematic takes the name of Von Neumann bottleneck. Figure 1.1 shows the architecture of V.N. and the Harvard one, also widely used. both provide for a continuous exchange of information between CPU and memories. Various techniques have been seen over the years to alleviate this disparity in performance, the most effective was probably that of creating a memory hierarchy. Fast, expensive and small memories were interposed between the CPU and gradually larger but slower memories. In doing so, it has been

possible to limit the total waiting time of the processor as data is often immediately available. Currently, new architectural solutions are being sought to solve the problem, a new type of approach known as logic-in-memory will be briefly analyzed. Moreover, emerging technologies are nowadays studied for the realization of 3D circuits and to solve the problems of increasing dissipated leakage power in CMOS circuits. For these reasons, the principle of operation of the MTJ and pNML devices will be briefly analyzed in this chapter.

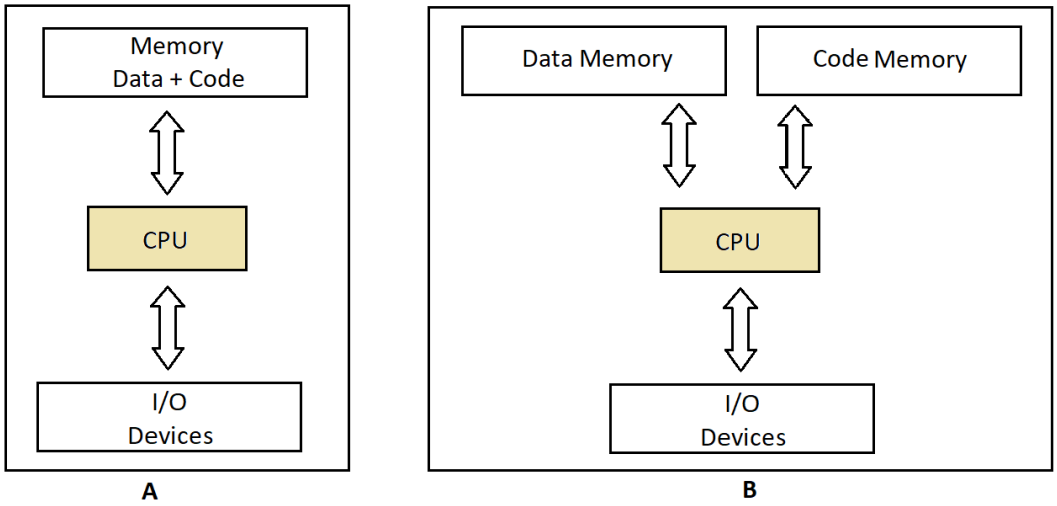


Figure 1.1. (A) Von Neumann architecture (B) Harvard architectures

1.1 Logic in Memory

The purpose of the Logic in Memory approach is to limit the number of memory accesses by bringing the computation into the memory, the limited number of data transfers brings benefits in terms of performance and energy. There are different approaches to Logic in Memory, it is possible to make a classification based on how the storage and computation units interact with each other, in this perspective, four main classes are identified [4]:

- **Computation-near-Memory (CnM):** Memory and logic are still two separate entities, bringing them as close as possible becomes essential to

reduce the length of interconnections and increase the memory bandwidth.

- **Computation-in-Memory (CiM):** This solution is designed to keep the memory array and data writing and reading methods unchanged. The processing functions are inserted in the analog circuits used for reading, usually the sense amplifier circuits are modified. Logic circuits based on sense amplifiers will be analyzed in the next chapter.
- **Computation-with-Memory (CwM):** The results of the operations are taken from a memory in which they were previously saved, usually it is possible to use a look up table that provides the address of the memory containing the result.
- **Logic-in-Memory (LiM):** The storage cells are modified to add computational capabilities inside them, the cells can communicate with each other in different and predetermined ways.

Configurable Logic-in-Memory Architecture (CLiMA)

There is a further approach in which both LiM and CiM strategies converge, Figure 1.3 shows the structure of a Configurable Logic-in-Memory (CLiM) array. the CLiM cell is a memory cell with an additional programmable circuitry capable of performing various operations. Extra-column logic and Extra-row logic are added to the array to perform specific operations. The operations that cannot be carried out within the array are delegated to external units (CnM), this architecture is designed to be as flexible as possible and to exploit the various advantages that the three strategies (LiM, CnM, CiM) offer.

Data movement in CLiM array

The memory array is designed not only to save the data processed by a cell on it but also to allow data mobility over the entire memory. The options allowed inside the memory are shown in figure 1.4 and are the following:

- **intra-row:** computation between more cells in the same row.
- **intra-column:** computation between cells in the same column.
- **inter-column:** computation between two column.

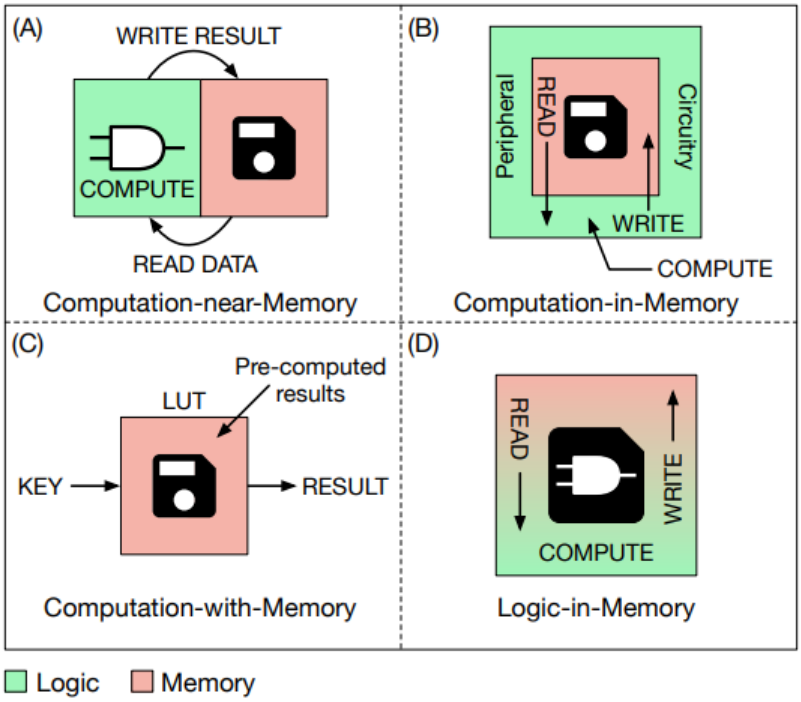


Figure 1.2. Different approaches for in-memory computations. (A) Computation-near-Memory, (B) Computation-in-Memory, (C) Computation-with--Memory, (D) Logic-in-Memory. [1]

- **inter-row:** computation between two row.

The computation skills listed above are possible thanks to additional multiplexers inside the cells, which allow the data to be saved in the neighboring cells in all directions.

Programmable-LiM

Is an approach to build different types of LiM arrays [3], the high level scheme is the one shown in figure 1.5. Memory cells can be traditional or with computational skills (LiM), operations involving cells of the same row are governed by a row interface (RI), while a memory interface manages operations involving multiple rows. The structure is designed to be flexible, the number and position of the LiM cells is decided during the design phase, even the functions of the RI and MI can be multiple. The structure is thought

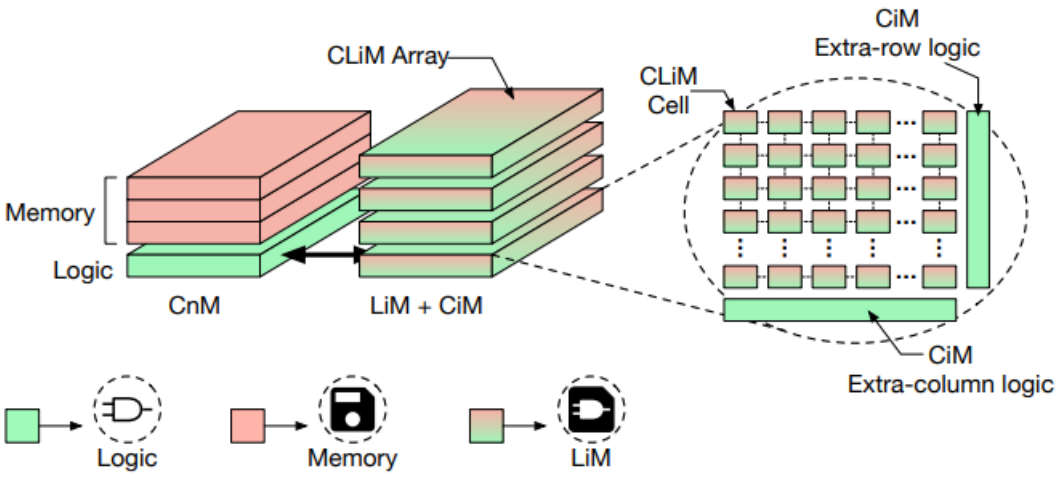


Figure 1.3. Configurable Logic-in-Memory array [1]

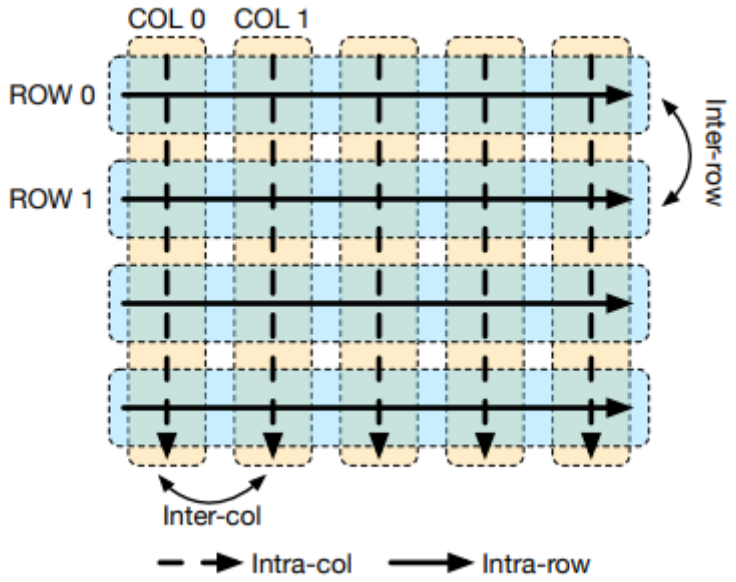
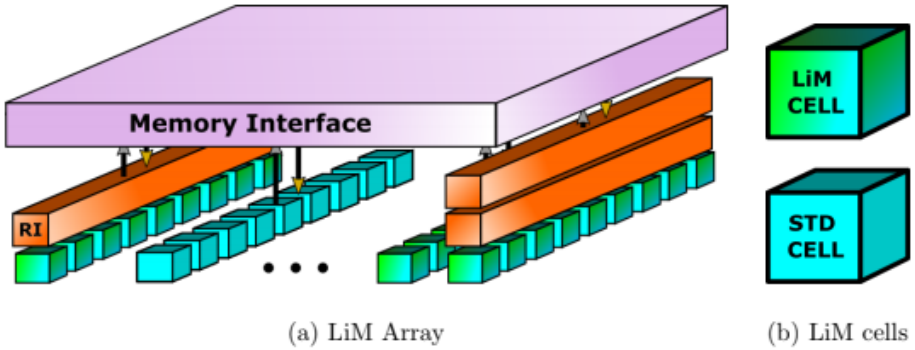


Figure 1.4. Data movement in CLiM array [1]

to be built through the use of structures present in a library and modular with each other. The overall system is designed to consist of a LiM, a CPU and a scheduler. The LiM unit can be used as a traditional memory that exchanges data with the CPU, or to carry out operations inside it. The scheduler decides which operations should be performed with a LiM approach and which not.



(a) LiM Array (b) LiM cells

Figure 1.5. (A) High level schematic of LiM array, (B) Different types of LiM cells [3]

1.2 MTJ technology

A Magnetic Tunnel Junction (MTJ) is a nano structure composed of two ferromagnetic (FM) layers separated by an insulating layer [6]. The magnetization of one of the two layers is fixed, this layer is called fixed layer. The magnetization of the variable layer can be parallel to that of the fixed layer (P) or antiparallel (AP). The figure below shows the Vertical structure of magnetic tunnel junction with CoFeB for the two ferromagnetic layers and with MgO as insulator. The device has two different resistance values in the two configurations (R_{ap}, R_p), for this reason the binary information is contained in the resistance value.

Device reading

In order to read the contents of the MTJ cell, circuits composed of a hybrid CMOS/MTJ circuitry are used. The cell to be read is inserted in circuits

based on sense amplifiers, which exploit the ability of a resistance to influence the discharge time of a circuit.

Device writing (SST-MTJ)

There are several techniques that allow to modify the polarization of the variable layer, the most used is the one based on the spin transfer torque switching mechanism (SST) and we speak in these cases of SST-MTJ. The SST technique is based on making a current I flow in the device for a sufficiently long time, the direction of the current determines the orientation of the polarization. If the current I is a current parallel to the polarization of the fixed layer, the anti parallel configuration (AP) is obtained. Conversely, it is possible to configure the device in the parallel configuration (P) through a current of opposite sign. The switching time (τ) depends both on the intensity of the current and on parameters that depend on how the device is built.

MTJ/CMOS logic gates

Through the use of hybrid MTJ/CMOS circuitry it is possible to create logic gates in which one of the inputs is represented by the MTJ element. All the logic/arithmetic elements can be built by interposing a CMOS logic tree between the MTJ cells and the sense amplifier circuit, the logic tree changes according to the logic function to be implemented.

1.3 pNML technology

Nano Magnetic Logic (NML) is a promising emerging technology in which information is stored in the magnetization of a nanomagnets, binary information can be encoded thanks to its bistable magnetization. They have characteristics of non-volatility, high density integration, low power consumption and possible integration to CMOS. The two main implementations of this technology are: in-plane Nano Magnetic Logic (iNML), and perpendicular nano Magnetic Logic (pNML). Figure 1.7 shows the differences between these two implementations, in iNML devices (A) the magnetization is on the same plane as the magnet, in pNML (B) the magnetization is perpendicular instead. The figure also shows how the two polarizations are associated with the two binary values, in pNML the polarization pointing up is associated with the value '1'. The reversal process of pNML is governed by domain wall

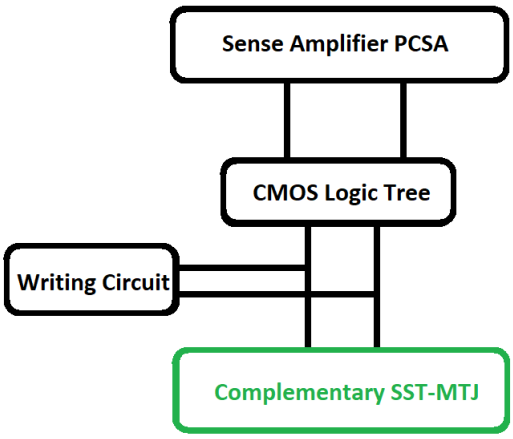


Figure 1.6. Logic gates implementation [8]

(DW) nucleation, the application of an external perpendicular field nucleates the DW. To guarantee the signal flow directionality, one side of the magnet is more sensitive to magnetic field changes. This region is called artificial nucleation center (ANC) and it is obtained by a partial Focus-Ion-Beam (FIB) irradiation. The writing of a pNML device therefore takes place through the nucleation of the ANM and the subsequent propagation of the signal along the magnet. The magnetization of the ANM is affected by the coupling of the magnetic fields of neighboring magnets. however, the fields produced by the neighbors are not sufficient to change the status of the device. In order to modify the magnetization it is necessary to apply an additional external field, this field is produced by a clock. The sum of the fields produced by the neighbors and that of the clock is sufficient to magnetize the device in one of the two configurations.

1.3.1 pNML Random Access Memory

pNML technology can be used to create distributed RAM memories [22], the block diagrams of a standard and a distributed memory can be seen in figure 1.8. In a distributed memory, the content of the cell to be read is scrolled by making it pass inside the underlying cells. This implementation of a RAM

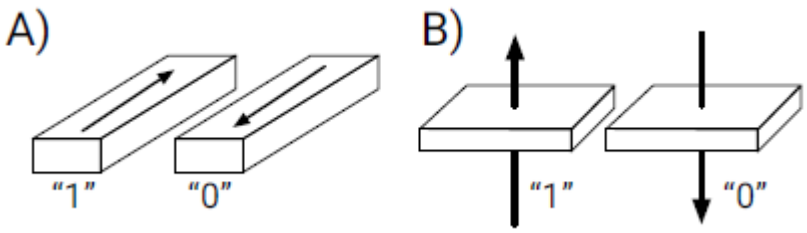


Figure 1.7. Bistable magnetization in Nano Magnetic Logic devices. (A) iNML, (B) pNML [10]

became functional for memories built in pNML technology. This strategy will be the basis of the structures analyzed in chapter 2. The distributed cells can be changed from the standard ones by adding an output multiplexer, the output of the cell can assume the value stored in the cell itself, or the content of the previous one. Having cells with small integrated multiplexers allows to have a lower input-output latency than in the case of a single output multiplexer.

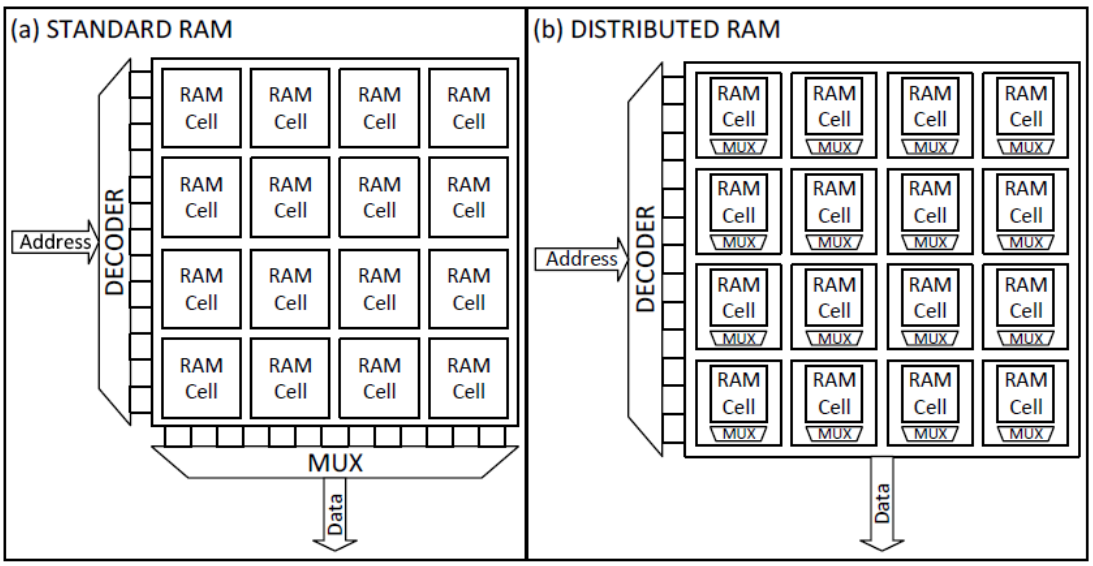


Figure 1.8. Memory block diagram. (a) Standard memory, (b) Distributed memory [15]

Chapter 2

3D hybrid system

So far it has been quickly analyzed, separately, the operating principle of the two technologies. In this chapter the goal is to study the project methodologies aimed at creating a hybrid MOS/MTJ/pNML system. Such a hybrid system is interesting as it can be developed on a three-dimensional (3D) structure, furthermore, the compatibility of the pNML logic with the data in MTJ format is immediate, therefore it does not require the conversion of the format by reading as happens in a MOS/MTJ structure. The idea is to have a generic CMOS circuitry, this communicates with a data memory in which the information is stored in MTJ cells, the contents of the cells can be processed through a pNML circuitry physically located above the memory, in the passage a pNML memory can be placed between the MTJ memory and the logic where a copy of the data is made. It is useful to think of the MTJ memory as the main data memory and the pNML memory as a kind of register file.

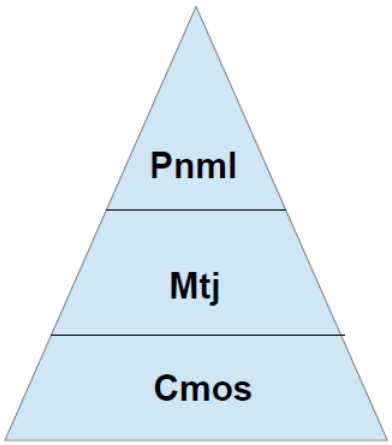


Figure 2.1. 3D sistem

The goal is to create useful guidelines for the implementation of this hybrid system, for this purpose, the following points will be examined:

- **Transfer from MTJ to pNML**
- **Transfer from pNML cells to cells of the same type**
- **Reading of the pNML memory:** how to send data out
- **Logical / arithmetic plan**
- **Transfer from the logic plane to the pNML cells**
- **Writing of the MTJ memory**

2.1 Transfer from MTJ to pNML

The minimum transferable unit is assumed to be the MTJ memory row, this will be called line in pNML memory. To transfer only a subset of the row, a multiplexer must be introduced. Given a memory MTJ composed of 2^n rows and a memory pNML composed of 2^m lines (figure 2.2), the purpose is to study the transfer from the first structure to the second in the following cases:

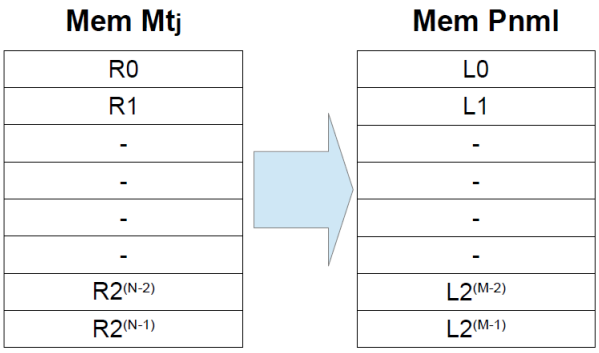


Figure 2.2. From MTJ mem. to pNML mem.

- **Transfer with the same memory** $n = m$
- **Transfer with** $n > m$

2.1.1 Transfer with $n = m$

In the two cases two further distinctions are made:

1. Single destination transfer:

   The free layers of the MTJ cells are extended until they become the possible inputs of the pNML cell, an input multiplexer is necessary in order to select one of the possible inputs (Fig 2.3).

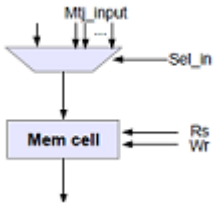


Figure 2.3. MTJ input in pNML cell

In the case in question there is a one-to-one correspondence between row and line, which is why each line communicates only with one line. This solution has the minimum number of inputs on the multiplexer, it is a simple but inflexible solution, moreover, the pNML memory is a simple copy of the main memory. Such a solution can be useful in systems whose purpose is to interpolate the data inside a memory without however wanting to modify the main memory. The figure 2.4 shows an example of this type of transfer.

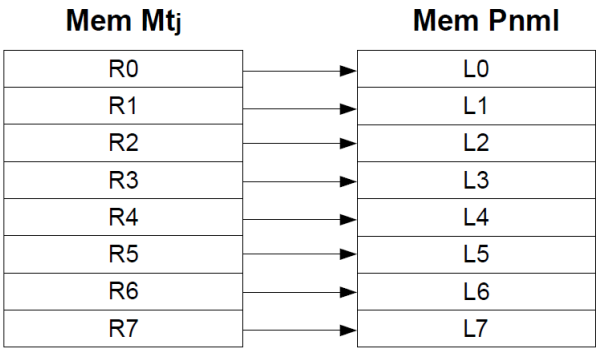


Figure 2.4. Single destination transfer

2. Multiple destination transfer:

The aim is to increase the flexibility of the system by introducing the possibility of having multiple destinations for each single row. Each row can communicate with B lines. Let B be the number of blocks, then, $B = 2^n/2^k$ with 2^k number of lines per block. Given the generic row R_i with $[0 <= i <= (2^n - 1)]$, it communicates with the line j of all the blocks with $j = remainder(i/2^k)$ with $[0 <= j <= (2^k - 1)]$. This solution presents greater flexibility, greater complexity and more area due to the fact that on each multiplexer there are B inputs. The figure 2.5 below shows a simple example of this transfer mode with two blocks and therefore two possible destinations for each row.

