Czech technical university in Prague

Electrotechnical faculty

Department of microelectronics

Master's thesis

ROM generator design

Author: Martin Švarc

Thesis advisor: prof. Ing. Jiří Jakovenko, Ph.D.

Study programme: EK

Prague 2024

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Švarc Martin** Personal ID number: **491838**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Microelectronics**

Study program: **Electronics and Communications**

Specialisation: **Electronics**

II. Master's thesis details

Master's thesis title in English:

Complete ROM Generator Design

Master's thesis title in Czech:

Návrh kompletního generátoru ROM

Guidelines:

1) Familiarize yourself with the scripting language SKILL and learn how to use it. Study the process of designing a ROM in detail.
2) Design a complete ROM generator that is able to create the memory circuitry and layout. The properties of the generated memory will incude: rows 8-256, number of words in a row 16, word length up to 64 bits, supply voltage 1.8 V, and operating frequency 50MHz. Base the generator on an already functional ROM design.
3) Include a simulation of access time and a decoding simulation into the generator. The memory access time simulation will be derived from its layout.
4) Build a user interface in the design program Cadence for using the generator and passing helpful information about the generation process to the user. 5) Optional: Also generate a digital simulation memory model.

Bibliography / sources:

Robust SRAM Designs and Analysis, Jawar Singh, Saraju P. Mohanty, Dhiraj K. Pradhan, Springer Science+Business Media New York 2013
CMOS SRAM Circuit Design & Parametric Test, Andrei Pavlov, Manoj Sachdev, Springer Science + Business Media B.V. 2010

Name and workplace of master's thesis supervisor:

prof. Ing. Jiří Jakovenko, Ph.D. Department of Microelectronics FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **02.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

prof. Ing. Jiří Jakovenko, Ph.D. Supervisor's signature	prof. Ing. Pavel Hazdra, CSc. Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt Student's signature

Affidavit

I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where stated otherwise by reference or acknowledgment, the work presented is entirely my own.

In Prague ....
Author's signature

Acknowledgement

First and foremost, I would like to express my gratitude to my mentor at Allegro Microsystems Richard Starý for his invaluable guidance, knowledge, and feedback throughout this whole project. In a similar light, this project would not be possible without the opportunity and tools provided by Allegro Microsystems. I also wholeheartedly thank my thesis advisor Jiří Jakovenko for setting up this opportunity and for his help in finalizing this project. Last but not least, a thank you to my family and friends whose company, support and good times in between the long hours spent on this project have kept me focused and motivated.

Abstract

Time to market is a crucial factor that can determine the costs of integrated circuit design development. Tools automating parts of the designing process can save on development time by essentially skipping them. In this project, a read-only memory generator, based on an existing memory design, was developed for this purpose with use of the Cadence SKILL language. This design is a 1.8 V asynchronous memory with a 12-bit input address. The number of bit lines directly corresponds to the width of the output data bus. The generator capabilities are memory schematic and layout generation, memory reprogramming, automatic decoding and post-layout access time simulations, and generation of a functional model for decoding simulations in Verilog. These functions can be run separately by using a graphical user interface integrated directly into Cadence Virtuoso menus. Created memories range from 128 B to 65.536 kB with access times from 4.2 ns to 6.9 ns under nominal conditions. Corner runs show an increase of, at most, 78% from the original value. Additionally, the generated memory layout area spans from 21397 μm^2 to 829776 μm^2. The maximum memory generation time was 1 hour and 31 minutes.

Keywords: ROM, memory, generator, compiler, SKILL

Abstrakt

Čas, než se produkt dostane na trh, je jedním z rozhodujících faktrorů, které ovlivňují vývojové ceny integrovaných obvodů. Nástroje, které automatizují části průběhu vývoje, zkracují tento čas jejich přeskočením. ROM generator, založen na existujícím spolehlivém návrhu paměti, byl vyvinut v tomto projektu k dosažení tohoto cíle s použitím jazyka SKILL. Zmíněný návrh je asynchroní paměť s 1.8 V napájecím napětím a 12-bitovou vstupní adresou. Počet sloupců těchto pamětí přímo určuje šířku sběrnice výstupních dat. Schopnosti generátoru jsou generace schemat i layout paměti, přeprogramování paměti, automatické dekódovací simulace, automatické simulace přístupového času vycházející z layout generované paměti a vytvoření funkčního modelu pro digitální simulace ve Verilogu. Zmíněné funkce mohou být spuštěny odděleně pomocí grafického uživatelského rozhraní, které bylo integrováno přímo do Cadence Virutoso. Velikosti generovaných pamětí se pohybují od 128 B do 65.536 kB s hodnotami přístupových časů od 4.2 ns do 6.9 ns za normálních podmínek. Simulace spuštěné přes rohy vycházejí s maximálně 78% nárůstem přístupového času od původní hodnoty. Zároveň se layout plocha generovaných pamětí pohybuje od 21397 μm^2 do 829776 μm^2. Maximální generační čas paměti byl 1 hodina a 31 minut.

Klíčová slova: ROM, paměť, generator, kompilátor, SKILL,

List of Abbreviations

ADE	Analog Design Environment
AFS	Analog fastSPICE
ATD	Adress Transition Detector
BL	Bit Line
CE/CS	Chip enable/Chip Select
CIW	Command Interpreter Window
CLK	Clock
CMOS	Complementary Metal-oxide Semiconductor
COLE	Column enable
CRC	Cyclic Redundancy Check
DPC	Dummy Pre-charge
DRC	Design Rule Check
HDL	Hardware Description Language
IC	Integrated Circuit
LATE	Latch enable
SAOD	Sense amplifier output dummy
LVS	Layout vs Schematic
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
OCEAN	Open Command Environment for Analysis
PC	Pre-charge
RAM	Random Access Memory
ROM	Read-only Memory
UNIX	Uniplexed Information Computing System
VNEG	Voltage negative
VPOS	Voltage positive (Supply voltage)
VSUB	Voltage substrate
WL	Word Line

List of Figures

List of Tables

Table of Contents

Chapter 1: Introduction

Although research today focuses mainly on RAMs (random access memories), ROMs (read-only memories) remain a critical part of microcontrollers. Data stored in a ROM is programmed in the manufacturing process and remains stored even on power loss. It usually contains instructions for device start up, firmware or other permanent data that does not require fast access. Technological improvements in IC (integrated circuit) design ensure that new and improved electronic memories are being developed constantly.

Cost effectiveness is an important factor of every IC design. While a ROM already has an advantage over RAM in this respect, further reductions in costs can be made. Time to market significantly affects the cost of a design and reducing it is an everyday struggle of companies. Though an electronic memory is only a small part of an IC design, the time to develop it can become considerable, especially concerning the memory layout. Efforts are being made to automate the process of memory design as much as possible, reducing the time to market. These range from simple scripts that generate or program memory cells to fully fledged memory compilers or generators. They can not only generate but also have additional features to help with the design and verification process such as automatic testing and reprogramming of the generated memory.

With the use of a memory generator, designers can automatically generate electronic memories by specifying the size needed for their application. The generator can differ in types of memory and in the way it is optimized, e.g. speed, reliability, efficiency, or area consumption. In the case of ROMs, the generator must also program the generated memory with data. However, memory generators are not freely available. Additionally, the generator must have its parts separately developed for each new memory design to achieve high-quality results. Consequently, it is common practice that companies develop and use in-house memory generators, that are based on memory designs that were already developed, tried, and tested. The goal of this work is to create one such in-house generator for the purpose of saving time on ROM design and enabling designers to focus their efforts on other projects.

A ROM generator for 180nm CMOS (Complementary Metal-oxide Semiconductor) technology, based on an existing design and intended for this use, is to be developed, and tested with tools part of Cadence software in this project. The generator must be able to create memories with rows ranging from 8 to 256 and word length at least up to 64 bits. The number of words per row will be 16 and the memory supply voltage 1.8 V. The generated memories will be capable of working at a 50 MHz operating frequency.

Apart from the ability to generate a ROM with both schematic and layout views for different sizes specified by the user, the generator should also allow the user to quickly test and measure important parameters of the generated memory. Furthermore, the generator will be controlled by an easy-to-use user interface that passes information about ongoing memory generations to the user. Additionally, a functional digital model for digital simulations of the generated memory can be generated.