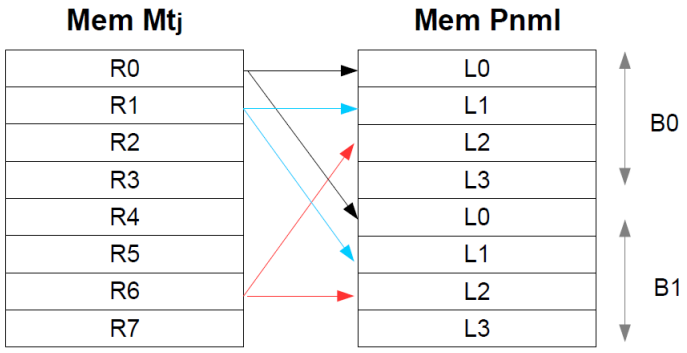


Figure 2.5. Multiple destination transfer

2.1.2 Transfer with $n > m$

Being the pNML memory interposed between the MTJ data memory and the logic plane, very often there is interest in having it smaller and therefore faster than the first. Also in this case there are two further distinctions:

1. Single destination transfer:

   Also in this case the first memory is composed of 2^n rows and the second of 2^m lines, being $n > m$, each single line must contain one of the $2^{(n-m)}$ possible rows(Fig. 2.6). In this case a $2^{(n-m)}$ input multiplexer is required. Thinking of a hypothetical control unit that generates the signals necessary for the transfer of the row, n addressing bits will be necessary. Of these, the $n-m$ MSB can be used as a multiplexer selector, the rest will identify the line.

2. Multiple destination transfer:

   It is possible to increase the level of flexibility by dividing the memory into blocks and using the strategy seen above, however, the number of possible MTJ inputs on each cell becomes $2^{(n-m)} * B$. This solution if not well designed can greatly increase the complexity.

2.1.3 Design choices

In a structure with the same size of memory there is a larger area of the pNML memory but a more essential transfer circuitry. So these considerations can

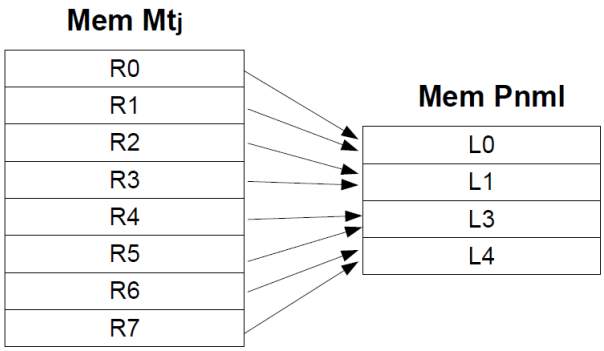


Figure 2.6. Single destination transfer n>m

be useful:

- If you want a fast memory and you are not constrained in the transfer time, better $n > m$.

- If you want a slower and larger memory or if you are constrained in the transfer time, better $n = m$.

- The subdivision into blocks increases flexibility at the expense of transfer time, does not affect the size of the memory and therefore does not affect its internal speed (line reading, etc.).

The figure 2.7 summarizes how the size of the input multiplexer varies according to the design choices.

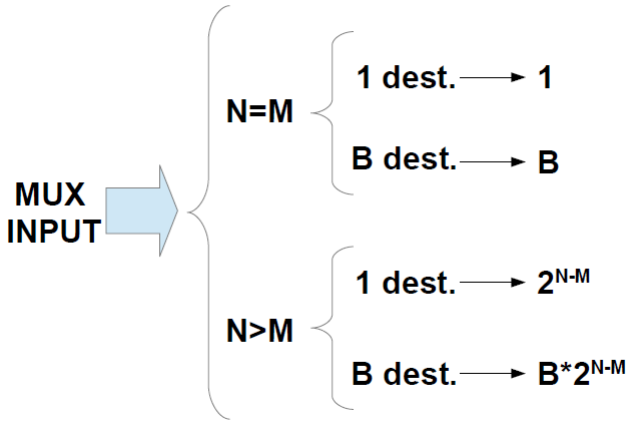


Figure 2.7. Number of MTJ input in the multiplexer of each pNML cell

2.2 Transfer between pNML cells

The hybrid structure that is being studied in this chapter could be used in order to create a system that uses logic in memory as a strategy of interpolation of data in memory, from this point of view it is useful to introduce methodologies that allow to move data to the internal memory itself. It is assumed to have a distributed pNML memory which allows a flow of lines within the memory, therefore also in this case the line is the movable elementary unit. In the previous chapter it was seen the elementary structure that allows to have a distributed memory, in our case it is necessary to introduce an input multiplexer as it is not the only upper line to be a possible input. Three types of elementary structures have been identified that can make up memory. The architecture of these three structures is visible in the figure 2.8, followed by a brief description of the advantages and disadvantages of each. The input data (*datain*) is the generic data contained in another line of the memory that at that moment crosses the cell, it is known from what was studied in the previous chapter that in a distributed memory the output data can assume the value of the stored data or the one entering the cell itself, in the latter case the cell is in a transparency mode. The two modes of reading or transparency are selectable with the output multiplexer. If the cell wanted to be written, the desired input must be selected and the reading signals (Rs, Wr) must be activated.

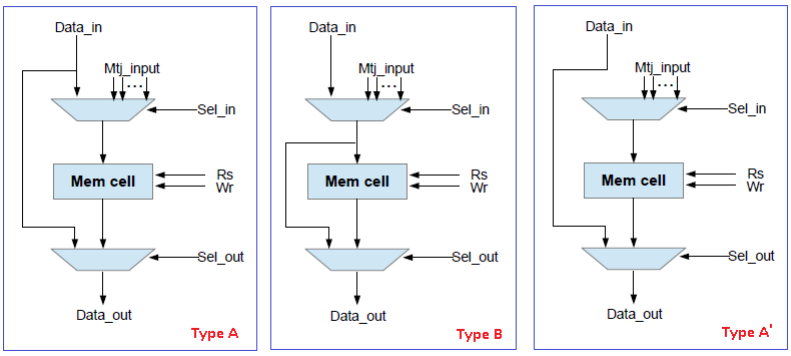


Figure 2.8. Types of elementary structures

The three structures have however some differences:

- **Type A:** The exchange of data between lines is allowed, there is no path through which the data coming from the MTJ memory can flow through the distributed memory without entering the cell. The reading (or transparency) and writing of the cell can take place at the same time given the separate paths.
- **Type B:** Transfer between lines is allowed. In this structure, the MTJ data can bypass the cell and go directly to another part. however, the storage of the MTJ data is not allowed if at that moment the cell is involved in a read operation, the read and write operations are conflicting.
- **Type A':** This structure is similar to the type A one but does not allow the transfer between lines, however it has been introduced as it will be useful in other sections of the thesis.

The transfers between cells are therefore supported by structures A and B, the flow can occur along one direction only or along two directions. The first case is now analyzed.

2.2.1 Unidirectional flow

In this case the data flow can only take place in one direction, it is an inflexible solution which nevertheless allows to study the components necessary to support the movement of data within the memory. There are several possibilities:

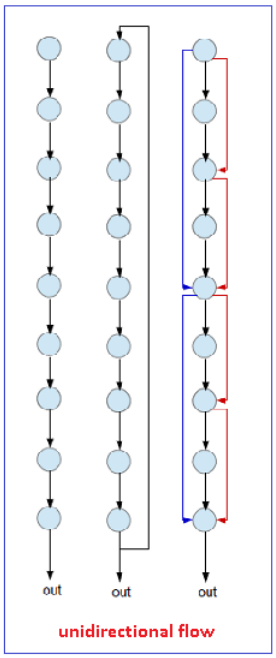


Figure 2.9. Unidirectional flow. From left to right: unidirectional flow through adjacent lines, unidirectional circular flow, unidirectional tree flow.

1. Unidirectional flow through adjacent lines:

   The information flows from top to bottom and passes through each cell, it is the simplest solution both from the point of view of memory hardware and as regards the control signals. In the left side of figure 2.10 an example realized with elementary structures of type A is visible, supposing the transfer of the line i into the line j ($j > i$) is required, it is necessary to: select the output multiplexer of cell j in the mode of reading, select the output multiplexers of all the intermediate lines in transparency and finally set cell i in writing mode ($selin$, Wr, Rs). How to generate the control signals starting from the addresses of the lines

will be better examined in the next section. What has been said remains valid even when using type B structures.

2. Unidirectional circular flow:

   In order to also allow the transfer of data from a lower line to a higher one, the first and last line can be connected. The flow remains unidirectional but the transfer is no longer just from top to bottom. Net of the connection between the last and the first line, the architecture and therefore its functioning remain unchanged.

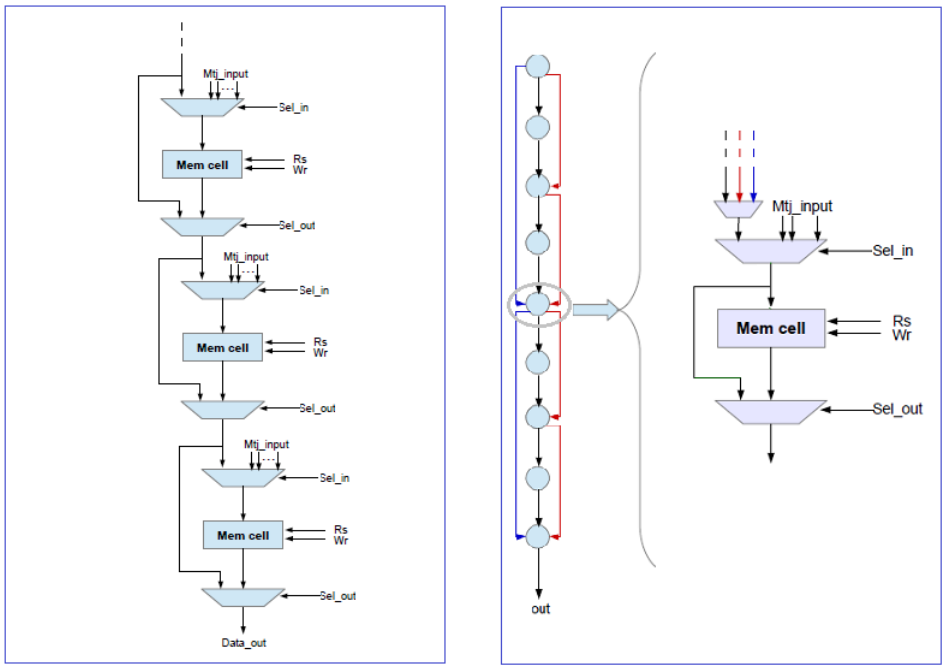


Figure 2.10. Unidirectional flow hardware architectures

3. Unidirectional tree flow:

   Additional multiplexers can be used to minimize the maximum number of cells traversed. This technique involves a small increase in area but also a more complex control. in the right part of the figure 2.10 it is visible how it is possible to manage more input lines with the simple addition of a multiplexer, the fact of having separate multiplexers could simplify the control in fact keeping everything as before (reading,

transparency and writing) will be sufficient a additional control unit to manage the additional multiplexers, the selection will be made according to the starting address and the destination address. That said, if the project requires particular attention to consumption, a system will be needed that makes only the necessary lines transparent in order to avoid unnecessary switching of the data. Given a system with unidirectional tree flow, the maximum number of inputs in the additional multiplexer is equal to: $\log 2(numberoflines)$.

2.2.2 Bidirectional flow

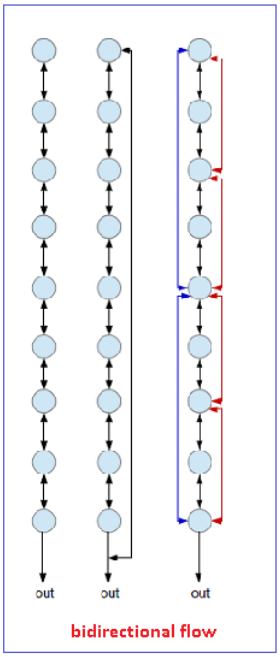


Figure 2.11. Bidirectional flow. From left to right: bidirectional flow through adjacent lines, bidirectional circular flow, bidirectional tree flow.

1. Bidirectional flow through adjacent lines:

   With reference to the figure 2.13, the information flows in two directions and passes through each cell. An additional multiplexer on the cell is required to choose the direction, the output of the new multiplexer can be one between the output of the previous cell and the output of the

next one. Flow direction is controlled by the *up/down* signal. The figure 2.13 also shows an example with elementary structures of type B, note that in this case also *Selin* must be correctly set.

2. Bidirectional circular flow:

   Compared to the previous case, a direct connection is added between the last and the first line. In this case, the generation of the control signals must be done on the basis of the shortest path to reach the destination.

3. Bidirectional tree flow:

   As seen previously, additional multiplexers can be used to minimize the maximum number of cells crossed. Figure 2.12 shows the changes made on the architecture to make it bidirectional. The inputs of the up / down multiplexer are selected from among all the possible ones, both up and down. Compared to the unidirectional case, double the additional multiplexers will be needed.

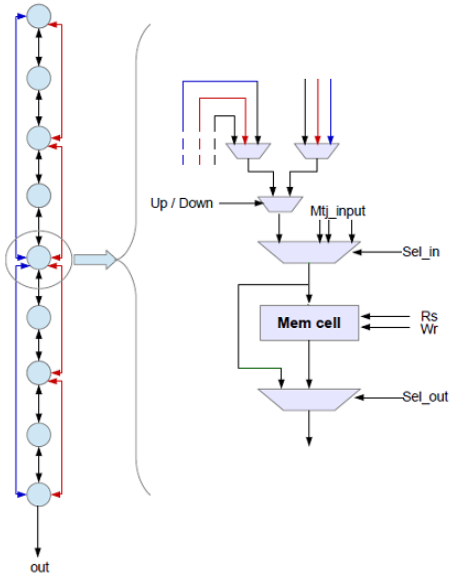


Figure 2.12. Bidirectional tree flow hardware architectures

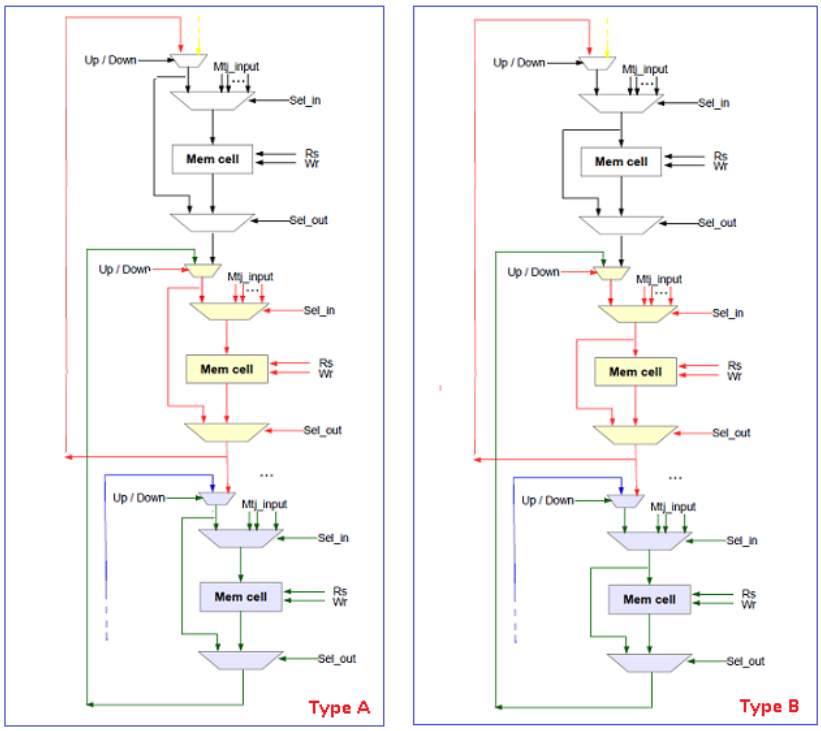


Figure 2.13. Bidirectional flow hardware architectures

2.2.3 Flow with transfer between groups

In order to create a useful memory for logic applications in memory, it may be useful to divide the memory into groups if. For example, an ALU is associated with each pair of groups. In this case there is interest in moving the data from one group to another without too much interest in the line. Such a solution, as we will see later, can also be exploited to increase the memory reading parallelism. All the techniques studied up to now are valid and can be adopted within the single group. In the example shown (figure 2.14), the transfer is allowed between lines with the same position in the different groups. Within the same group, movements through adjacent cells are allowed.

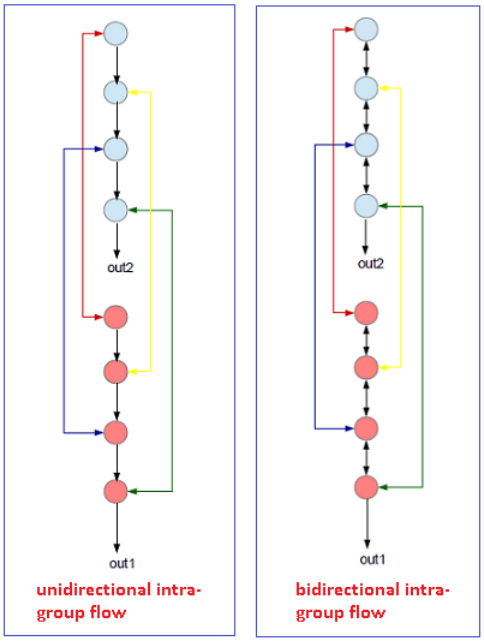


Figure 2.14. Flow with transfer between groups

2.3 pNML memory reading

The structures just seen also allow the reading of the cells, the read line goes through the memory down to the exit. If there is no interest in memories that allow internal transfer, type A' cells can be used with the same methods seen above. However, the goal is to implement structures that allow the reading of several lines at the same time. In this perspective, two distinct techniques have been identified. A third solution is also possible, given by combining the two.

1. Creation of groups with transfer between group.
2. Creating multiple exit routes.

2.3.1 Line reading

In order to understand how a subdivision into groups of the memory can allow the reading of more lines at the same time, it is useful first of all to

analyze how the reading of a line occurs in a structure made by a single group. The example 2.15 shows a memory made with elementary structures of type A'.

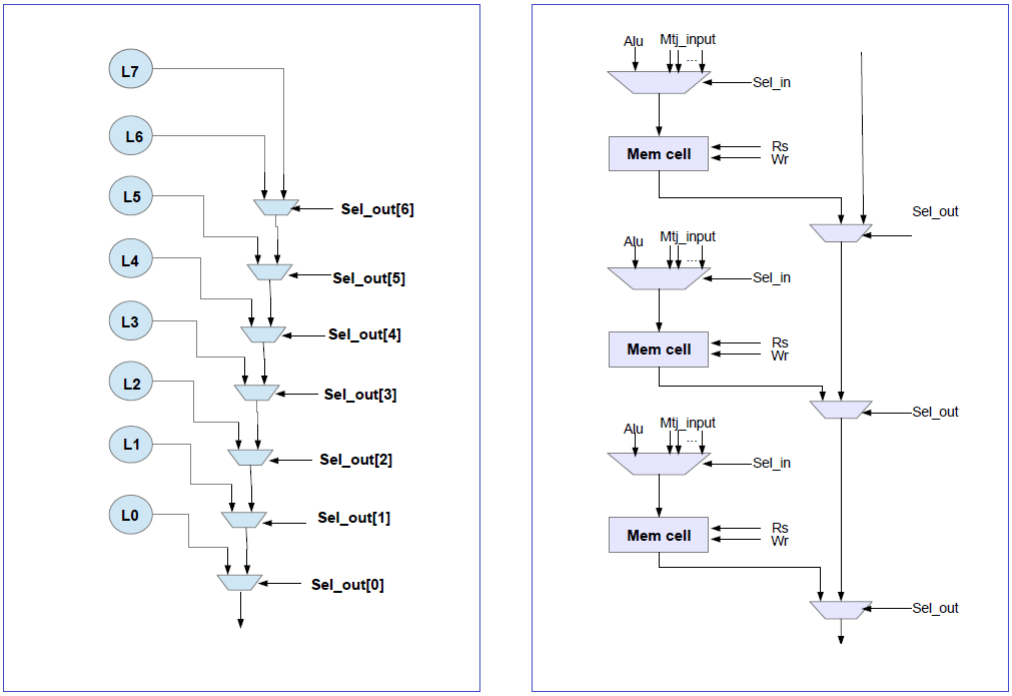


Figure 2.15. Memory reading

The only thing necessary is the address of the line to be read (note that line 0 is the one connected to the output). The address is encoded in a signal consisting of all the output multiplexer selectors: $Sel =' 0'$ cell content, $Sel =' 1'$ previous output. The address is encoded in such a way that its value indicates the number of ones in the vector of the selectors starting from LSB, in this way it is possible to set all the multiplexers between the line and the output in transparency mode. Here are some examples of addresses from which the selection vectors are obtained:

Ad="010" → Sel_out="0000011"
Ad="000" → Sel_out="0000000"
Ad="100" → Sel_out="0001111"

2.3.2 Transfer between two groups

In order to understand how it is possible to read several lines simultaneously through this technique, let's take for example the structure in the figure 2.16. The memory is the same as before but divided in two.

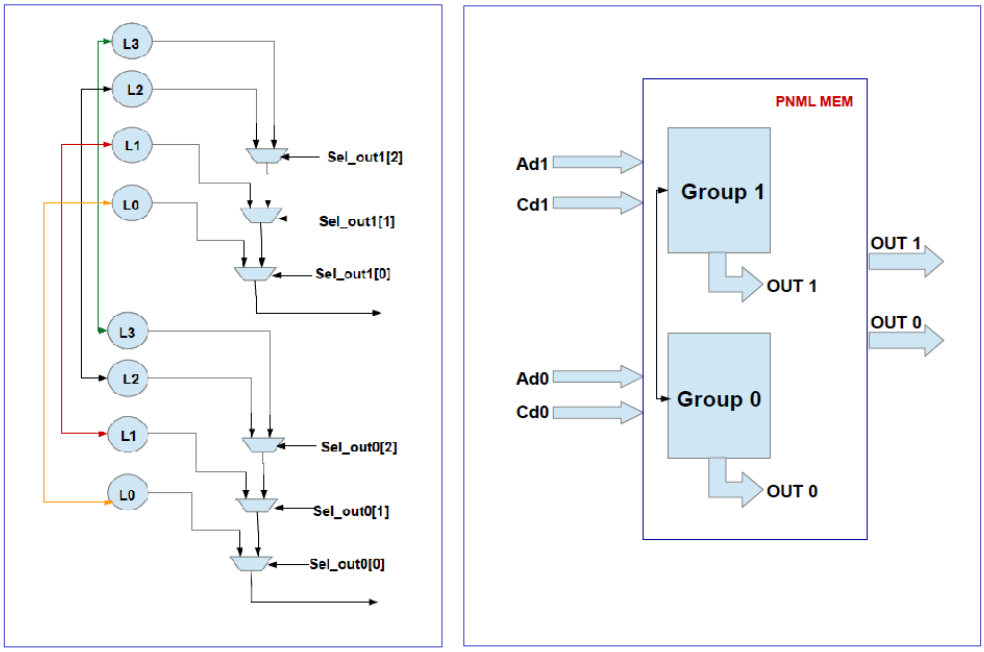


Figure 2.16. Memory divided into two groups

The simultaneous reading is done through the two available outputs, one for each group. In the simplest case in which the lines belong to different groups, the reading is immediate. In the event that the two lines are on the same group it is necessary to transfer one of the two, in this case the group receiving the information will have the task of making it flow to the exit. On the right of the figure 2.16 we can see the block diagram of the

memory, the signals Ad are the addresses of the line to be read already encoded in the vector of the output selectors, the vectors Cd are used to transfer information between the groups. The figure shows the hardware changes that must be made in order to support the transfer between groups, an additional multiplexer is interposed between the output multiplexer and the contents of the cell. The purpose of this modification is to allow the reading of the same address line of the other group. Supposing that the line i of $group0$ through $group1$ wants to be read, it will be sufficient to read the line i of $group1$ by setting the multiplexer on the line of $group0$. The vectors C are those containing the selectors of the added multiplexers.