Certain ROM dimensions and sizes are more likely to be used than others. For example, the word length is always determined by a microprocessor that is intended to use the memory. The data bus width is usually divisible by 8 following a byte structure. However, when error-detecting codes are used, the width increases by a number of bits depending on the code used. A single parity bit, that detects an odd number of errors, makes the data bus width odd. Similarly, multiple bits can be claimed by more advanced checks like CRC (cyclic redundancy check). Although these are used sparingly with ROM designs, there is no reason to restrict the generator needlessly. As such, the generator will be implemented to a full range of capabilities of the existing design with word lengths including 64 bits and more.

The chapters covering the design of the memory generator are ordered as follows: Chapter 2 discusses the basics of read-only memory design and its function; Chapter 3 describes software used in development of the generator, including the SKILL language; Chapter 4 continues with a detailed overview of the ROM design chosen for the generator; Chapter 5 covers the developed ROM generator capabilities and functions; Chapter 6 analyzes the data resulting from generating and simulating memories created by the generator; Chapter 7 is a conclusion summarizing the results of this project.

Chapter 2: Read only memory overview

A ROM has several characteristic properties. Reading from the memory does not change the values stored. ROMs are also non-volatile which means that they do not lose stored information when their power supply is removed. That is why they are usually used for storing essential data, used by microprocessors, that does not need to be modified throughout their life cycle.

Although different types of field-programmable ROMs exist and are widely used, this thesis focuses on those that are programmed in their manufacturing process, also called mask ROMs. Mask ROMs are the least space-consuming and cheapest memories as they are factory programmed and their cells only use a single transistor to store a bit of data. A modern ROM deploys several different subcircuits and techniques to ensure quick and reliable operation. Figure 1 depicts a simplified block diagram that shows how different parts of the memory can relate to each other.

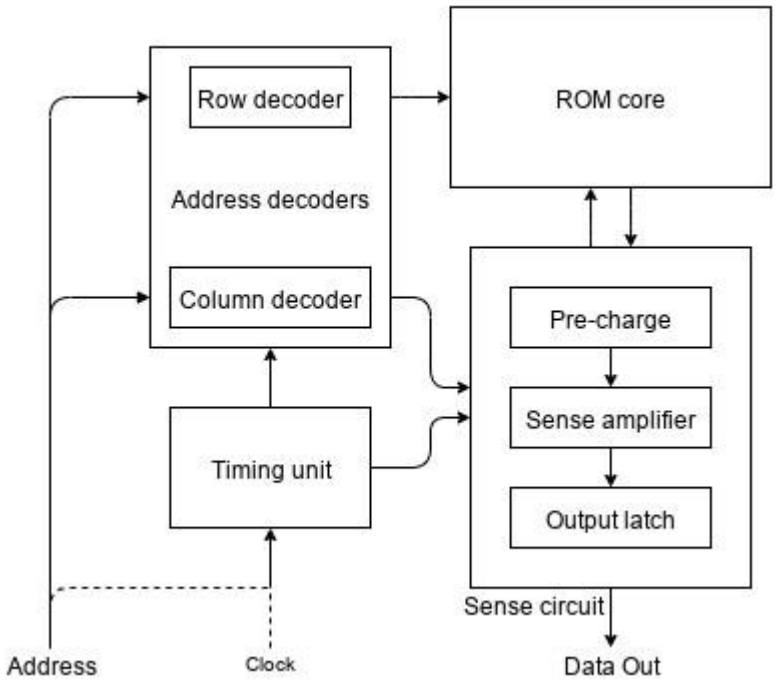

Figure 1: A ROM block diagram

Upon reading data from a ROM, the outcome of a single cell depends on the transistor threshold voltage and connection to a bit line (BL). The output can equal either "1" or "0" when the row or word line (WL) voltage is greater or equals the threshold voltage of the controlling transistor while the transistor is disconnected from the bit line. Any other state results in the opposite value. Consequently, the memory is always programmed in its manufacturing process.

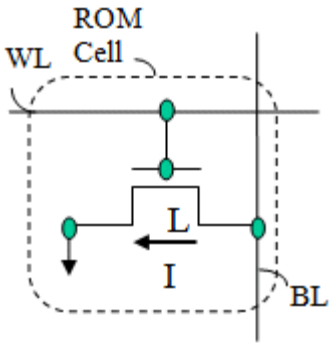

Figure 2: ROM cell topology [1]

A common and simple way to program a read-only memory is connecting or not connecting a transistor to the appropriate column or bit line with a metal contact. This method does not necessarily increase chip size as the interconnect can be placed on different layers or can take the form of a single via. Other methods include channel implantation, which creates enhancement-mode or depletion-mode transistors and changing the threshold voltage with the width of the gate oxide [2].