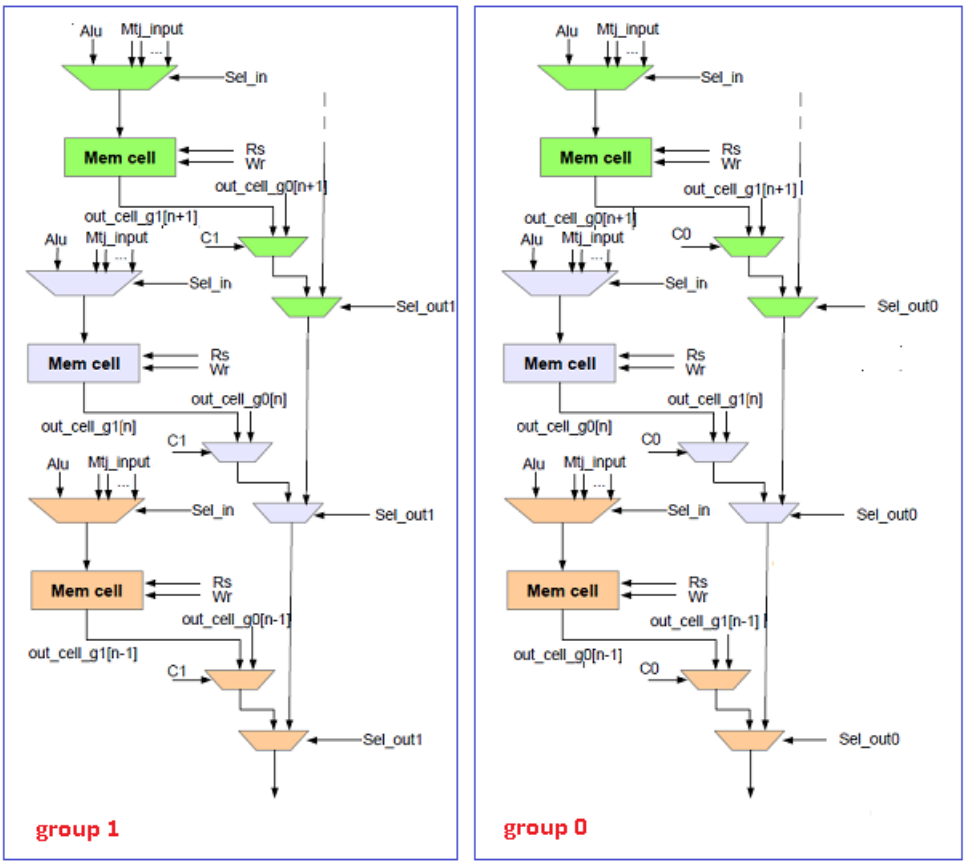


Figure 2.17. Memory group reading architecture

Generation of reading signals

The figure 2.18 summarizes through a block diagram the fundamental steps to read several lines simultaneously using this method. $R1$ and $R2$ are the addresses to read, at this moment the memory is seen as unique. The command generator has the task of generating the addresses of the individual groups (R private of MSB) putting them in the right order. If $R1$ and $R2$ have the same MSB, an address must be transferred to the other group. $C1$ and $C0$ are one-bit signals that indicate whether to activate C corresponds to the address $A0/A1$. As the passage through a different group increases the path to the exit it is convenient to move the line closer to the exit.

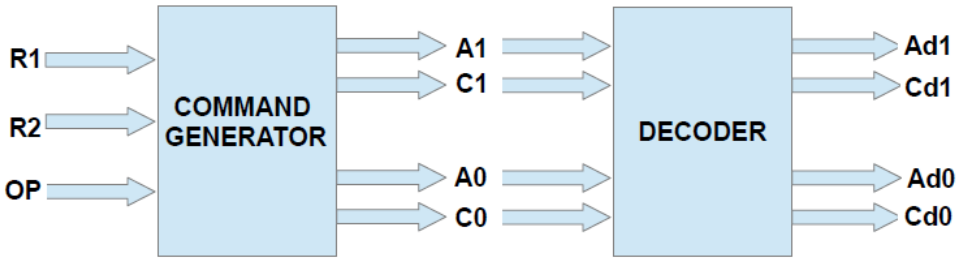


Figure 2.18. Generation of reading signals

- **Example:** Suppose to have the goal to read memory lines 4 and 6, so $R1 ='' 100''$ and $R2 ='' 110''$. Since $R1$ and $R2$ have the same MSB they belong to the same group, $R1$ is the smallest value and must be transferred. At the command generator output we have: $A0 ='' 00''$, $A1 ='' 10''$, $C0 ='' 1''$, $C1 ='' 0''$. Finally follows the decoding, from which: $Ad0 ='' 000''$, $Ad1 ='' 001''$, $Cd0 ='' 0001''$, $Cd1 ='' 0000''$.

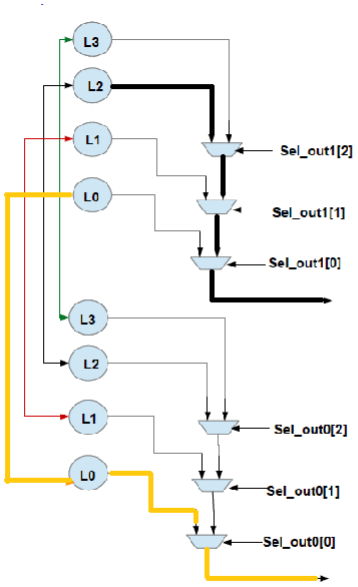


Figure 2.19. Reading path

2.3.3 Transfer between more than two groups

The concept can be extended to read 4, 8 .. 2^n words simultaneously. The memory must be divided into two groups and proceed, using the same division by two method, on each group. The figure 2.20 shows how the starting memory would look once it has been divided into four blocks. Through this method it is possible to read multiple words from memory at the same time (one for each block), the system implementation requires to allocate new hardware resources. In particular, having G groups of L lines each, for each group L multiplexer with G inputs (red in Fig. 2.21) are required to select one of the possible lines of equal addressing of each block, however, each grouping eliminates an output multiplexer on each block (green).

Generation of reading signals

If there are more than two blocks, the control is slightly complicated. The goal is to have a compact memory seen from the outside, the inputs will therefore be the only addresses of the lines to be read. Suppose to have a memory made up of 2^m lines, the address will therefore be made up of m bits.

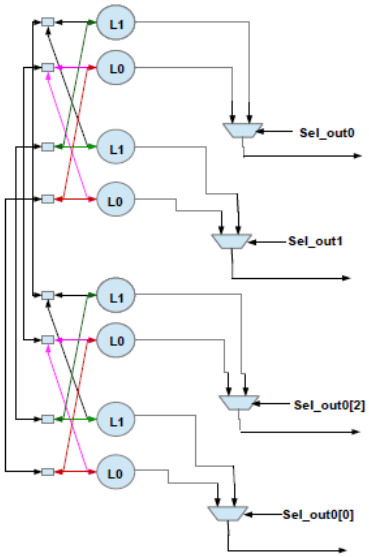


Figure 2.20. Memory divided into four groups

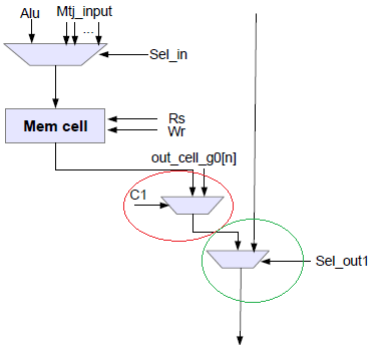


Figure 2.21. Line and output multiplexer

The memory is then internally divided into G groups, being G a power of two 2^g, it means that each block will have $2^{(m-g)}$ lines. It therefore takes $m - g$ bits to point every single line in a block. The remaining g bits of the starting address can be used as multiplexer selector (red), in this perspective it is necessary to find an intelligent way to select the possible lines. The simplest solution is the one in which the value of the selector indicates the number of

the block from which the line is to be taken. Referring to the block structure of the previous case, it can be extended as shown in the figure 2.22.

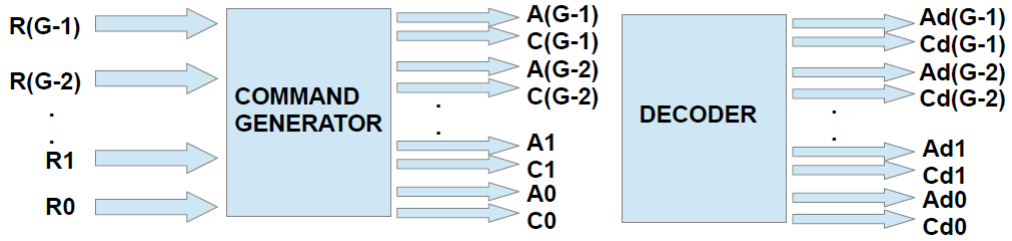


Figure 2.22. Generation of reading signals for more that two groups

Each address R is associated with the pair C and A, in reality the command generator consists in the simple separation of the address as shown in the figure 2.23, therefore it does not complicate the control in any way. The decoder always works in the same way in the generation of signals Ad. The signal Cd is instead a vector equal to the size of the block in which each element is made up of g bits, the purpose of the decoder is to set the element A of the vector equal to C. This technique is general and can also be used in the simple case of two blocks.

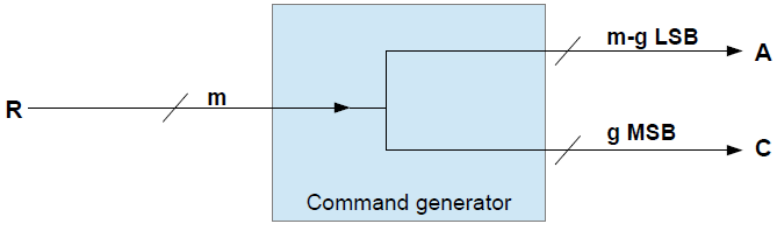


Figure 2.23. Command generator

Interconnection increase

This subsection aims to study what are the costs, in terms of additional hardware, of having a subdivision of the memory into groups. In particular, the purpose is to analyze the total number of interconnections necessary to read the memory. There is also an increase in the real memory area, in fact with the increase of the groups it is necessary to increase the area of the line multiplexer. The analysis now aims to study the problem of interconnection of reading control signals, trying to understand how they change. In this regard, the table is divided into four columns:

	Out mux	Line mux	Signals l.mux	Decoder out
no groups	$2^m - 1$	0	0	$2^m - 1$
2^g groups	$2^m - 2^g$	2^m	g	$(2^m) * (g + 1) - 2^g$

Table 2.1. Hardware complexity

- **Out mux:**

  Indicates the number of multiplexers to be inserted at the line output over the entire memory. These multiplexers always have unitary weight in terms of control signals, since the only signal needed is a one-bit selector. In the analysis the size of the line is not taken into account, this is because the subdivision of the memory does not affect the size of the line, all the values in the table must be multiplied by the number of cells in a line to get the real total number. If the memory is unique the number is equal to 2^{m-1} because two cells share a multiplexer. If the subdivision is made, each group has $2^{(m-g)}$ line, for each group we therefore have $2^{(m-g)} - 1$ multiplexer. To obtain the total number it is finally necessary to multiply by the number of groups. Once multiplied the value in the table is obtained.

- **Line mux:**

  Indicates the number of line multiplexers to be inserted on the entire memory. Remember that these multiplexers are those useful for selecting all possible lines with equal addressing within the group (red). If there is no subdivision these are not necessary. If there are groups, it is sufficient to observe that on each line it is necessary to insert one before the output multiplexer.

- **Signals l.mux :**

  Number of control signal on the single line multiplexer (bit number of the selector), equal to g.

- **decoder out:**

  Indicates the number of outputs of the decoder block. This is the most important parameter because it quantifies the total number of interconnections necessary to control the memory. Is the sum of selectors of the two types of multiplexer $(g * 2^m) + (2^m - 2^g) = (2^m) * (g + 1) - 2^g$

The control signals are not the only ones to weigh on the interconnections, necessary for reading, when this is divided into groups. The values of the lines must in fact be carried from one group to another, these signals are input to the line multiplexers. Adding these additional interconnections to those coming out of the decoder 5.3.1 is obtained.

$$[(2^g + g) * 2^m] + (2^m - 2^g) = (2^m) * (2^g + g + 1) - 2^g \tag{2.1}$$

This expression is the one that indicates the total number of read interconnections, the graph in the figure 2.24 shows its value with m between 5 and 20 for different values of G $(= 2^g)$. for $G = 0$ only the decoder outputs are used. It is useful to analyze the growth in the number of interconnections, due to the subdivision, by comparing it with the number of interconnections with the same m but with a memory not subdivided into groups. The graph 2.25 shows that the ratio decreases with the amount of groups, it can also be noted that the latter depends on the number of groups but only slightly depends on the number of memory lines. However, the ratio normalized respect to the parallelism of reading (figure 2.26) provides a better indication than the usefulness of the division into groups. This parameter indicates how much the interconnections to read G words increase compared to the interconnections outside the memory that would be needed if the memories were really G.

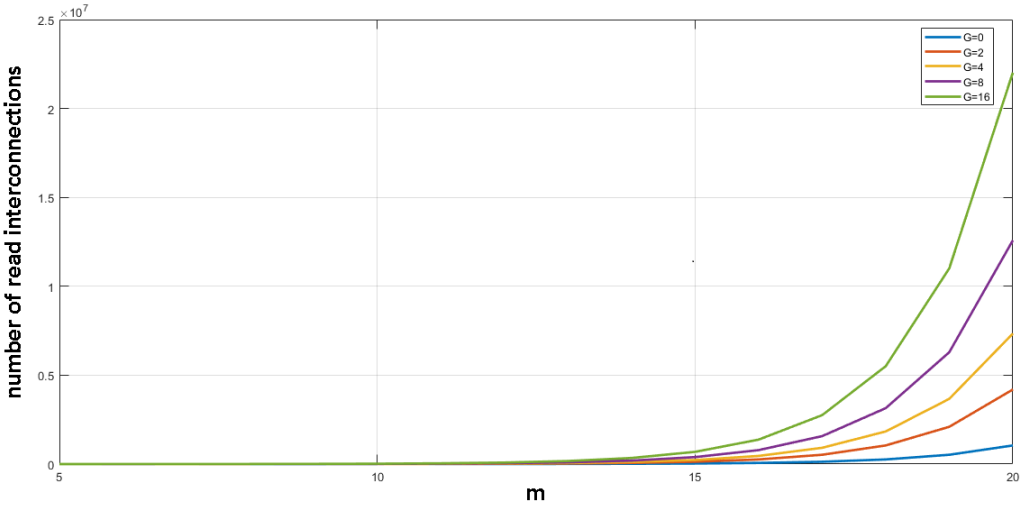


Figure 2.24. Read interconnections vs m with different g

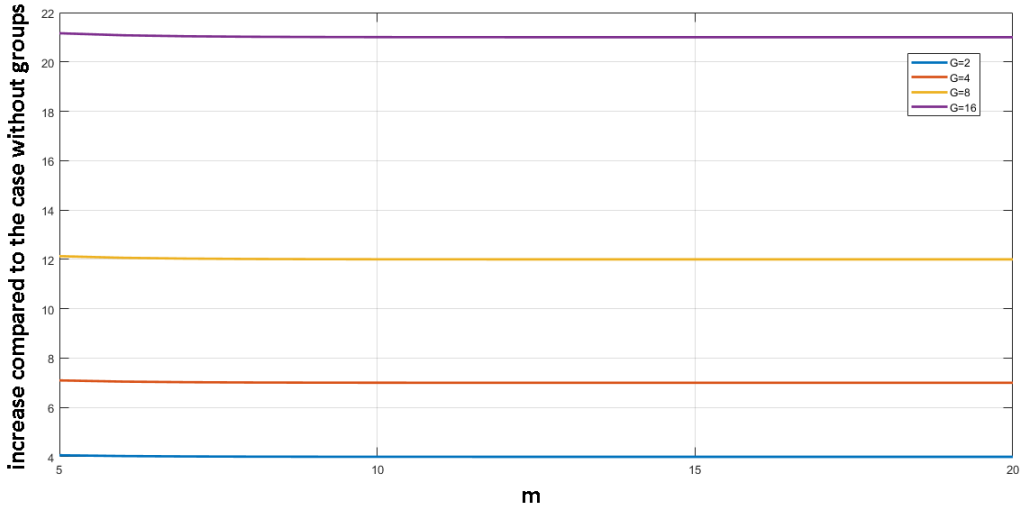


Figure 2.25. Increase respect to no-groups

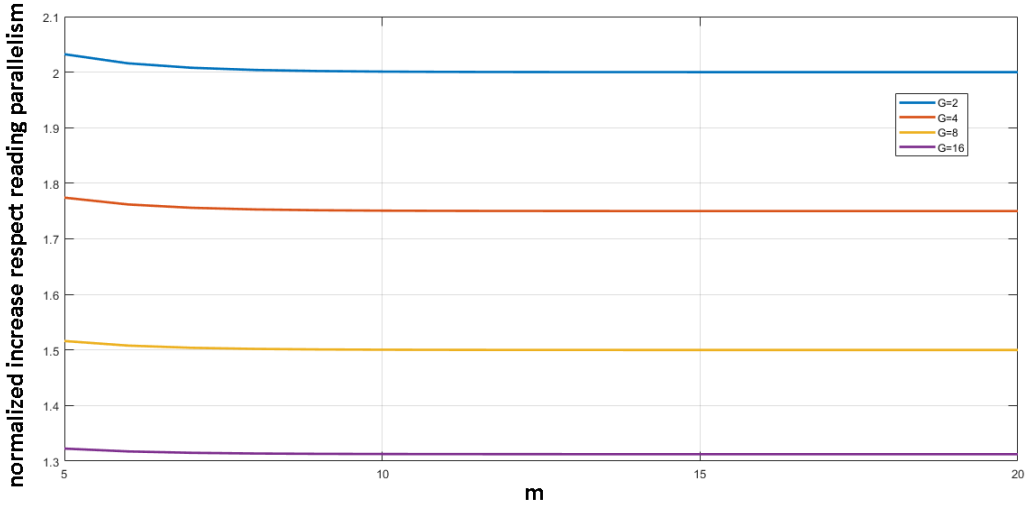


Figure 2.26. Normalized ratio vs m

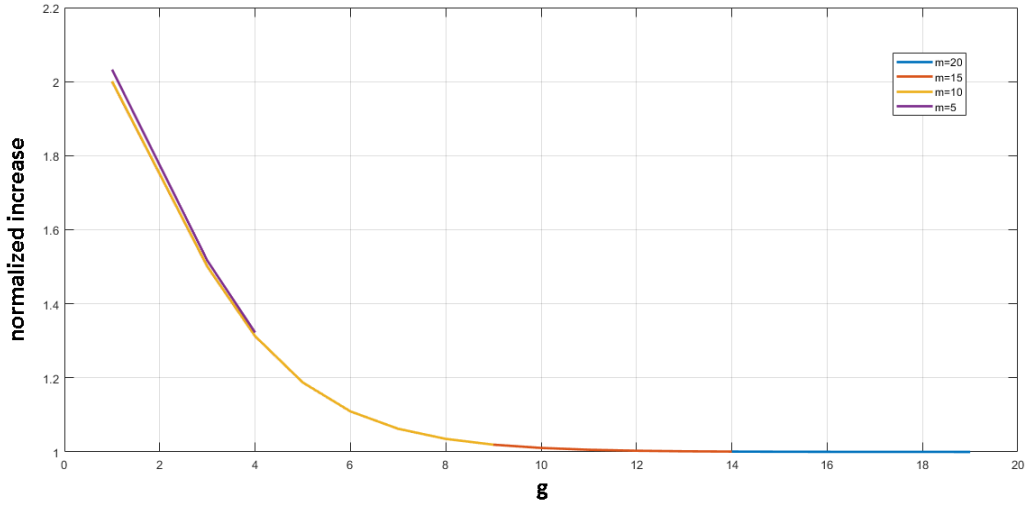


Figure 2.27. Normalized ratio vs g

The figure 2.27 shows the trend of the normalized ratio, for several m, as a function of g. It can be noted that the value becomes unitary as a first approximation for values of g greater than ten, this value however represents a very strong parallelization, difficult to use in reality. For more realistic values, greater interconnections are required respect to have more memory, however, if it is true that as regards the external interconnections there is a loss, there is a considerable saving in the area of the memory that should not be replicated. Another important advantage of the subdivision into groups is that of having a reduced critical path compared to a single memory.

Critical path

The main benefit of having a grouped memory is to significantly reduce the critical path. Assuming that the propagation of the signal on the line is negligible, the critical path can be seen as the sum of the multiplexers that the data must cross to reach the output. Since the number of inputs affects the input / output delay, it is possible to imagine multiplexers with more than two inputs as composed of a series of multiplexers of two.

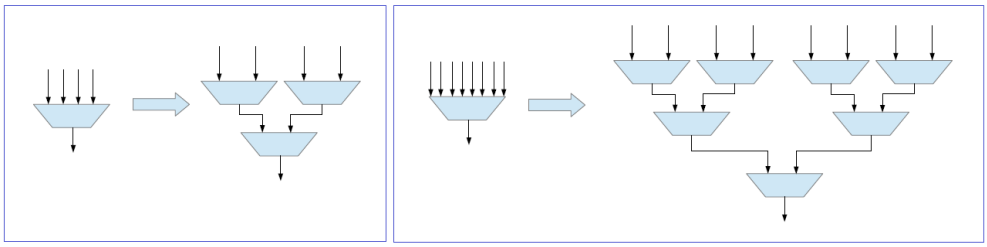


Figure 2.28. Splitting of the multiplexer

In the case of standard distributed memory, the critical path is $2^m - 1$ multiplexer. Once the subdivision has been applied, the equation 5.2 provides the new critical path.The first part of the equation $(2^{(m-g)} - 1)$ represents the maximum number of crossings for each group, the additional value g represents the number of line multiplexer levels to cross. Figure 2.29 shows the trend of the normalized critical path as a function of m, figure 2.30 shows the trend as a function of g for different values of m.

$$Cp = 2^{(m-g)} - 1 + g \tag{2.2}$$

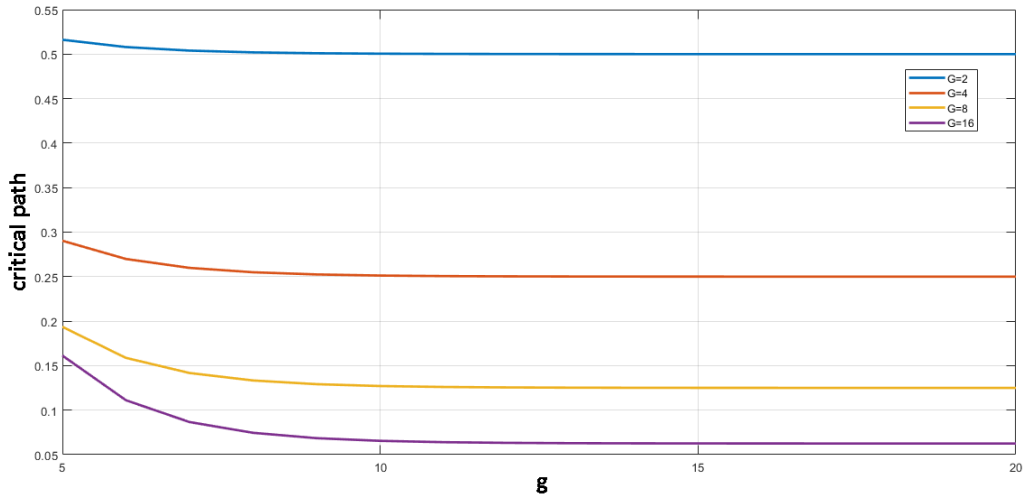


Figure 2.29. Critical path vs m

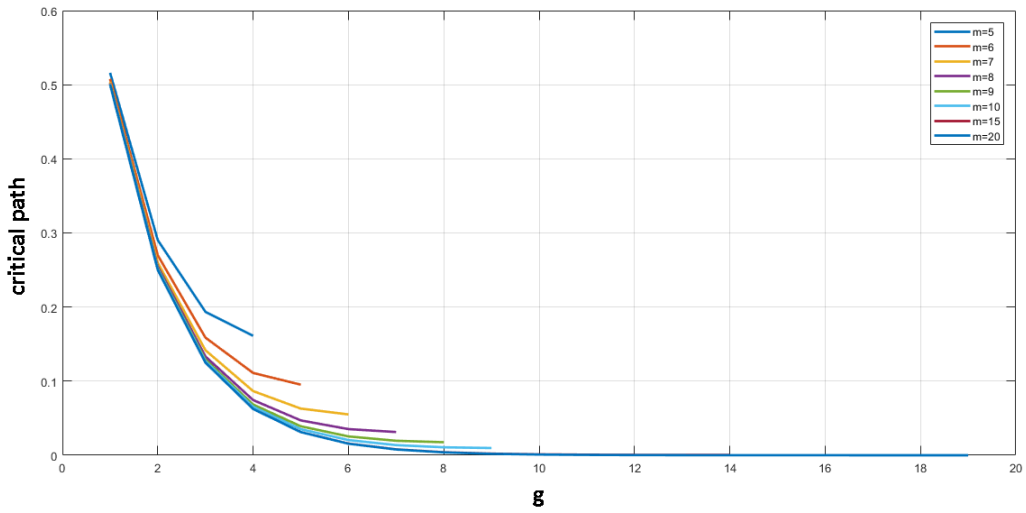


Figure 2.30. Critical path vs g

Area

Now it is necessary to try to estimate the increase in memory area, the increase in area is essentially due to the multiplexers to be inserted to select the line. It is supposed to know the area (At) of the memory before it is subjected to a division. It is also assumed to know the ratio K between the total area At and the area dedicated only to the reading circuitry (mux out and mux line), so $K = At/Ar$. The total area after the subdivision into groups is therefore:

$$Ag = At * (1 - k) + At * k * [((2^g - 2^m) + 2^m * (2^g - 1))/(2^m - 1)] \quad (2.3)$$

The area is the sum of the portion of non-reading circuitry plus that dedicated to reading multiplied by the rate of increase. By simplifying the results:

$$Ag = At * (1 - k) + At * k * 2^g \quad (2.4)$$

dividing by At:

$$Ag/At = (1 - k) + k * 2^g \quad (2.5)$$

The trend is shown in figures 2.31 and 2.32. Figure 2.33 instead shows the value of Ag/At normalized with respect to the parallelism of the data.

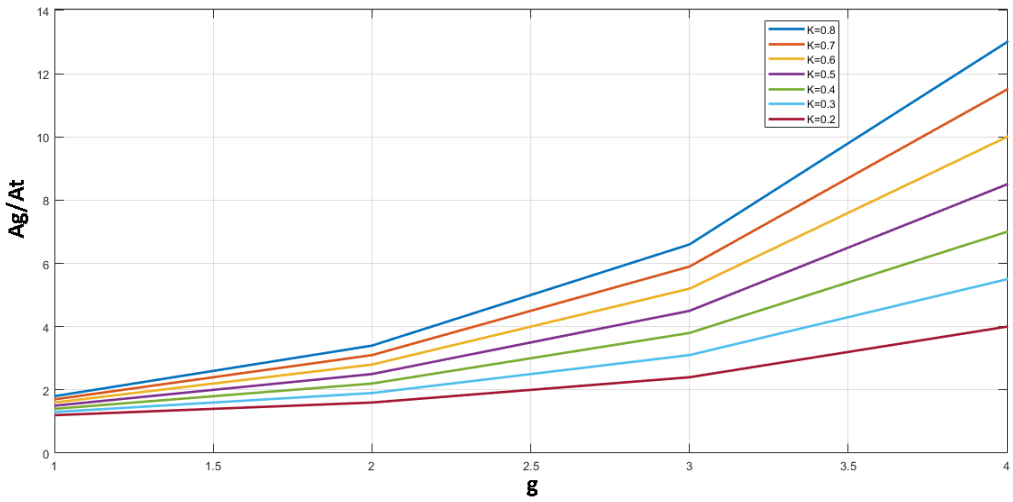


Figure 2.32. Ag/At vs g (2)

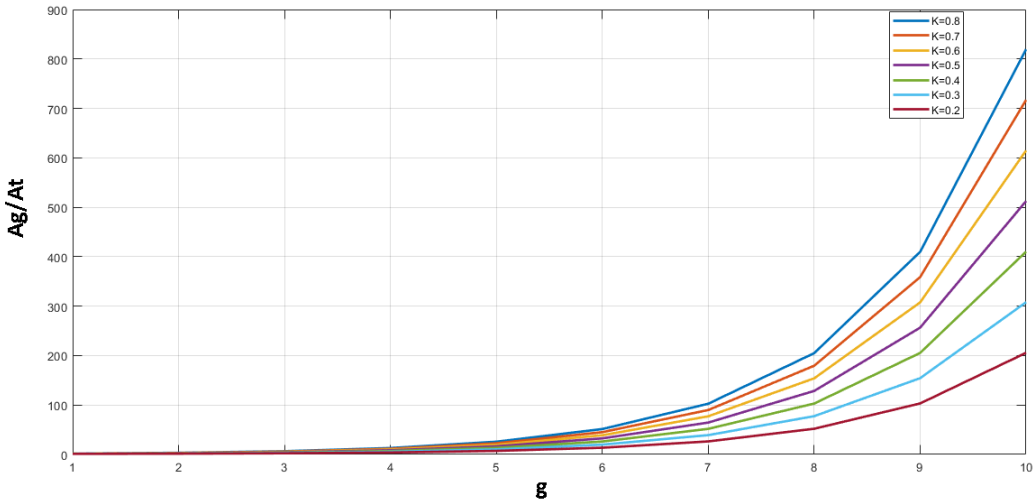


Figure 2.31. Ag/At vs g (1)

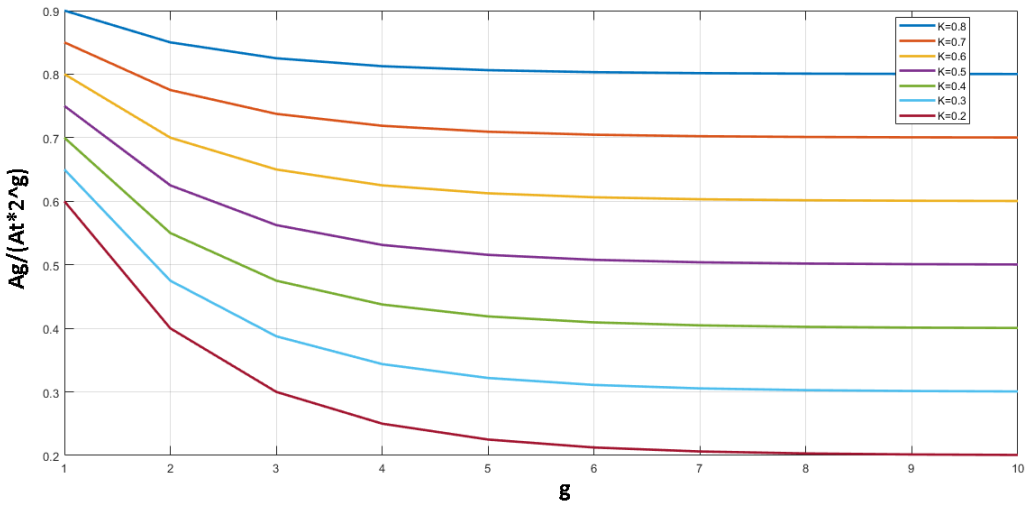


Figure 2.33. Ag/At normalized vs g

2.3.4 Multiple paths

Another method that can be used to increase the number of lines that can be read at the same time is to create more output paths, this technique simply consists in replicating the output paths, therefore it is necessary to introduce new output multiplexers in the memory. Figure 2.34 shows an example with 2 output paths.

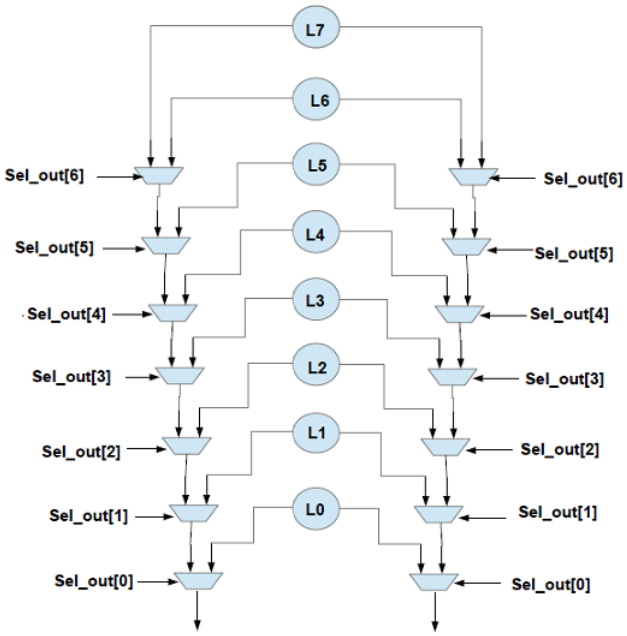


Figure 2.34. Memory reading multiple paths

Figure 2.35 instead shows the same example showing the internal architecture in the case of elementary structures of type A and type A'.

Interconnections of reading signals

In this case there is a linear increase of the interconnections necessary to carry the control signals for reading, these pass from $2^m - 1$ to $N * (2^m - 1)$ with N number of paths.

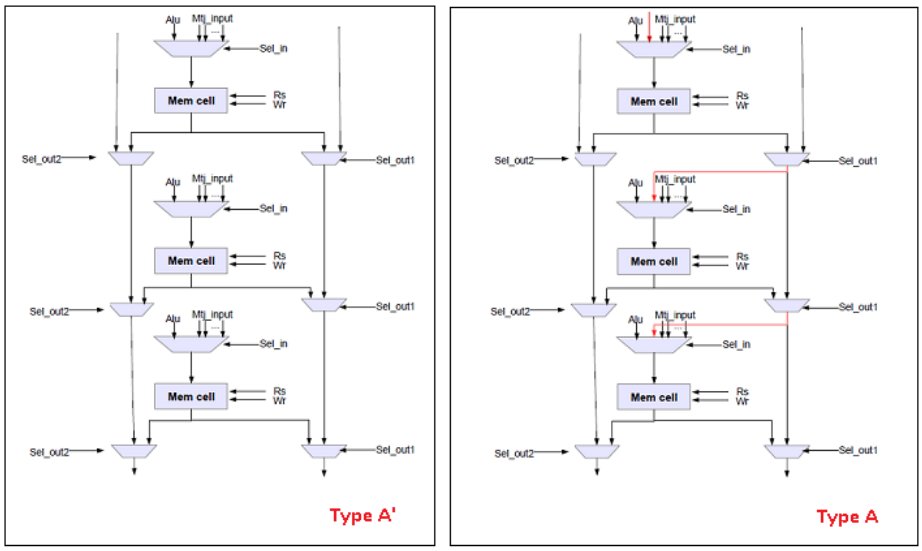


Figure 2.35. Memory reading multiple paths architecture

Area

Differently from what seen on the group memory, it is not necessary to introduce line multiplexers. There is therefore only the need to insert new output multiplexers. Therefore the following equations can be used:

$$An = At * (1 - k) + At * k * N \tag{2.6}$$

$$An/At = (1 - k) + k * N \tag{2.7}$$

From this point of view, the two methods are identical with the same number of possible parallel readings.

Critical path

The critical path remains unchanged at $2^m - 1$

2.3.5 Mixed technique

By making a comparative analysis of the two techniques it can be noted that the division into groups improves the critical path but requires a greater number of interconnections. A mixed technique may be used using both techniques at the same time.

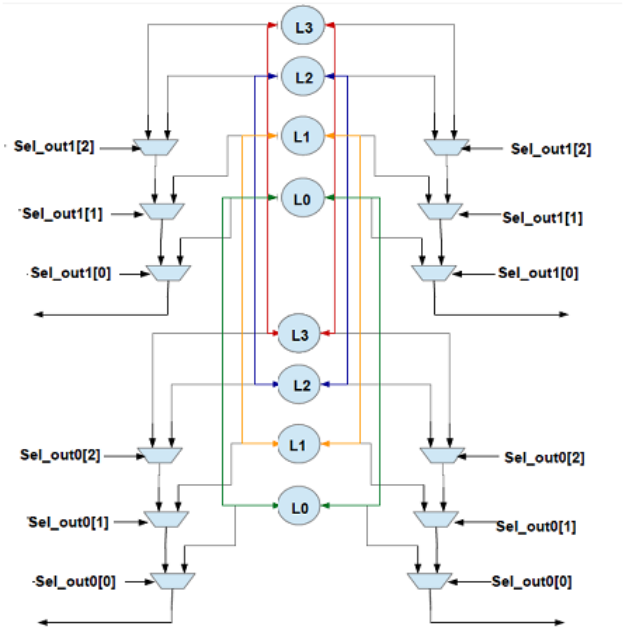


Figure 2.36. Memory reading mixed technique

The two branches are independent from each other, the control circuit can be replicated identical and be used on different branches. the connection between different groups always takes place between the same branches. The figure 2.37 shows the hardware implementation of this technique.

Interconnections

The number is obtained by multiplying 5.3.1 by N. In the example shown in the figure we have $g = 1$ and $N = 2$.

$$NI = [(2^m) * (2^g + g + 1) - 2^g] * N \tag{2.8}$$

Critical path

The critical path is influenced only by the division into groups, therefore the formula associated with it is valid.

Area

The increase due to the subdivision must be multiplied by N:

$$Am/At = (1 - k) + k * (N * 2^g) \tag{2.9}$$

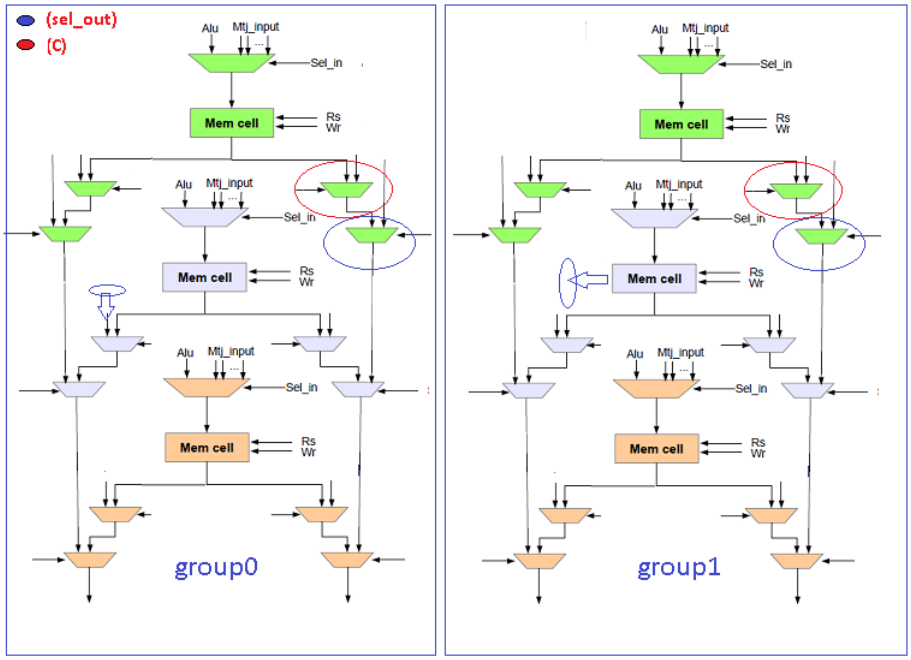


Figure 2.37. Memory reading mixed technique architecture

2.3.6 Critical path

For simplicity of treatment, up to now, it has been assumed that the information flowed from top to bottom. To halve the number of maximum multiplexers to cross, the information can be taken from half of the memory/group. The most identifying bit of the address is used as a selector for the final multiplexer, the remaining bits are encoded and sent the same in both the high and low half. It is necessary to modify the operation of the multiplexers of the lower part, in fact, these must scroll the data with a 0. This technique halves the critical path for free. All the previous techniques are also valid with groups made in this way.

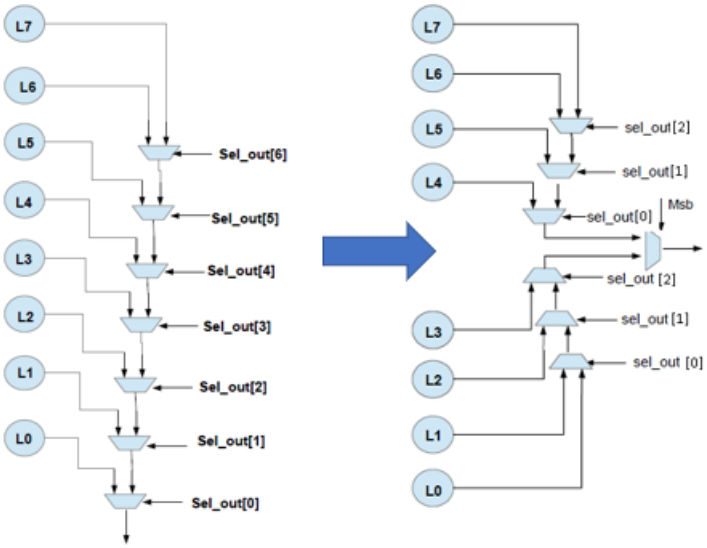


Figure 2.38. Critical path reduction (1)

Another technique to reduce the critical path could be to add by-pass multiplexers. This technique can be combined with the previous one if necessary.

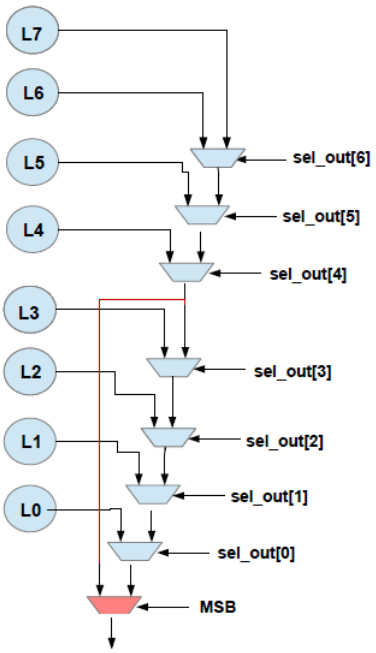


Figure 2.39. Critical path reduction (2)

2.3.7 Methodologies for using transfer between groups

The transfer technique between groups has been seen up to now with the aim of increasing the reading parallelism, however, the architecture shown above makes sense only if the plan is to create a memory that also allows the transfer between cells of the same group. If, on the other hand, the memory does not allow the transfer of the lines to the same group, which therefore allows reading only, it is better to build a tree structure that carries the output data as on the left part of the figure 2.40. There is a third possibility which provides, in addition to the transfer between lines of the same group, the transfer over the entire memory. the simplest example with unidirectional transfer between adjacent lines is shown in the right part of the figure 2.40. to read a line using the transfer between groups, the black colored multiplexer must always be transparent.

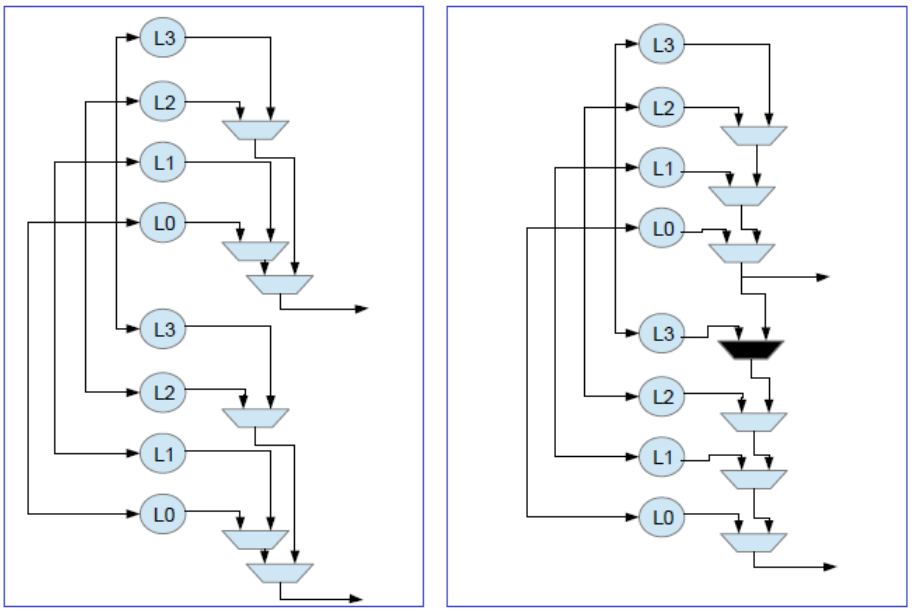


Figure 2.40. Different type of transfer between groups

2.4 Logical/Arithmetic plan

The arithmetic part can be implemented directly inside some memory lines, assuming however that some data can be processed outside the memory, it is useful to define how a logic plan can be structured. Two distinct cases can be identified.

1. Single ALU: In this case the simplest method is to have two groups, the outputs of each group are the inputs of the ALU. As an alternative to a parallel reading, the data can be read and sent individually.

2. Multiple ALU: It is possible to use the techniques seen above to read several lines at the same time, an ALU is assigned to each pair of lines read.

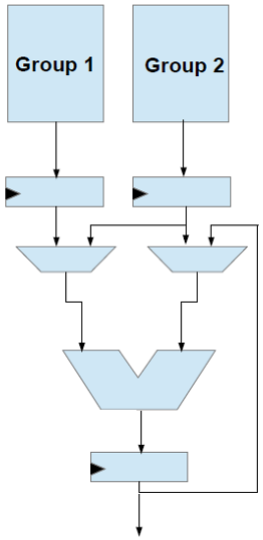


Figure 2.41. Single ALU

2.5 Transfer from the logic plane to the pNML cells

It is sufficient to send the output of the ALU to the input multiplexer of the cells. Transfers from MTJ to pNML and that from ALU to MTJ cannot be run simultaneously on the same line.

2.6 Writing of MTJ memory

The writing of the MTJ memory can be done in the traditional way through mos circuitry. Another possibility is to pass from pNML to MTJ technology by writing the free layer of the latter, in this way it is possible to write the cell directly, for example from the arithmetic logic plane.

Chapter 3

VLIW processor

Very long instruction word (VLIW) processors are designed to increase the level of parallelism in the execution of instructions, complex instructions are often carried out through simpler instructions executable in parallel. The CPU contains multiple functional units, a single VLIW instruction encodes multiple operations, each instruction slot is designed to act on a specific functional unit with specific latencies. The compiler has a fundamental role, the dependency check does not take place dynamically as in other processors but is done during compilation. The compiler therefore has the task of avoiding errors due to data dependencies and providing instructions only to be executed. The main problem therefore becomes the extreme dependence of programs on the compiler, a program optimized for a VLIW processor must almost always be recompiled to work efficiently with future generations. However, these processors offer excellent performance due to the simplicity of execution.

3.1 Transport triggered architecture

Transport triggered architecture (TTA) processors are an evolution of the VLIW ones. The instruction has the task of controlling traffic on the buses, each slot therefore contains a move instruction for a specific bus. Figure 3.1 shows the generic architecture of a TTA processor. The FU registers can be normal or triggered, when a datum is moved on the latter the FU is activated.

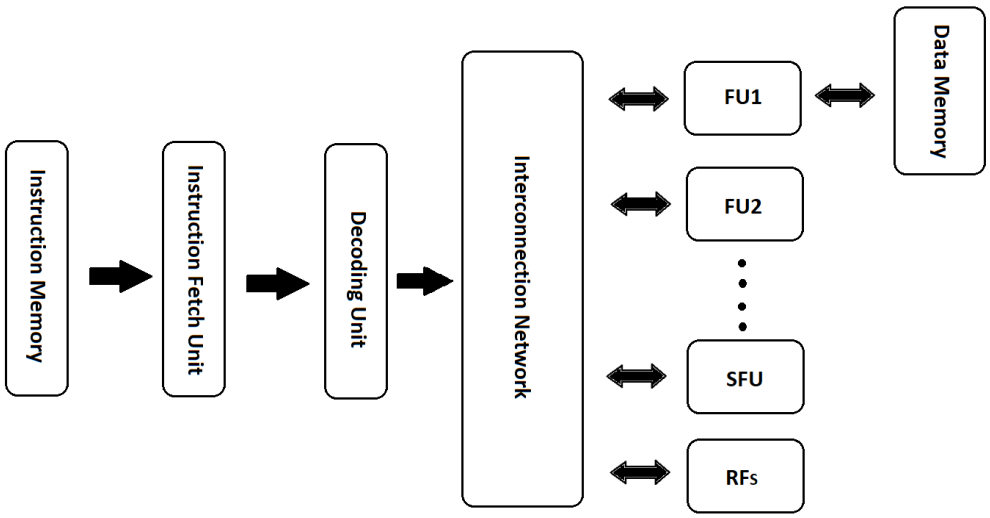


Figure 3.1. General architecture of a transport triggered architecture [17]

3.2 TTA 3D hybrid system

The TTA architecture is an example where the use of a hybrid system could be used with benefits. First of all it is possible to replace the registers with a pNML memory, the latter, using one of the techniques seen, must have a reading parallelism equal to the number of buses. Replaced the registers with a pNML memory, it is convenient to separate the writing buses from the reading ones, this allows to perform writing and reading on it at the same time.

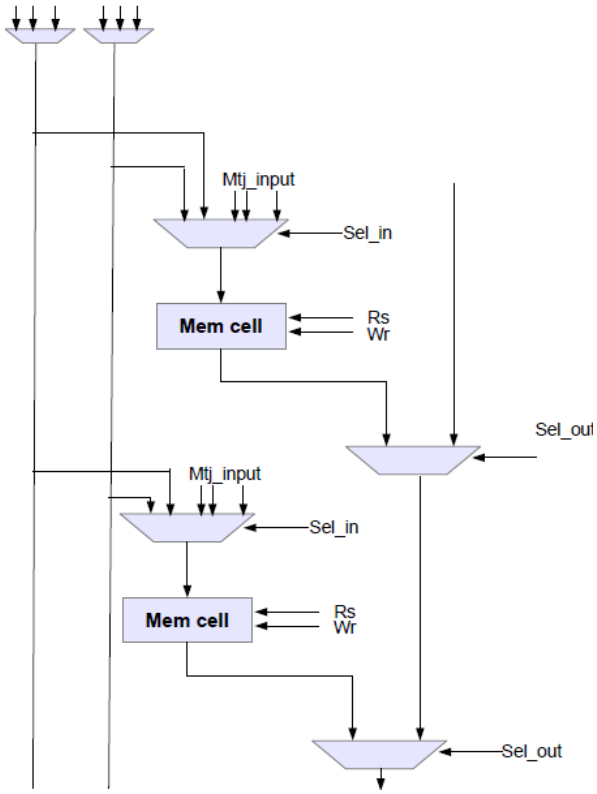


Figure 3.2. Reading and writing circuitry

This change is made possible by the fact that, unlike traditional memory, the reading circuitry is totally separate from the writing one (figure 3.2). The figure 3.3 shows the architecture of the processor designed to be realized through a hybrid system. red is associated with pNML technology, green is MTJ, black is CMOS. Conversion circuits are obviously required. The instruction will consist of a number of slots equal to the total number of buses (write + read). The mtj memory can communicate directly with the pNML memory to write the latter, alternatively, it is possible to use the standard method using a load / store unit. Since even in the MTJ memory the read and write paths are separate, these two operations can be performed simultaneously. It is possible to say that a TTA processor constructed in this way has less conflict between the instructions, this allows a greater paralyzing of the instructions. All the advantages related to technology remain (3D, leakage, area, etc.)

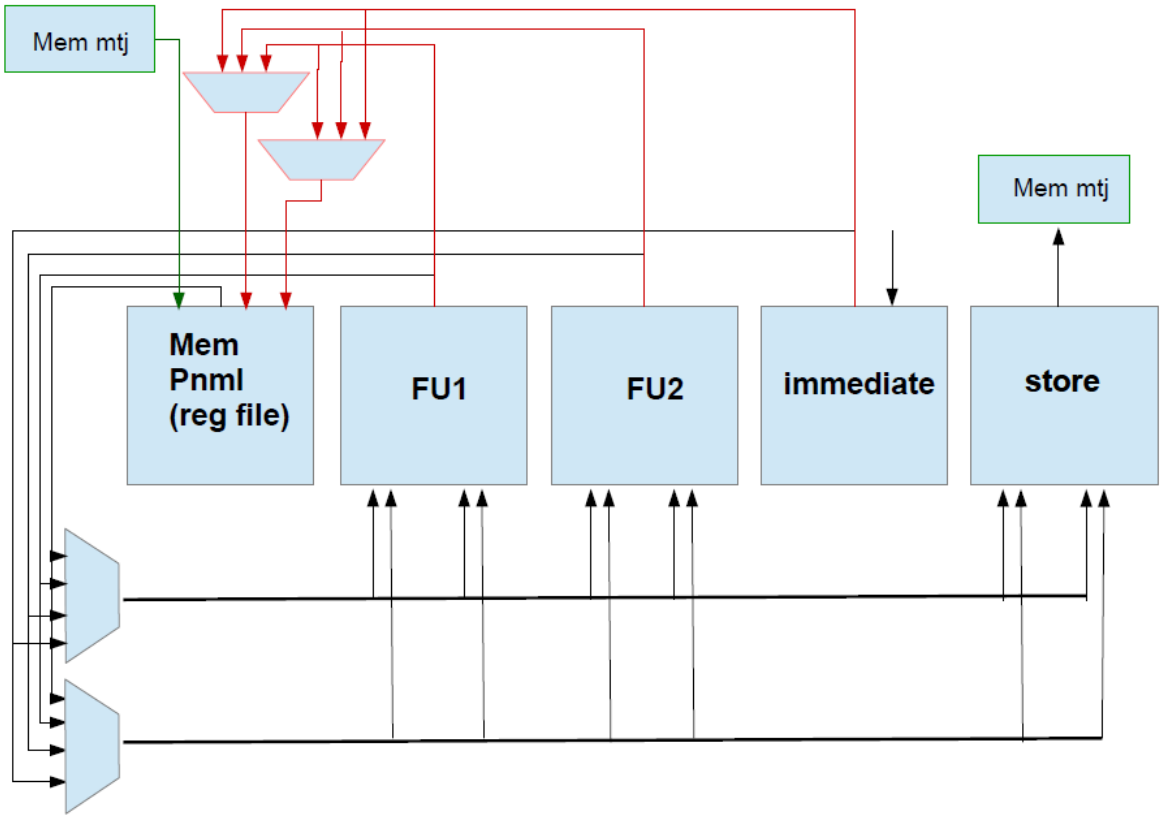


Figure 3.3. TTA made with hybrid system

Chapter 4

Move data memory

The idea is to create a memory that allows the movement of the lines through the techniques seen in chapter two. The purpose of this chapter is to identify the conditions to be respected to carry out operations on it and how these can be obtained. First of all, it is necessary to define the structure that interconnects the lines with each other. Suppose the lines are connected as shown in figure 4.1 and the aim is therefore to move the lines between them through a tree structure. It is convenient to have two distinct branches for downward and upward movements, in this way, it is possible to analyze the two branches separately. The purpose is to understand given N move instructions, potentially parallel on the memory, which of these can be executed and how to execute them. Let's analyze to start only the right branch. Therefore suppose the N movements to descend on the memory, it is necessary to understand if these are possible simultaneously or not. In fact, each movement occupies lines, another movement is possible only if there is a free path to take. The first thing to note is that it is always possible to execute a move instruction if the starting address is greater than the destination address of the other moves. Complications arise when this condition is not respected, in that case we speak of overlapping moves. Another thing to underline is that two overlapping movements can always be performed, this is true because whichever is the position of the starting or ending addresses it is always possible to use two separate lines, this concept will become clearer later.

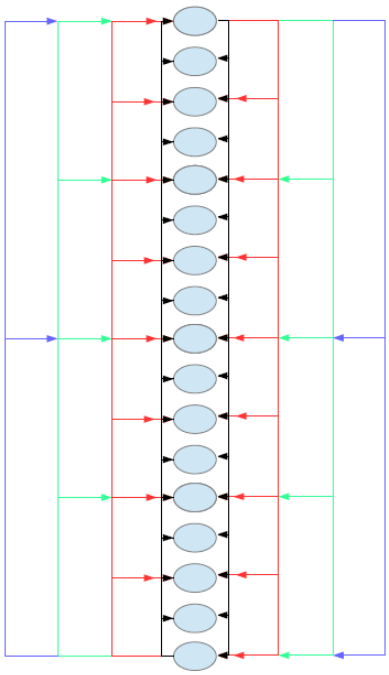


Figure 4.1. Connection of lines

Things get complicated in the case of three movements. By three movements we mean three concurrent movements, that is, those movements in which the paths overlap. The figure 4.3 shows a first example of three overlapping instructions, destination and departure of a generic move are represented with the same color. In the example the instructions in blue and yellow are executed but block the instruction marked with red color. These three movements cannot in any way be executed simultaneously, this happens because if you want to operate three concurrent instructions, it is necessary that at least one instruction involves a third outermost line. Note that to ensure maximum moves it is necessary to take the most external paths possible, we will always use this strategy in the following examples.

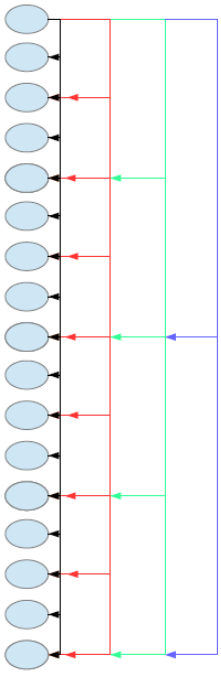


Figure 4.2. Connections for downward movements

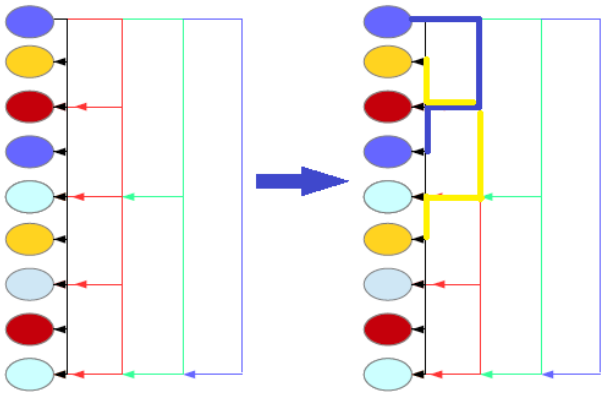


Figure 4.3. Three overlapping movements: example 1

The left side of the figure 4.4 shows an example similar to the previous one, the right side instead shows an example in which the three overlapping

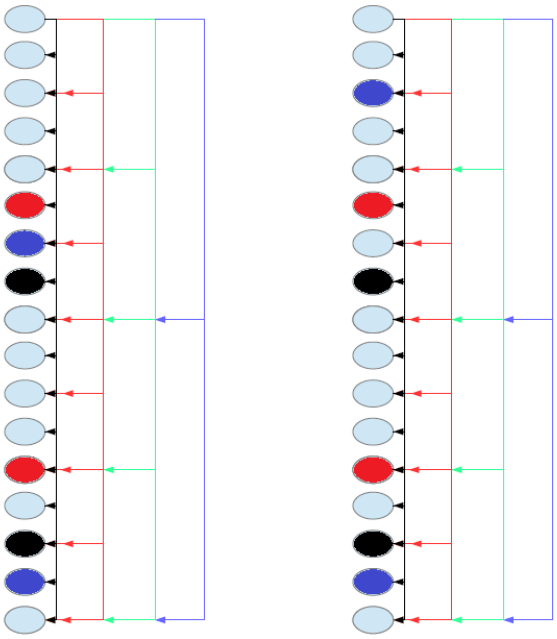


Figure 4.4. Three overlapping movements: example 2

instructions can be performed simultaneously. Still in the figure on the left, it can be seen that the problem is represented by the layout of the departures, their arrangement does not allow any instruction to take the outermost path (green or blue) while this is possible in the example on the right. It is now necessary to introduce the concept of section.

4.1 Section

The figure 4.5 shows an example of a small memory consisting of 9 lines. The number of lines in a tree structure is always made up of a power of two plus one. The concept of section must be defined on the basis of the degree, the degree is associated with the number of instructions that must be verified can be executed in parallel. Sections of degree one and two are not considered because, as previously mentioned, two overlapping instructions do not require any checking.

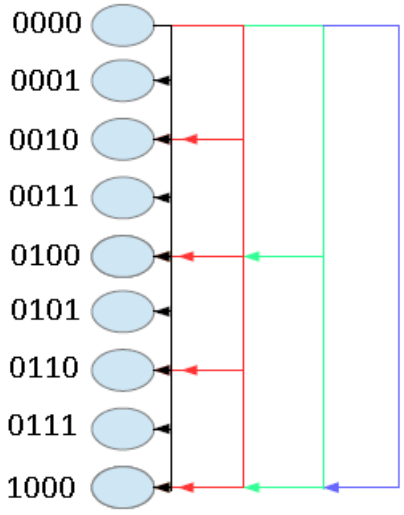


Figure 4.5. Line addressing

4.1.1 Grade three sections

Three lines belong to the same section of degree three if they share all the bits starting from the MSB down to the third LSB. If this condition is verified, the case in which the two remaining LSBs of one of the three addresses are equal to zero must be excluded. In the latter case, the address in question is out of the section. Later this condition will be checked for illustrative purposes in more than one example.

4.1.2 Grade N sections

The concept can be generalized to any degree, N instructions belong to the same section of degree N if they share all the bits from MSB at least significant N bit. As before, if the first condition is met, it must be checked if there is an address with all N LSBs equal to zero.

4.2 Introduction to the general conditions

Through the concept of section it is possible to define the first simple conditions that must be respected in case of overlapping moves. Let's analyze

the simplest case of three statements to begin. From the examples shown above it is clear that the three destination addresses must not be on the same section of degree three, the same goes for departures. Otherwise, an instruction must be postponed. the example below (fig. 4.6) introduces a further problem, checking the conditions of departure and arrival separately is not sufficient, it is in fact necessary to check that the arrivals of two instructions do not hinder the departure of the third. The general conditions will be formalized later.

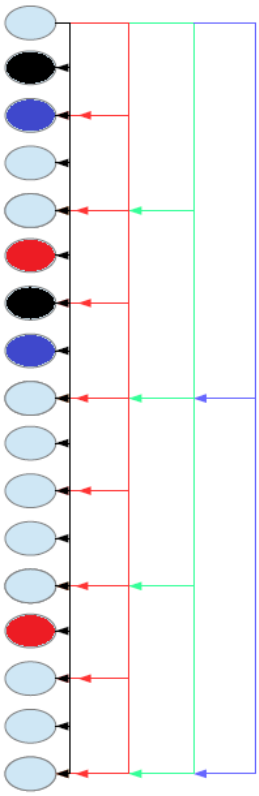


Figure 4.6. Three overlapping movements: example 3

4.3 Ordering of instructions

The graph in figure 4.7 shows an example of random move instructions, the first fundamental step is to have an orderly arrangement of the instructions.

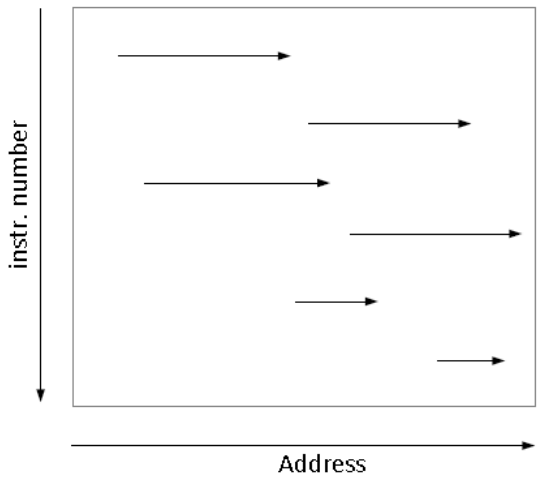


Figure 4.7. Move instructions

The best thing is to have the instructions ordered according to the destination addresses. Ideally, this must be done by the compiler.

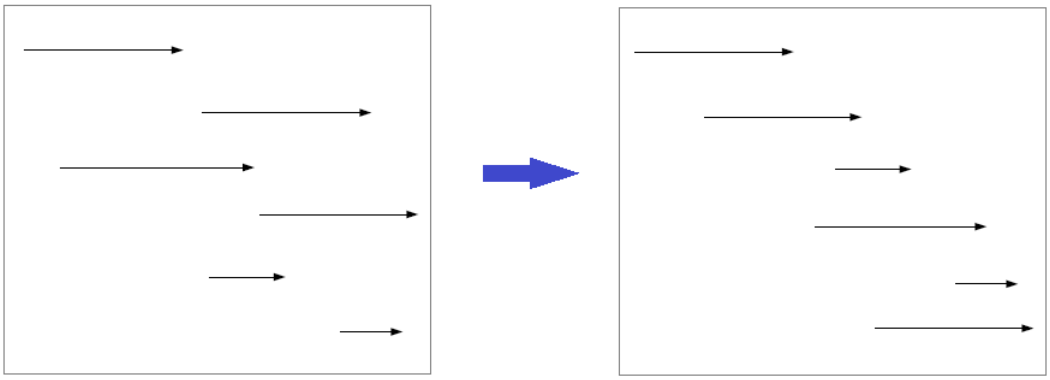


Figure 4.8. Ordering of instructions

4.4 True overlaps and overlaps at the start

4.4.1 Overlaps at the start

Once the instructions have been ordered, it is possible to calculate the overlaps at the start, they must be calculated for each instruction. They are equal to the number of previous instructions that have a destination address greater than the starting address of the instruction in question, one must also be added (instruction itself). Previous instructions mean those that have a lower destination address. Figure 4.9 shows an example consisting of six instructions where the value of the overlaps at the start (SO) is shown on the left.

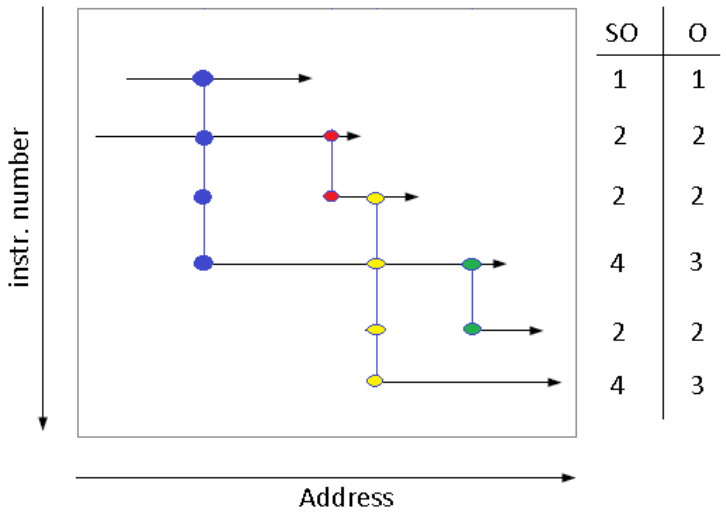


Figure 4.9. overlaps at the start

4.4.2 True overlaps

The overlaps at the start are, as will be shown shortly, a useful parameter for verifying the conditions but they must not be considered in a number equal to the true overlaps. If the true overlaps (O) want to be obtained, the number of really concurrent instructions, it is necessary to modify the overlaps at the start. How this is to be done will be analyzed later. With reference to figure 4.9, note that for the last instruction SO differs from O,

this is due to the fact that the penultimate instruction and the second one are not overlapping.

4.5 Example with $N = 6$

the 33-line memory (32+1) shown in figure 4.10 is taken as an example. on it, the six moves shown in figure 4.11 are intended to be completed.

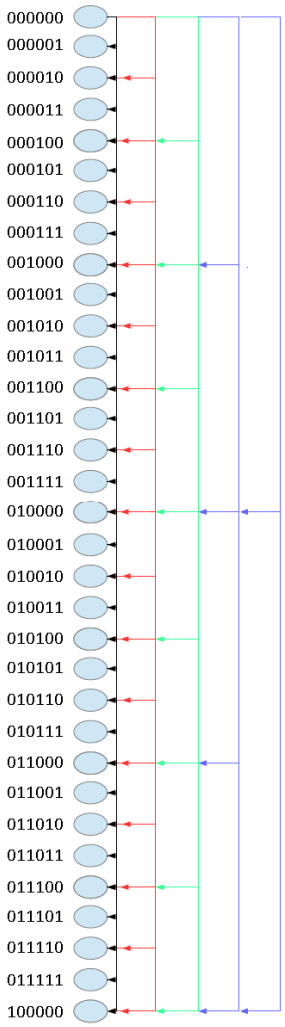


Figure 4.10. Memory under consideration

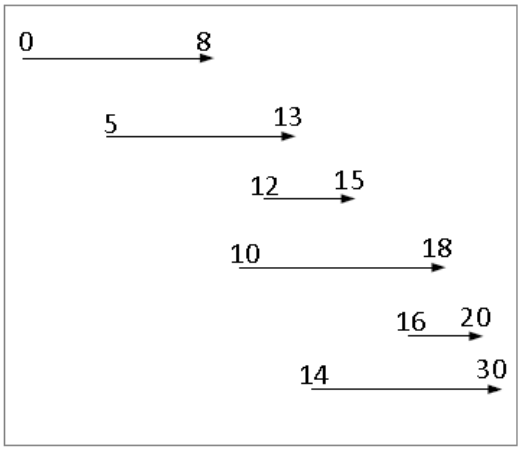


Figure 4.11. Six move instructions

4.5.1 Calculation of SO and O

The first thing to do is to calculate the values of the true overlaps and those at the start. Assuming the method for calculating the true overlaps is known, the result in figure 4.12 is obtained as previously.

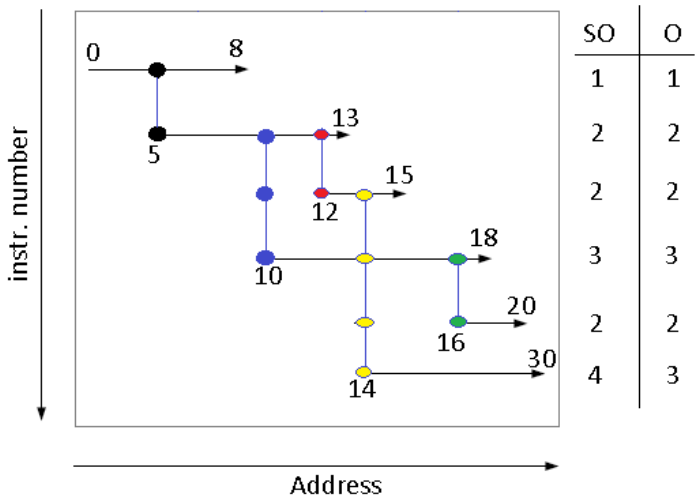


Figure 4.12. Values of SO and O

4.5.2 Execution first three instructions

The first three instructions can be safely executed because they have a number of true overlapping less than or equal to two. The figure below shows the ideal paths for these instructions.

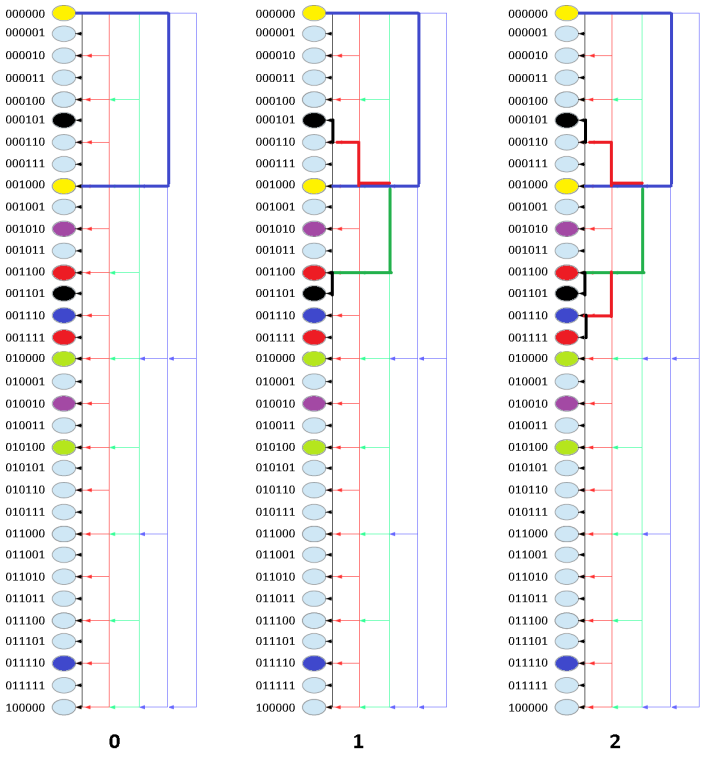
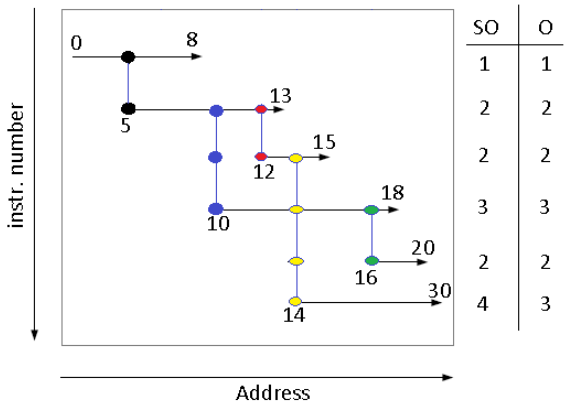


Figure 4.13. paths for first three instructions

4.5.3 Instruction number 3

Instruction number three requires some conditions to be checked as it has $O = 3$ and $SO = 3$. It is necessary to check the conditions for both the arrival address and the departure address. The easiest part is to check that there are no problems with the arrival, the latter must in fact be trivially compared with the arrival addresses of the two previous instructions. The conditions are met because 13,15,18 do not belong to the same grade three section (lower part of figure 4.14).



5	000101	10	001010	10	001010
10	001010	12	001100	13	001101
12	001100	13	001101	15	001111

13	001101
15	001111
18	010010

Figure 4.14. Condition check instruction 3

The conditions at the start are slightly more complex, the starting address of the instruction in question must be compared with both the arrival and departure addresses of the previous two instructions. There are therefore the four possible combinations listed below.

Si	Si	Si	Si
Si-1	Si-1	Di-1	Di-1
Si-2	Di-2	Pi-2	Di-2

Figure 4.15. Conditions at the start O=3 and SO=3

The condition highlighted in red can be eliminated, in fact, if the condition $[S_i D_{(i-1)} D_{(i-2)}]$ is verified, this is implicitly verified. this is true because $D_{(i-2)}$ is always closer to $D_{(i-1)}$ than $P_{(i-2)}$. In summary, in the case of an instruction with $O = 3$ and $SO = 3$, the following four conditions are required:

Si	Si	Si	Di
Si-1	Si-1	Di-1	Di-1
Si-2	Di-2	Di-2	Di-2

Figure 4.16. General conditions O=3 and SO=3

For illustrative purposes, Figure 4.17 shows an example where the condition $[S_i D_{(i-1)} D_{(i-2)}]$ can be tricky. It is even clearer why one of the four starting conditions was discarded.

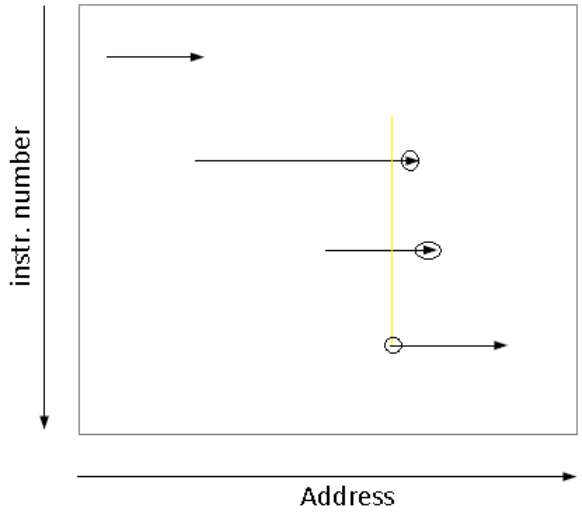


Figure 4.17. $[S_i D_{(i-1)} D_{(i-2)}]$ example

The four final conditions are therefore those shown in figure 4.16. As they all meet the conditions on the section, instruction number three can be

executed. figure 4.18 shows the paths of instruction number three which, as expected, has a free path available to be executed. Instruction number four can certainly be executed as it has $O = 2$.

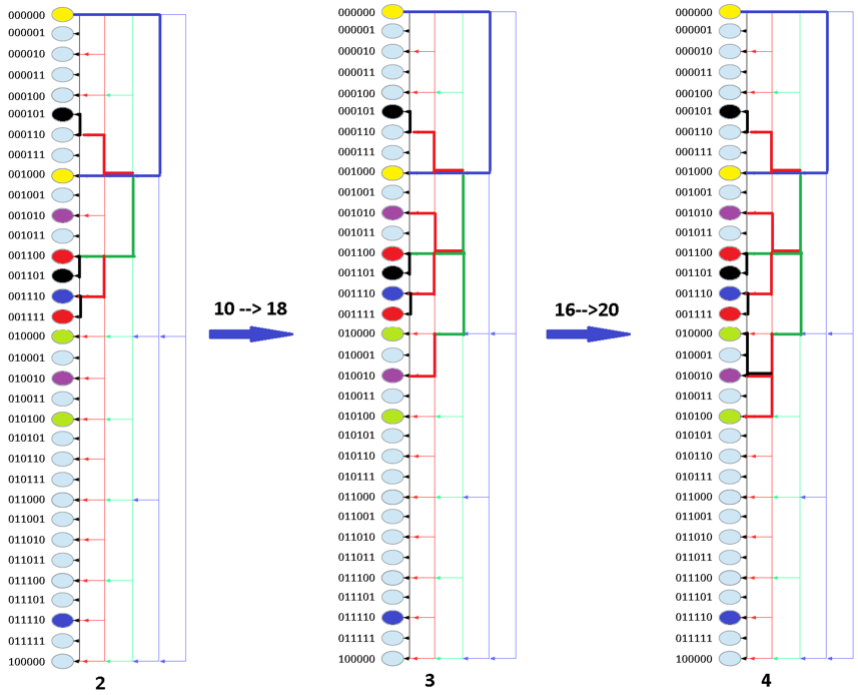


Figure 4.18. Paths for instructions 3 and 4

4.5.4 Instruction number 5

Instruction number 5 and instruction number 3 are different. Although they have an equal number of true overlaps, they differ in the number of overlaps at the start. This does not affect the conditions on arrival but only on those on departure. SO can therefore be ignored for conditions on arrival. The figure 4.19 shows the fact that the first condition is met. On the starting conditions, things get complicated when an $O = 3$ corresponds to a $SO = 4$. The starting address must be compared with all the possible combinations formed by the starting addresses and the destination addresses of the three upper lines. In the case of $SO = 3$, the two upper lines were enough.

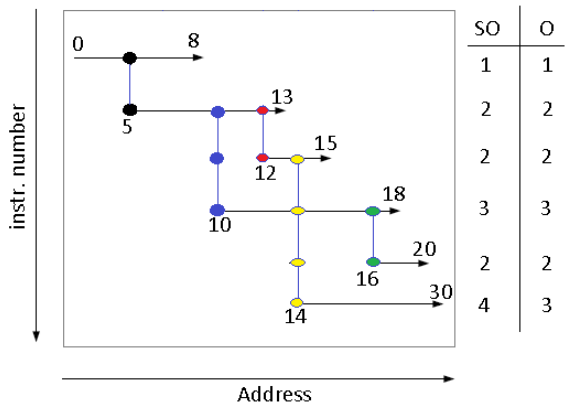


Figure 4.19. Condition on arrival check instruction 5

Si	Si	Si	Si	Si	Si	Si	Si
Si-1	Si-1	Si-1	Si-1	Di-1	Di-1	Di-1	Di-1
Si-2	Si-2	Di-2	Di-2	Si-2	Si-2	Di-2	Di-2
Si-3	Di-3	Si-3	Di-3	Si-3	Di-3	Si-3	Di-3

Figure 4.20. Macro-groups O=3, SO=4

Since there are three addresses to be compared, there are 2^3 macro-groups (fig. 4.20), the base two is due to the fact that the instructions have only start and finish. Each macro-group must then be broken down into elementary groups of three, each elementary group must contain S_i. Each macro-group is therefore formed by 3 elementary groups. There are therefore 3X8=24 elementary groups in total (fig. 4.21). However, each elementary group is repeated twice, once with the remaining 'bit' in S and once with the remaining 'bit' in D.

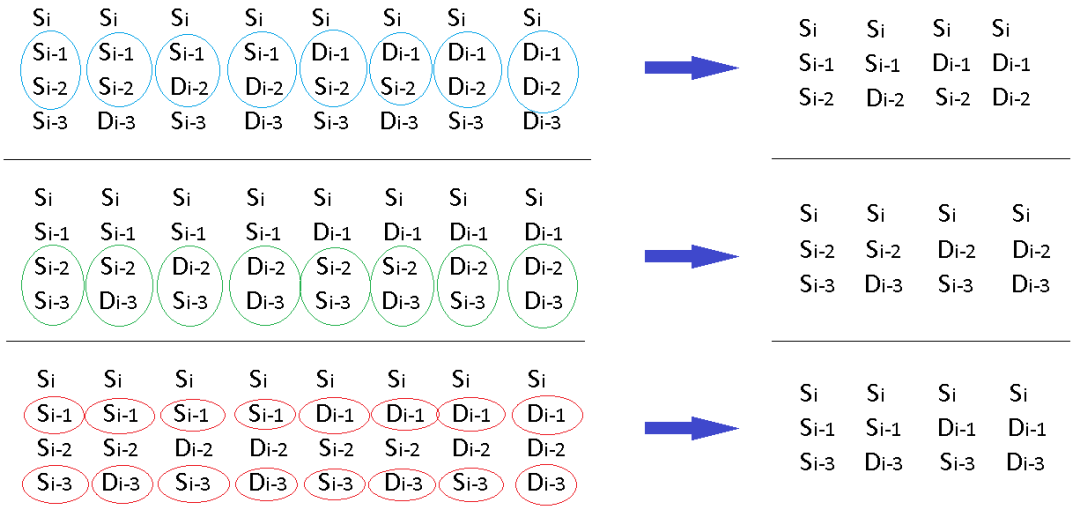


Figure 4.21. Elementary group O=3, SO=4

Starting from the 24 initial conditions, once the repetitions are eliminated, the 12 shown in figure 4.22 are obtained.

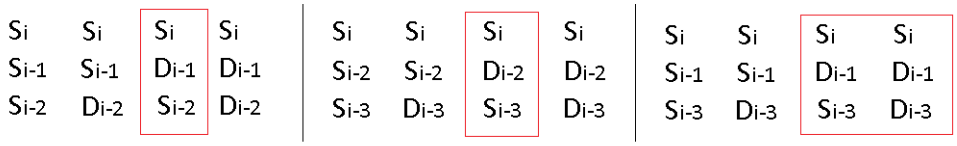


Figure 4.22. Elementary group O=3, SO=4 without repetitions

As previously done it is possible to eliminate some conditions. In fact, thinking about the 12 conditions it is possible to pass to 8, those circled in red can be eliminated. All the conditions that contain $D_{(i-1)}$ can be deleted because $D_{(i-2)}$ is definitely the closest. A similar argument can be made for the couple $D_{(i-2)}$ and $S_{(i-3)}$, the latter is always preceded by $D_{(i-3)}$. In the example under examination, the conditions are met and it is also possible to execute this instruction in parallel. Figure 4.23 shows the path of this last move instruction.

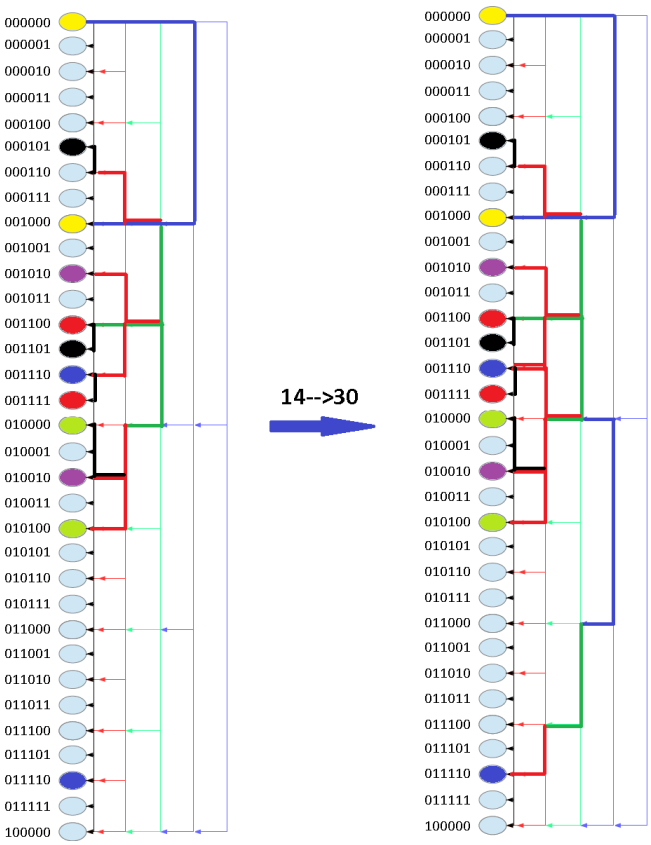


Figure 4.23. Paths for instructions 5

4.6 Calculation of true overlaps

It was seen in the previous example that the conditions on arrival depend on the value of O while those on departure depend on both O and SO. From a theoretical point of view any $O - SO$ pair is allowed, this is shown in figure 4.24. It now remains to understand how the values of true overlaps can be calculated. The calculation of the overlaps at the departure is relatively simple because it is enough to compare the departure with the previous arrivals, the calculation of O is much more complicated. Four methods have been identified for the calculation of O:

- **Exact method**
- **Approximate method by excess**

- **Approximate method by semi-weighted excess**
- **Approximate method by weighted excess**

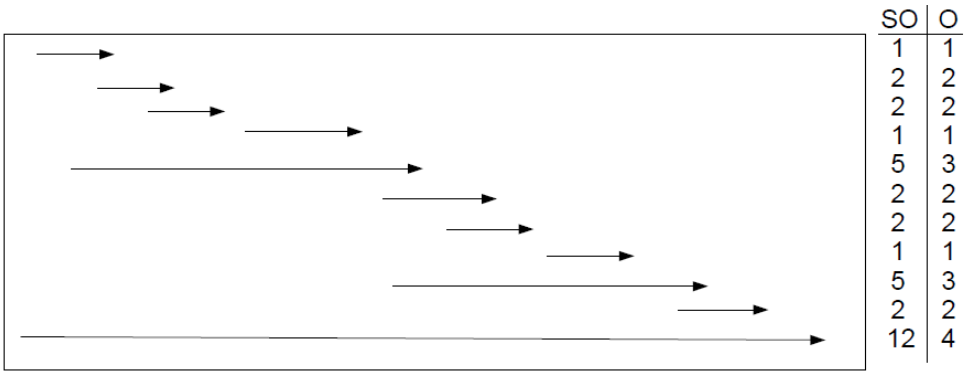


Figure 4.24. Move example

4.6.1 Exact method

If true overlaps are to be calculated precisely, the only working method is to make a sort of integral superposition. Proceed by creating for each instruction a vector of 0 and 1, the value 1 is associated with all the addresses between the start and the arrival. Once all the vectors have been created, the sum of all the bits of the previous vectors with the one under examination must be made, this must be done for each position that contains a 1 in the instruction under examination. Then proceed as in the figure 4.25, at the end the maximum value is taken. If this check is done in the compilation phase, it should be preferred to the others, the following strategies are simpler but have errors. They are recommended in hardware implementations.

4.6.2 Approximate method by excess

The new value of O is calculated in relation to the value SO and all previous values of O. The new value of O is simply the minimum between the current SO and the maximum of the previous values of O increased by 1. It is always wrong to excess, it is the simplest method.

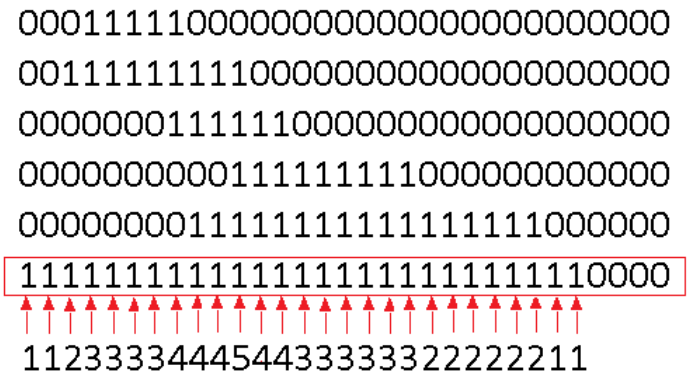


Figure 4.25. Exact method

4.6.3 Approximate method by semi-weighted excess

The method can be slightly modified by checking not all the previous values of O but only in a number equal to the value of SO. The methods are almost equivalent if there aren't really many instructions.

4.6.4 Approximate method by weighted excess

The approximation method seen above is based on the fact that a new instruction cannot in any case add more than one new true overlap than the worst case. However, the method is overly cautious because it is not necessary to check all the previous values but it is possible to stop earlier. If the generic pair $SO - O$ is taken, the value of O is associated with a number of instructions equal to SO. It is therefore possible to proceed in this way: check the value of SO_i to which it is necessary to associate O_i at that moment, let's suppose $SO_i = K$. Proceed by adding all the previous SO values until the condition $1 + SO_{(i-1)} + SO_{(i-j)} => K$ is satisfied. The greater value between $O_{(i-1)}$ and $O_{(i-j)}$ is taken plus 1 and the new value of O_i is the minimum between SO_i and the value just found. The figure 4.26 below shows the results of applying the different methods to the initial example.

SO	O
1	1
2	2
2	2
1	1
5	3
2	2
2	2
1	1
5	3
2	2
12	4

Exact

SO	O
1	1
2	2
2	2
1	1
5	3
2	2
2	2
1	1
5	4
2	2
12	5

Excess

SO	O
1	1
2	2
2	2
1	1
5	3
2	2
2	2
1	1
5	4
2	2
12	5

S.W. Excess

SO	O
1	1
2	2
2	2
1	1
5	3
2	2
2	2
1	1
5	3
2	2
12	4

W. Excess

Figure 4.26. Comparison between methods

4.7 Conditions at the start: general case with $O = 3$

It has been seen that the conditions on arrival depend only on O. The conditions on the start depend on the value of SO and on the value of O. It has been seen that for the pair $(SO = 3, O = 3)$ there are a maximum of 4 conditions which can be reduced to 3. for $(SO = 4, O = 3)$ there are 12 maximum conditions which become 8. The maximum number of conditions for any $(SO = N, O = 3)$ pair can be found by a mathematical calculation. To have the reduced value it is necessary to follow the steps seen in the example. Given $SO = N$, the number of macro-groups is given by 2^{N-1} Each macro-group is composed of N elements, being S_i fixed the other $N-1$ items contain the two remaining places. The number of elementary groups for each group is therefore given by the binomial coefficient $N - 12$. The number of total elementary groups is equal to:

$$2^{N-1} * N - 12 \tag{4.1}$$

Since 3 values are always kept fixed, for each elementary group we will have 2^{N-3} repetitions, the number of maximum conditions is therefore equal to:

$$Cmax = \frac{2^{N-1}}{2^{N-3}}N - 12 \tag{4.2}$$

from which:

$$Cmax = 2^2\frac{(N-1)!}{2 * (N-3)!} \tag{4.3}$$

The figure below shows the value of $Cmax$ for different values of SO

N=3 --> Cmax=4
N=4 --> Cmax=12
N=5 --> Cmax=24
N=6 --> Cmax=40

Figure 4.27. Cmax O=3

4.8 Conditions for $O > 3$

4.8.1 Conditions on arrival

For O greater than three it is necessary to check the conditions on all the sections with a grade starting from the value of O down to grade three. taken $O = N$ it is necessary to check: the grade section N with the destination address and the previous $N - 1$ destinations, the grade section $N - 1$ with the previous $N - 2$ destinations. The method proceeds in this way up to the grade 3 section.

4.8.2 Conditions on the start

Also in this case, it is necessary to check the conditions on all the sections with grade starting from the value of O down to grade 3. Having taken the generic pair $(SO = N, O = M)$ it is necessary to verify the conditions for all pairs $(SO = N, O = M)$, $(SO = N, O = M - 1)$. .. $(SO = N, O = 4)$, $(SO = N, O = 3)$ where the degree of the section corresponds to the value of

O. For the generic pair $(SO = N, O = M)$ it is also possible to calculate the maximum number of the conditions. The number of macro-groups depends on the value of SO, so it is equal to 2^{N-1}. Also in this case, each macro-group is made up of N elements, of which S_i is fixed. The size of the elementary group now depends on O, the remaining $N-1$ elements therefore contain $M-1$ places. The number of repetitions will be equal to 2^{N-M}. The following equation is therefore obtained for the pair $(SO = N, O = M)$:

$$Cmax = 2^{M-1}\frac{(N-1)!}{(M-1)!(N-M)!} \tag{4.4}$$

Since all the pairs shown must be respected, the total number $Cmaxtot$ is given by the sum:

$$Cmaxtot = \sum_{k=3}^{M} 2^{K-1}\frac{(N-1)!}{(K-1)!(N-K)!} \tag{4.5}$$

4.9 Algorithm for calculating instructions path

In the previous sections has been studied the methodologies necessary to understand when several instructions are executable at the same time, once the executable instructions have been selected it is necessary to identify the paths for each single instruction.

4.9.1 Ideal path calculation

First of all it is necessary to identify the ideal path to follow for each move instruction, however it is not always possible to itulize the latter. Let S be the starting address and D the destination address. The first step is to deprive the destination address of all 1s with the exception of the most significant one, D 'is the address thus obtained. Some examples follow.

D	D'
010010	010000
001111	001000
001000	001000
000011	000010

Table 4.1. D' calculation

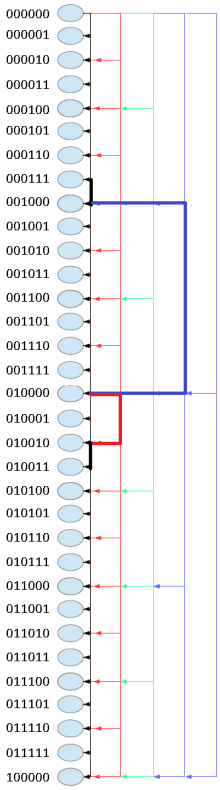


Figure 4.28. Instruction path

The paths leading from address S to D' must therefore be found. Starting from the value of S, the position of the first bit at 1 is identified starting from LSB. If the position of the bit is position 0 it will be necessary to initially take the innermost line, proceed by adding 2^0 to S. If the position of the first 1 is position 1, proceed by adding 2^1 and taking the second innermost line (the one that connects multiple addresses of two). This is done until the value of D' is reached. An example follows in which the move $7 -> 19$ wants to be executed. ($D = 010011$, $D' = 010000$, $S = 000111$), with reference to figure 4.28.

$$S' = 000111 + 2^0(blackline)$$
$$S' = 001000 + 2^3(bluline) = D'$$

Table 4.2. S' calculation (1)

Where S' are all the intermediate addresses between S and D. The last thing to do is to find the path that leads from D' to D. D' must reach the value D by adding 1 in all positions starting from the most significant of the ones present in D.

$$S' = 010000 + 2^1(redline)$$
$$S' = 010010 + 2^0(blakline) = D$$

Table 4.3. S' calculation (2)

Using this procedure it is possible to obtain all the intermediate addresses between the starting one and the destination one, the table below summarizes all the values of S 'for the previous example, also S and D are included.

S'
000111
001000
010000
010010
010011

Table 4.4. S' values

The flow diagram in figure 4.29 describes the algorithm to be developed. The aim is to find all the values of S' for a move instruction, this strategy allows to find all intermediate addresses of the ideal path. K is the one-bit position in D '.

4.9.2 Non-ideal paths

The method just seen allows to find for each instruction the ideal path that leads from S to D. If more instructions to be executed want to access the same line, it is necessary to find an alternative path for an instruction. given N instructions to be executed in parallel, proceed as follows:

1. The ideal path is found for each instruction, therefore all the values of S 'are known for each instruction.

2. Obtained the S' it must be verified that two consecutive values present in one instruction are not present in any other instruction. If this is true, ideal paths can be used for all move instruction.

3. If two or more instructions share a pair of consecutive values of S', it is necessary to split one of the paths into smaller paths in sub powers of two.

The table summarizes the values of S 'for two instructions (6 to 20, 5 to 24). it can be seen that both instructions, ideally, would like to use the line leading from 001000 to 010000.

S1'	S2'
000110	000101
001000	000110
010000	001000
010100	010000
	011000

Table 4.5. S1' and S2' before division

As previously mentioned it is therefore necessary to divide the path into smaller paths, the table below summarizes the final values. The subdivision was made on the values S1'

S1'	S2'
000110	000101
001000	000110
001100	001000
010000	010000
010100	011000

Table 4.6. S1' and S2' after division

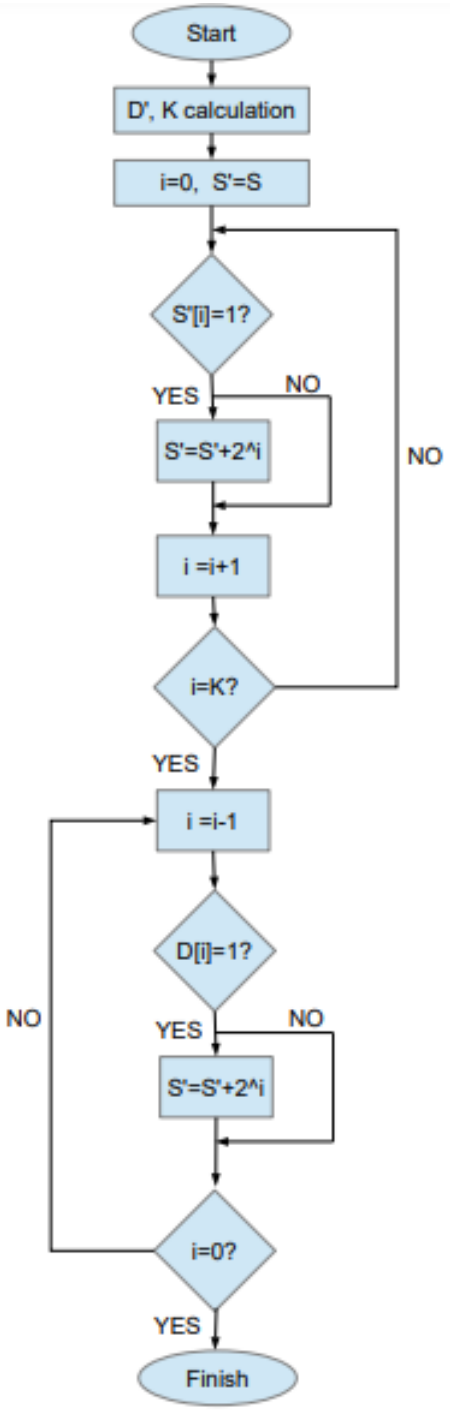


Figure 4.29. Flow diagram S' calculation

Chapter 5

A new processor with move data memory

The goal is to create a processor that uses move data memory, some memory lines will have to be designed with additional circuitry capable of performing operations on the data. From this perspective, it is possible to imagine some memory lines like functional units. Operations are therefore data moves from one line to another.

5.1 Role of the compiler

The compiler has a fundamental role, in addition to ensuring that there are no data dependency errors, its task is to verify if the moves can be done in parallel and which paths should be used. In this regard, it is possible to use all the techniques seen in the previous chapter. Once the conditions have been verified and the paths to be used have been identified, using the algorithm in figure4.29, it is possible to arrive at instructions such as those in figure 5.1. The single instruction is composed of several move instructions to be executed in parallel, each move instruction is composed of the starting address and the increments from which all the intermediate and final addresses are obtained. The increments are coded in order to decrease the number of bits of the single instruction. The value of X is a design parameter, any instruction that cannot be described by X increments must be divided.

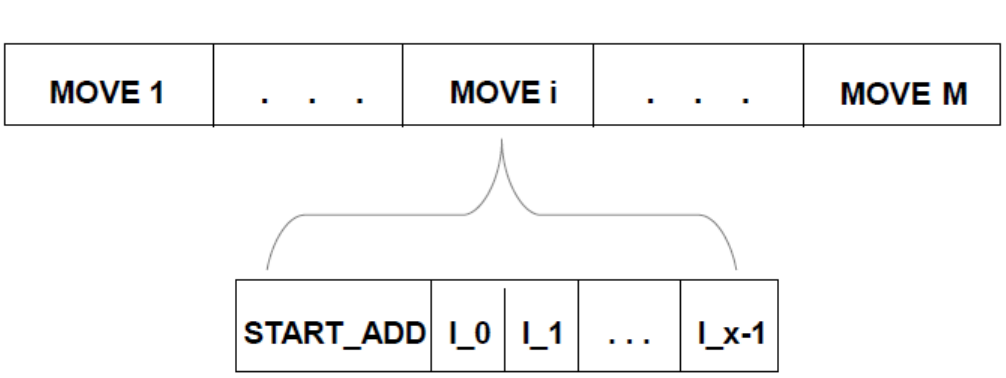


Figure 5.1. Processor instruction and composition of a single move

5.2 General scheme

The general scheme of the processor is visible in figure 5.2, the instruction is sent in input to a block that generates the control signals for the memory.

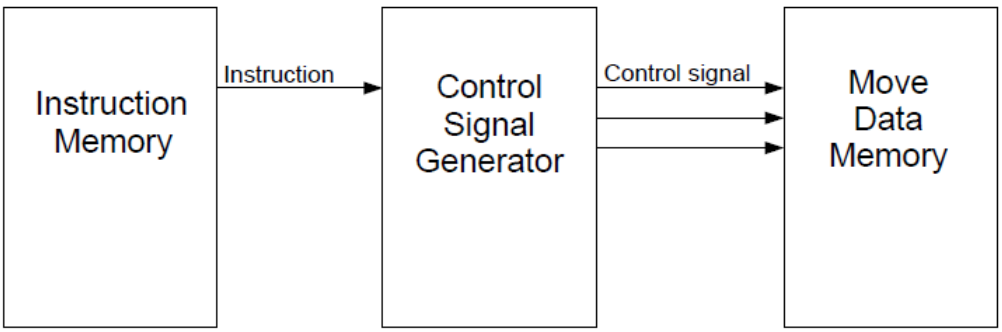


Figure 5.2. General scheme of processor

5.3 Hardware architecture of memory

It is necessary to implement a memory architecture that allows the movement of lines. The architecture must allow the passage between the various ways, a data that is using the innermost line must be able to pass to the outermost ones and vice versa. The figure 5.3 shows an implementation on a 9-line memory. The black colored multiplexers allow data to be scrolled from one cell to the lower adjacent one. Those colored in red connect all cells with multiple addresses of two. the green ones connect multiple addresses of four and so on. The multiplexers in pink are the multiplexers that select the input of the cell to be written. Finally, those in yellow have the task of moving the data into more internal or external lanes.

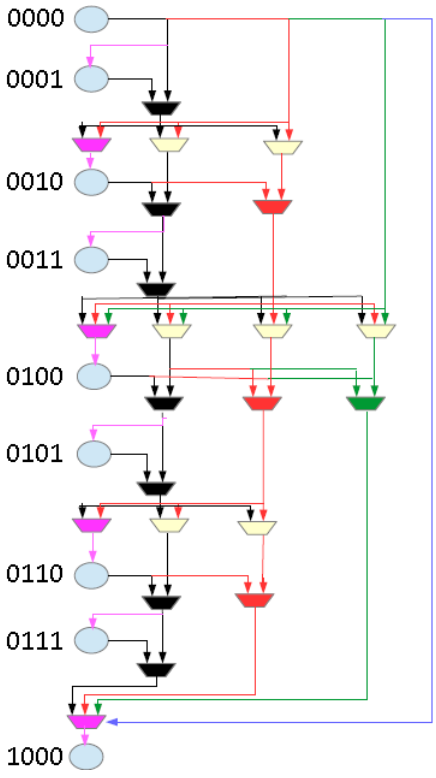


Figure 5.3. Hardware architecture for one direction line movement

5.3.1 Movements in both directions

The architecture seen above allows the lines to scroll only from top to bottom. If the aim is to have a memory capable of moving the lines in both directions, it is necessary to build a mirror structure that allows movements in the opposite direction (fig. 5.4). In this case it is necessary to insert, for each cell, an additional multiplexer which selects whether the data to be written (fig. 5.5). Each cell can in fact receive the data from both the upper and the lower part.

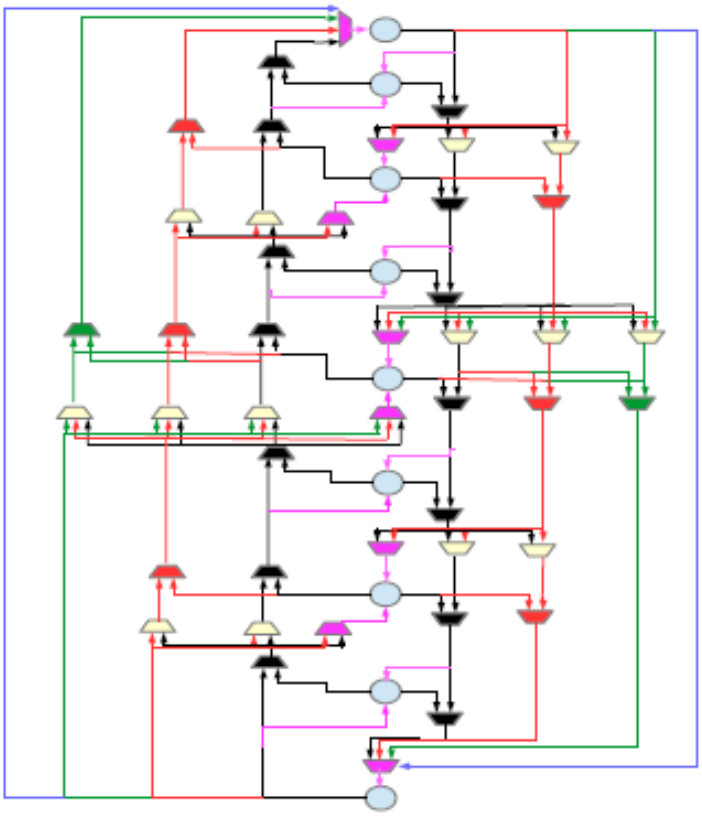


Figure 5.4. Hardware architecture for both directions line movement

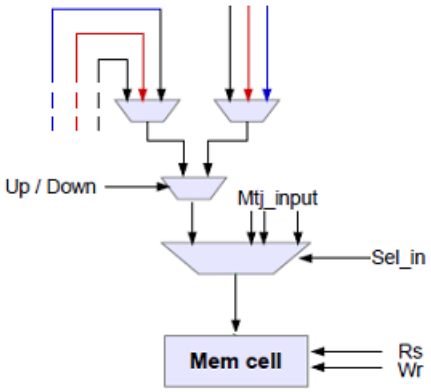


Figure 5.5. Up/down multiplexer

5.3.2 Memory block scheme

Memory can be seen as a succession of blocks (figure 5.6). The sub-blocks CT (traffic control) contain the output multiplexers of the cells, these have the function of choosing between the contents of the cell and the output of the previous block, in figure 5.3 they are black, red and green. The sub-blocks ST (traffic switches) have instead the task of moving the data on the different paths (yellow). Blocks C contain the memory cells and the input multiplexers (pink). The control signals are therefore: $CTsel$, $STsel$, mux_insel and WE (Write Enable). This last signal enables writing of the memory cell. All the multiplexers of the sub-blocks CT output the value of the cell when the selector is set to '0'.

5.4 Control signal generator

A more internal view of the signal generation block is visible in figure 5.7, for each move a block is needed that generates the signals for the memory, the final signals are obtained through an OR block between the signals of the various moves.

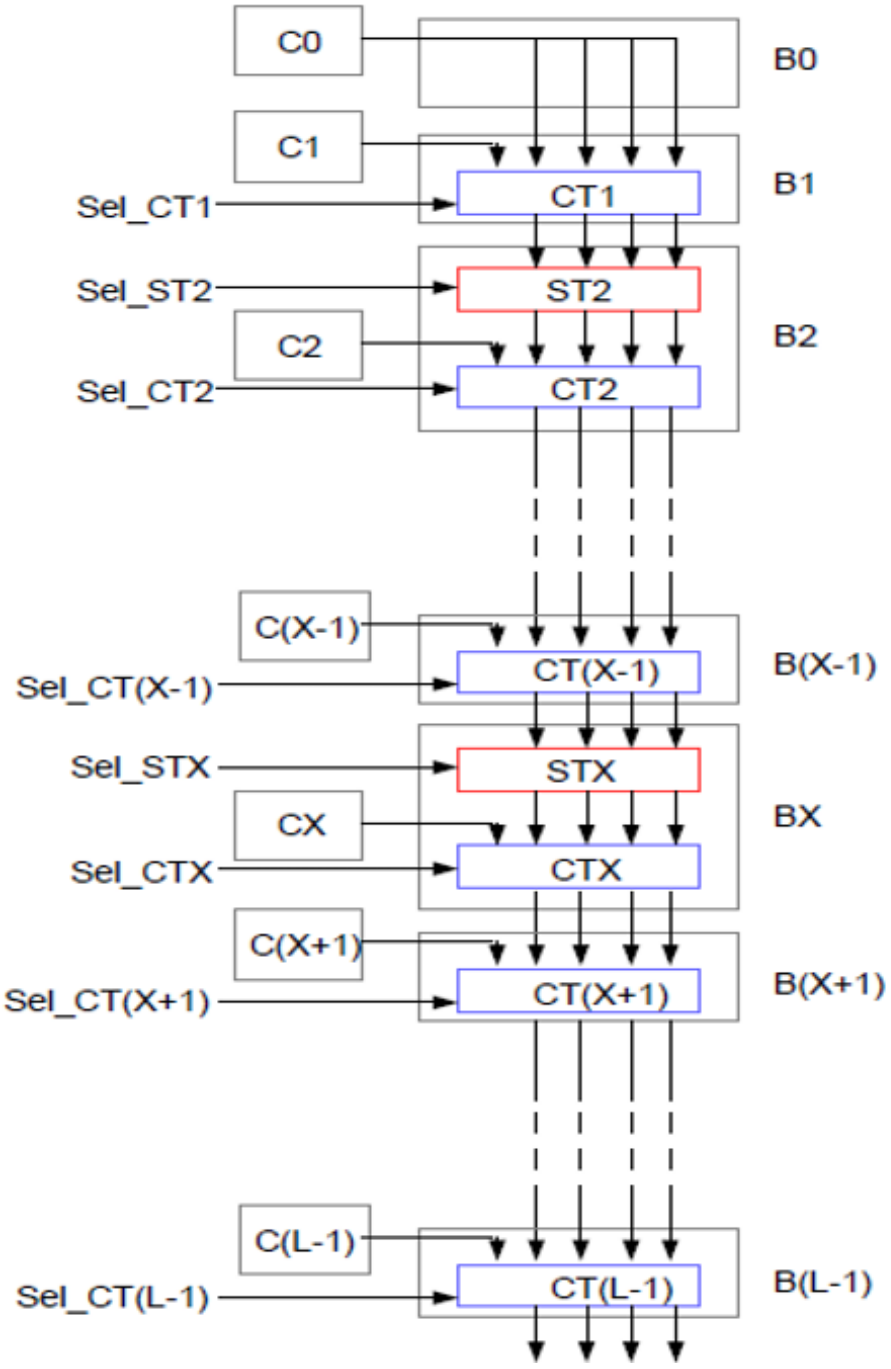


Figure 5.6. Memory block diagram

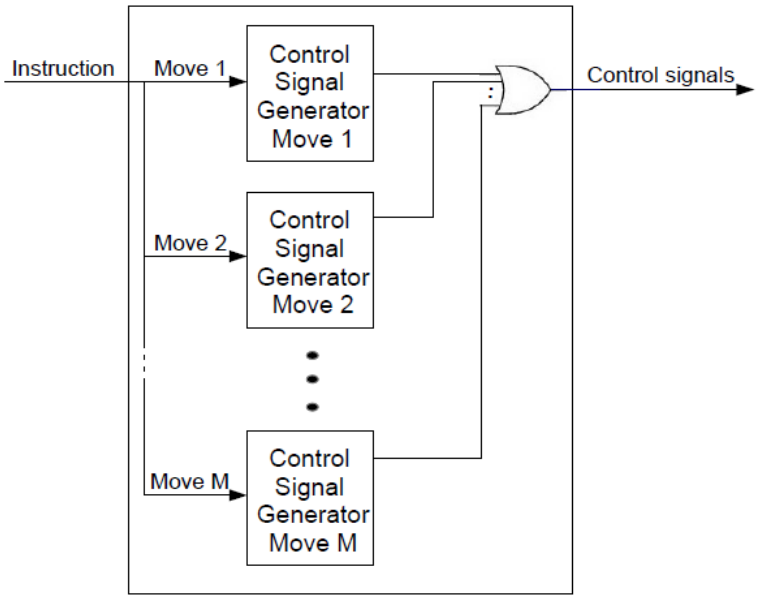


Figure 5.7. Control signal generator

5.4.1 Single move control signal generator

A high-level representation of the block in question is shown in figure 5.8, the block must take in input a move instruction and generate all the necessary control signals. There are the following blocks:

- **Increments Decoder:** Its function is to decode the increments so that they can be sent to a summing circuit. A value equal to 0 contained in the generic I_i increment must be associated with the value 0, any other value other than zero of I_i generates a value equal to $2^{(I_i-1)}$
- **Last Increments Seeker:** Starting from I_0 it finds the last increment different from 0, this block is fundamental for generating the commands of the input multiplexer of the destination cell. Through the increase it is in fact possible to obtain the path from which the data to be saved arrives. This block, depending on how the input multiplexers are made, may have to be followed by a decoding block.

- **Addresses Calculator:** It is a block composed of adders, its function is to generate all the addresses involved in the move. All intermediate addresses are sent to the $STsel$ and $CTsel$ signal generation block. The destination address is used to generate the write enable signals.

- **Decoder:** Is a classic block of line decoders which, given the address, sets the corresponding signal to '1'.

- **CT and ST Calculator:** It generates the two signals starting from the intermediate addresses and from vector I. Their behavior will be analyzed in detail below.

- **And block:** Is a block that generates the vector mux_insel, it is formed by AND gates in which the vector WE is used as a mask. It must be designed in such a way that the output signal has the correct number of bits.

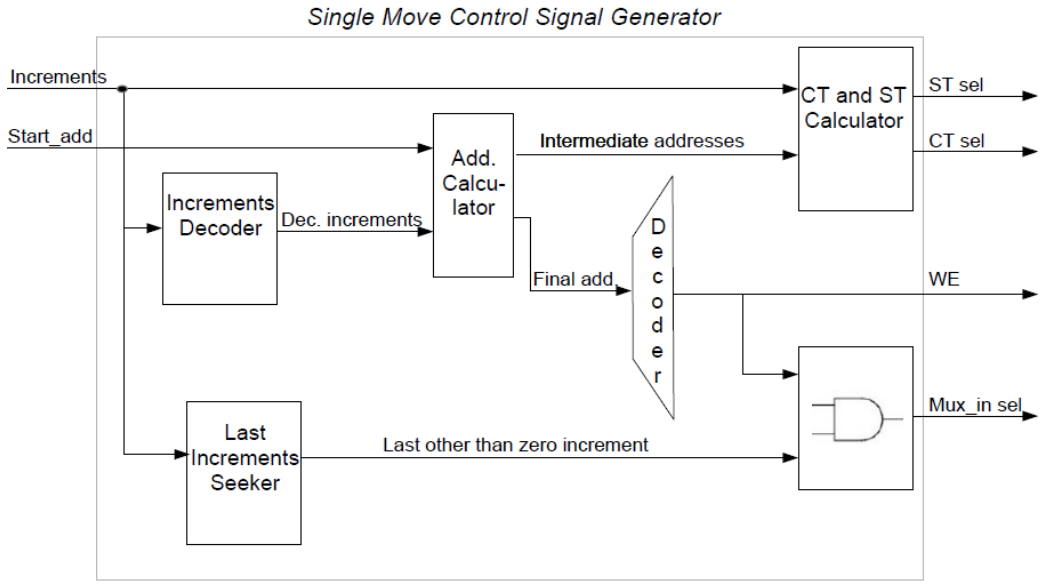


Figure 5.8. Single move control signal generator

CT and ST selector Generator

The addresses are sent to decoder blocks, the decoded addresses are used for the creation of the two signals. The generation of the $CTsel$ vector does not require the I_0 vector.

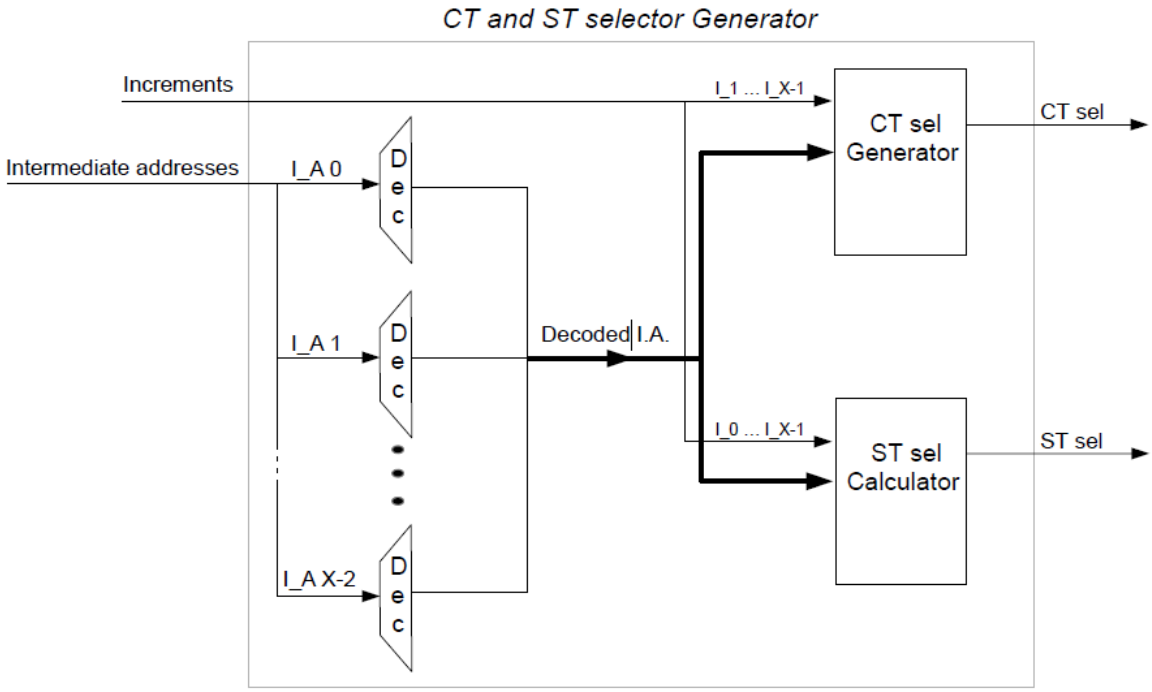


Figure 5.9. CT and ST selector Generator

CT sel Generator

Its block diagram is shown in figure 5.10. Each "CT sel Calculator" block has the task of generating the signals that must then be filtered through an AND block, in the latter the decoded addresses are used as a mask. All "CT sel Calculator" blocks must have output equal to 0 if the input is equal to 0, this is necessary because among the last intermediate addresses there may already be the final one, in these cases the address must be ignored. The final vector is obtained via an OR gate.

ST sel Generator

Like before, each "CT sel Calculator" block has the task of generating the signals that must then be filtered through an AND block, in the latter the decoded addresses are used as a mask. All "ST sel Calculator" receiving two values (I_i and $I_{(I-1)}$) as input, I_i provides the path to be taken while $I_{(I-1)}$ the one from which the information arrives. Like before, blocks must have output equal to 0 if the input I_i is equal to 0. The final vector is obtained via an or gate.

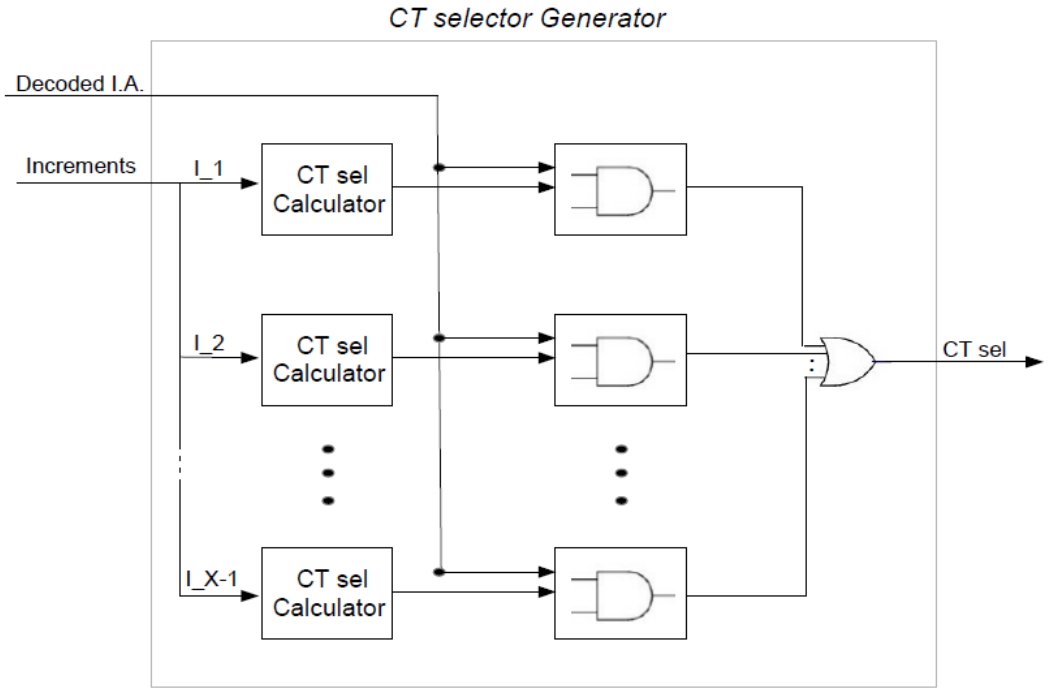


Figure 5.10. CT sel Generator

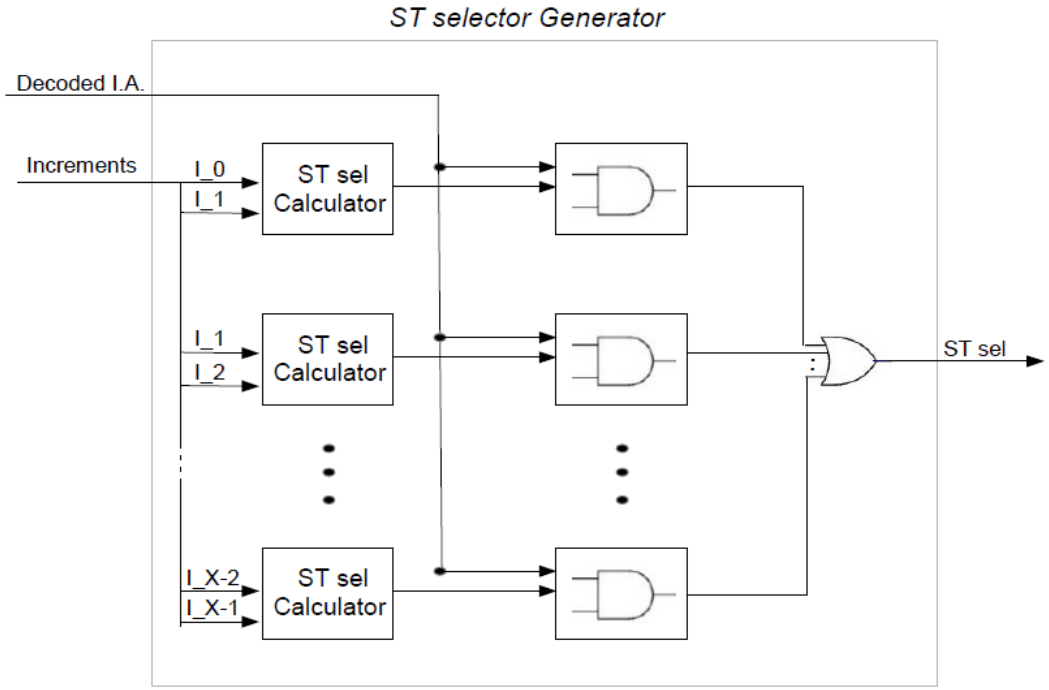


Figure 5.11. ST sel Generator

5.5 Pipelining

The signal generation block of the analyzed processor can be pipilined as desired. The critical path, and therefore the operating frequency, depend on the crossing time of the multiplexers inside the memory.

Chapter 6

VHDL processor implementation

In the previous chapter the main blocks to be developed were briefly described. In order to verify that such a structured processor can really work, the circuit has been described in VHDL language. For simplicity, a memory has been developed that allows only scrolling from top to bottom. Control circuits can simply be replicated and used to control movements in the opposite direction. The memory cells have been implemented through flip flops, no lines have been replaced with functional units as the objective was to verify the correct functioning of the move operations on the memory. Figure 6.1 shows the circuit created. The two enable signals are used to enable or disable the WE and *mux_insel* signals produced by the control signal generator. An external write enable is added to be able to write the memory, the external data to be written is contained in the input *datain*. The content of *datain* is sent as input to the input multiplexers of the cells, this is selected when the selector is 0. The memory structure has been slightly modified by adding an output multiplexer, the latter has as inputs the value contained in the last line and the output value of the input multiplexer of the same line. This multiplexer is necessary to be able to read any memory line and is controlled by the *mux_out* signal. First, the memory was created as a generic entity. The parameters that characterize the latter are: number of lines, number of columns and number of multiplexer levels. The control signal generation block has also been created in a generic way in order to set the number of parallel moves and the value of X.

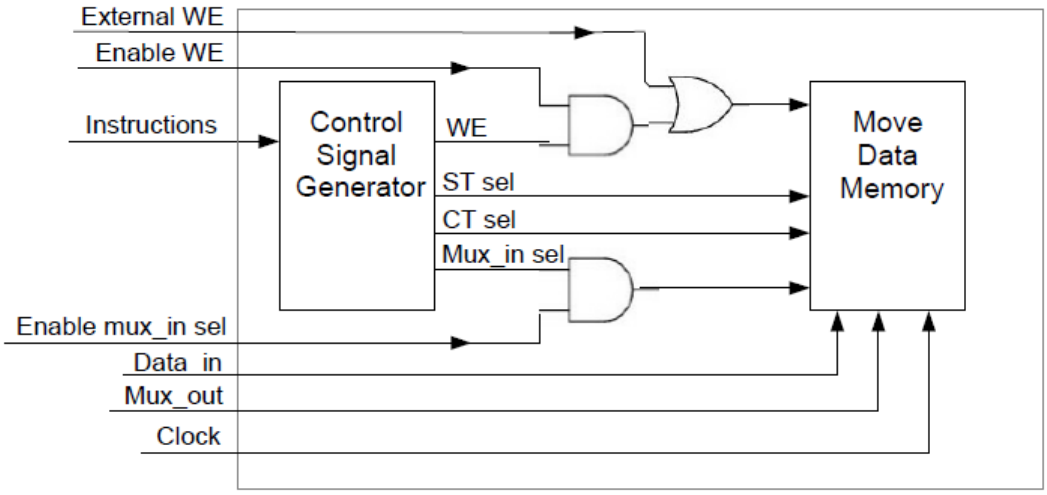


Figure 6.1. Circuit described in VHDL language

6.1 Addresses calculator

This block has the task of generating all the addresses of interest through the starting address and the increments. The starting address is formed by a number of bits equal to l, the number of memory lines is equal to $2^l + 1$. In theory, it would take $l + 1$ bit to address it, however, since line 0 cannot be a destination in any way, it is possible to maintain a parallelism of l bit. To do this it is necessary to decrease the addresses by one, in this way, the address 1 will be associated with the value 0 and so on.

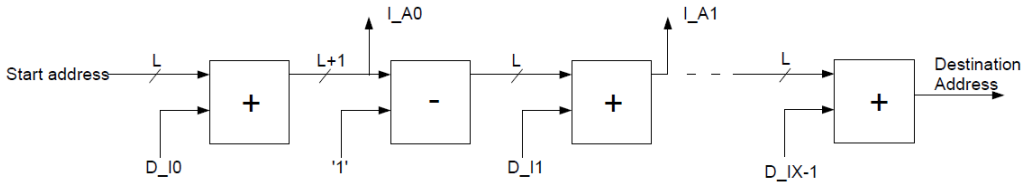


Figure 6.2. Addresses calculator block

Figure 6.2 shows the realization of this block, the number of output bits of the first block must be $l + 1$ since at this point it is possible to already have an address pointing to the last line.

6.2 ModelSim Simulation

The circuit in the figure was simulated through ModelSim. In the first simulation, the following parameters were set:

- **number of lines = 9**
- **number of columns = 8**
- **multiplexer levels = 3**
- **parallel moves = 1**
- **number of increment = 4**

The simulation result is shown in figure 6.3. When the simulation starts the memory contains no data, this is visible in the memory lines array. By setting the External WE input with all bits to one, the contents of the *Data_in* input are copied into memory. In this first phase it is necessary to set the two enable signals to 0 as everything is being controlled from the outside. The signal generation block is then enabled by setting the enable signals to $'1'$, in this phase the external WE signal must be reset to all zeros. Once the generation block is enabled, the move operation is carried out by copying line 0 to line 4 (0+2+1+1+0), followed by three other move operations. The last one is instead a memory reading operation, in order to read the memory it is necessary to set the inputs as if to move the line to be read on the last line, however it is necessary to deactivate the $enable\_WE$ signal and set $mux\_out\_sel =' 1'$. The same circuit has been simulated (figure 6.4) by setting a number of parallel moves equal to 3, in this case the reading operation involves only the line with destination 8.

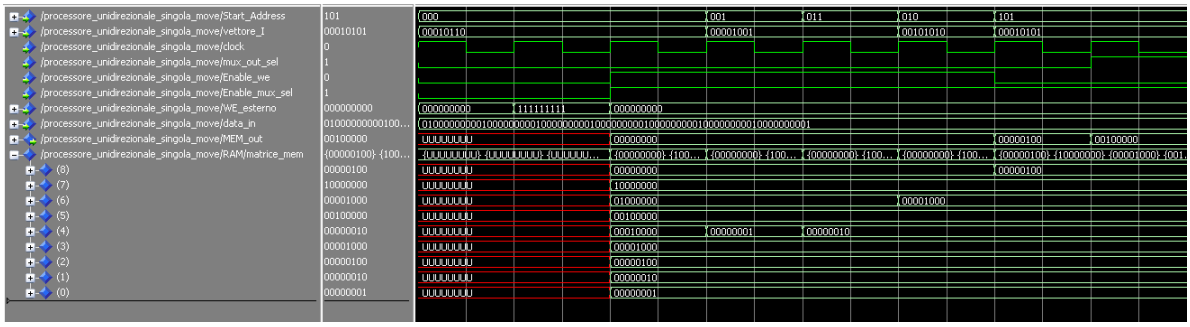


Figure 6.3. Single move processor simulation

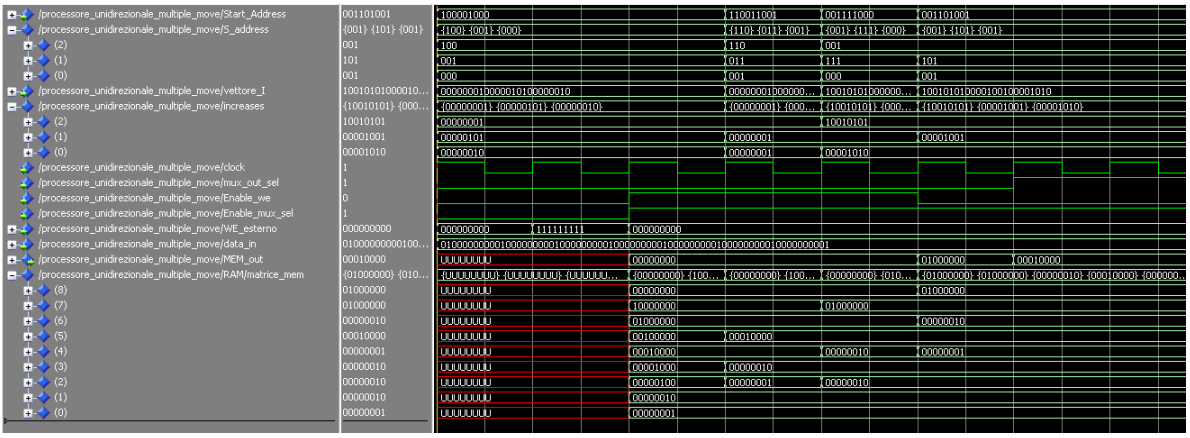


Figure 6.4. Multiple move processor simulation

6.2.1 Pipelined Architecture

The circuit under examination has been modified by introducing three levels of pipelining, the circuit obtained is visible in figure 6.5. In order to verify the correct functioning of the latter circuit too, a new simulation was made, the simulation results are shown in figure 6.6.

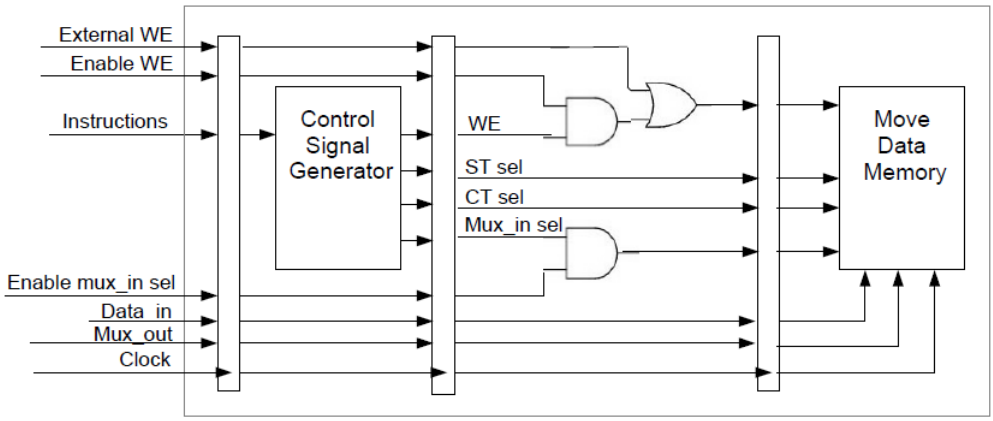


Figure 6.5. Pipelined circuit

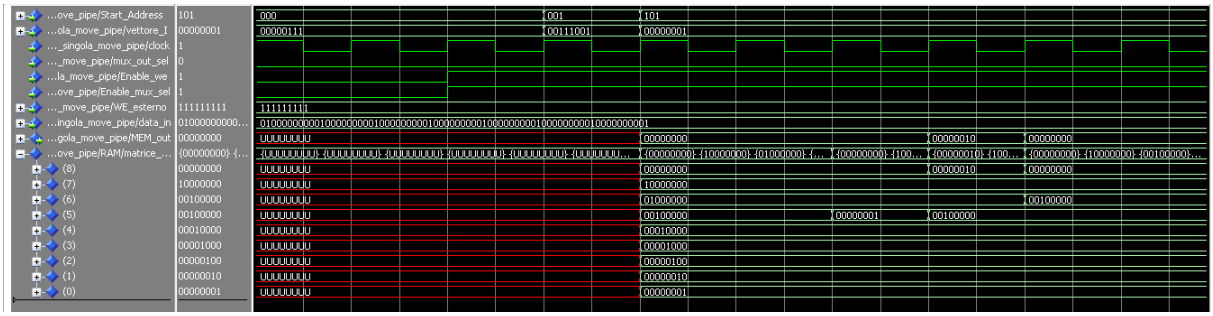


Figure 6.6. Pipelined single move processor simulation

Conclusions

The purpose of this thesis was to develop a Logic in Memory system. The proposed methodologies and architectures are a good starting point for creating a memory with good data mobility inside, the control system has proven to be easy to implement. however, it will be necessary in the future to develop new architectures and new control strategies to further improve mobility, for example by introducing mobility between columns. New strategies, such as the introduction of groups, will have to be studied in order to reduce the distance between departure and destination addresses. It will also be necessary to think of intelligent methodologies to create the functional units to put in the memory lines. In conclusion, the compiler implementation will play a decisive role.

Bibliography

[1] Giulia Santoro, Giovanna Turvani and Mariagrazia Graziano. *"New Logic-In-Memory Paradigms: An Architectural and Technological Perspective"*., Micromachines 10.6 (May 2019), p. 368. issn: 2072-666X. doi: 10.3390/mi10060368.

[2] Giulia Santoro. *"Exploring New Computing Paradigms for Data-Intensive Applications"*, PhD thesis, Politecnico di Torino, June 14, 2019.

[3] Umberto Casale. *"Programmable LiM: a modular and reconfigurable approach to the Logic in Memory"*, Master's thesis, university of Illinois at Chicago, 2020.

[4] E. Deng, Y. Zhang, J. Klein, D. Ravelsona, C. Chappert and W. Zhao. *"Low Power Magnetic Full-Adder Based on Spin Transfer Torque MRAM"*, in IEEE Transactions on Magnetics, vol. 49, no. 9, pp. 4982-4987, Sept. 2013, doi: 10.1109/TMAG.2013.2245911.

[5] W. Zhao, C. Chappert, V. Javerliac and J. Noziere. *"High Speed, High Stability and Low Power Sensing Amplifier for MTJ/CMOS Hybrid Logic Circuits"*, in IEEE Transactions on Magnetics, vol. 45, no. 10, pp. 3784-3787, Oct. 2009, doi: 10.1109/TMAG.2009.2024325.

[6] H. Cai, Y. Wang, L. A. B. Naviner, Zhaohao Wang and W. Zhao. *"Approximate computing in MOS/spintronic non-volatile full-adder"*, 2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH), Beijing, 2016, pp. 203-208, doi: 10.1145/2950067.2950101.

[7] You Wang, Hao Cai, Lirida Alves de Barros Naviner, Yue Zhang, Xiaoxuan Zhao, Erya Deng, Jacques-Olivier Klein and Weisheng Zhao. *"Compact Model of Dielectric Breakdown in Spin-Transfer Torque Magnetic Tunnel Junction"*, in IEEE Transactions on Electron Devices, vol. 63, no. 4, pp. 1762-1767, April 2016, doi: 10.1109/TED.2016.2533438.

[8] E. Garzón, B. Zambrano, T. Moposita, R. Taco, L. Prócel and L. Trojman. *"Reconfigurable CMOS/STT-MTJ Non-Volatile Circuit for Logic-in-Memory Applications"*, 2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS), San Jose, Costa Rica, 2020, pp. 1-4, doi: 10.1109/LASCAS45839.2020.9069027.

[9] J. W. Kwak, A. Marshall and H. Stiegler. *"28nm STT-MRAM Array and Sense Amplifier"*, 2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST), Thessaloniki, Greece, 2019, pp. 1-4, doi: 10.1109/MOCAST.2019.8741642.

[10] G. Turvani, F. Riente, E. Plozner, M. Vacca, M. Graziano and S. B. Gamm. *"A pNML Compact Model Enabling the Exploration of Three-Dimensional Architectures"*, in IEEE Transactions on Nanotechnology, vol. 16, no. 3, pp. 431-438, May 2017, doi: 10.1109/TNANO.2017.2657822.

[11] Stephan Breitkreutz, Josef Kiermaier, Irina Eichwald, Christian Hildbrand, Gyorgy Csaba, Doris Schmitt-Landsiedel and Markus Becherer. *"Experimental Demonstration of a 1-Bit Full Adder in Perpendicular Nanomagnetic Logic"*, in IEEE Transactions on Magnetics, vol. 49, no. 7, pp. 4464-4467, July 2013, doi: 10.1109/TMAG.2013.2243704.

[12] Stephan Werner Georg Breitkreutz-von Gamm *"Perpendicular Nanomagnetic Logic: Digital Logic Circuits from Field-coupled Magnets"*, Doktor-Ingenieurs,Technische Universitat Munchen, 2015.

[13] Fabrizio Riente, Grazvydas Ziemys, Giovanna Turvani, Doris Schmitt-Landsiedel, Stephan Breitkreutz-v. Gamm and Mariagrazia Graziano *"Towards Logic-In-Memory circuits using 3D-integrated Nanomagnetic logic"*, 2016 IEEE International Conference on Rebooting Computing (ICRC), San Diego, CA, 2016, pp. 1-8, doi: 10.1109/ICRC.2016.7738700.

[14] S. B. Gamm et al. *"Towards nanomagnetic logic systems: A programmable arithmetic logic unit for systolic array-based computing (Invited)"*, 2015 IEEE Nanotechnology Materials and Devices Conference (NMDC), Anchorage, AK, 2015, pp. 1-2, doi: 10.1109/NMDC.2015.7439269.

[15] A. Ferrara, U. Garlando, L. Gnoli, G. Santoro and M. Zamboni. *"3D design of a pNML random access memory"*, 2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME), Giardini Naxos, 2017, pp. 5-8, doi: 10.1109/PRIME.2017.7974093.

[16] L. Chang, Z. Wang, Y. Zhang and W. Zhao. *"Multi-Port 1R1W Transpose Magnetic Random Access Memory by Hierarchical Bit-Line*

Switching", in IEEE Access, vol. 7, pp. 110463-110471, 2019, doi: 10.1109/ACCESS.2019.2933902.

[17] W. Guo, J. Wei, Y. Yao, Z. Shi and S. Wang. *"Design of a Configurable and Extensible Tcore Processor Based on Transport Triggered Architecture"*, 2009 WRI World Congress on Computer Science and Information Engineering, Los Angeles, CA, 2009, pp. 536-540, doi: 10.1109/CSIE.2009.233.