A single cell is multiplied vertically to match the number of rows and horizontally to match the desired word length and number of words per row of the memory, thus creating a memory array, also called a memory core. It is in this step where manufacturing costs can increase if the layout of the cell is not space efficient. Other factors are also presented with increasing size of the memory that need to be accounted for. For example, assuming the supply rails to each cell have resistance, their voltage can vary depending on the layout of the memory. The high number of transistors can also lead to considerable power dissipation due to the high capacitance of the memory word lines and bit lines. These and other unavoidable factors that can lead to wrong output readings, if not accounted for by the circuit design, will be further discussed in this chapter.

2.1. Address decoding

As the memory core grows, so does the number of word lines, bit lines, and consequently the number of address signals to drive them. The requirements of today's state-of-the-art technology are increasingly more demanding for memory capacities with the number of word lines easily reaching hundreds. If the number of address signals were to remain unmanaged, not only would it be impossible to communicate with the memory, due to the mass of signals required, but it would also lead to substantial differences in path lengths and subsequently an increase in the time the signal takes to reach the word lines and bit lines. That is why the approach of implementing a memory by arranging the words linearly, where each single word would have its own independent address signal for possible reading, leaves much to be desired.

An address decoder decreases the number of addressing signals needed to access ROM data by a factor of $\log_2 N$, where N is the number of independent address points [3]. Additionally, it ensures that only a single row is accessed in a read cycle. This is necessary for the correct function of the memory, as reading from multiple addresses simultaneously would lead to incorrect readings. The decoder has a unique output assigned to every binary combination of input address signals, as shown in Table 1.

Table 1: Logic truth table of an address decoder

Address bit 0	Address bit 1	Address bit 2	Address point
0	0	0	R0
0	0	1	R1
0	1	0	R2
0	1	1	R3
1	0	0	R4
1	0	1	R5
1	1	0	R6
1	1	1	R7

Address decoders can be split into two types. While a row decoder is responsible for picking the correct row, a column decoder is used when there is more than one word in a single row. In this way, the address can be split into two parts where several bits are assigned to the row decoder and the rest to the column decoder. For example, with a 12-bit address and 4 bits used by a column decoder, the memory will have 16 words per row, and the remaining 8 bits will make up 256 rows. Column decoders help with power consumption because they enable switching of a smaller amount of bit lines simultaneously [3].

Though this approach can be used successfully with smaller memories, the decoding process can be further divided by a pre-decoding stage. Apart from being more space-efficient, this method is also more power-efficient and faster in large memories, reducing the loading on input address buffers [4]. Generally, a pre-decoder takes a set number of address bits and partially decodes them into a larger number of additional signals. The pre-decoding stage consists of OR/NOR and AND/NAND gates that convert the address into multiple combinations of sums and products of all the address bits and their negatives. Possible implementations of decoders are depicted in Figures 3 and 4.

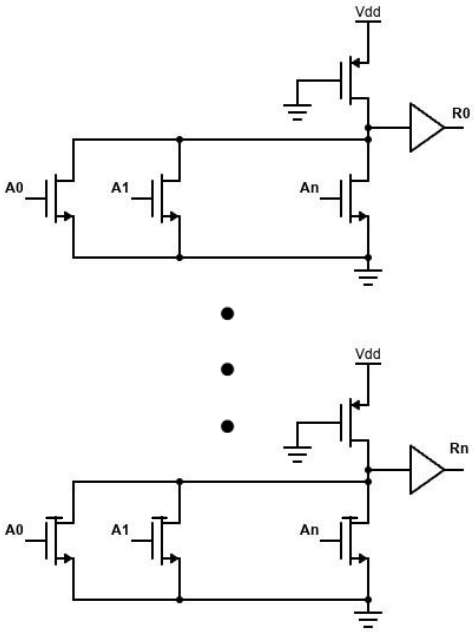

Figure 3: A static singles-stage row decoder

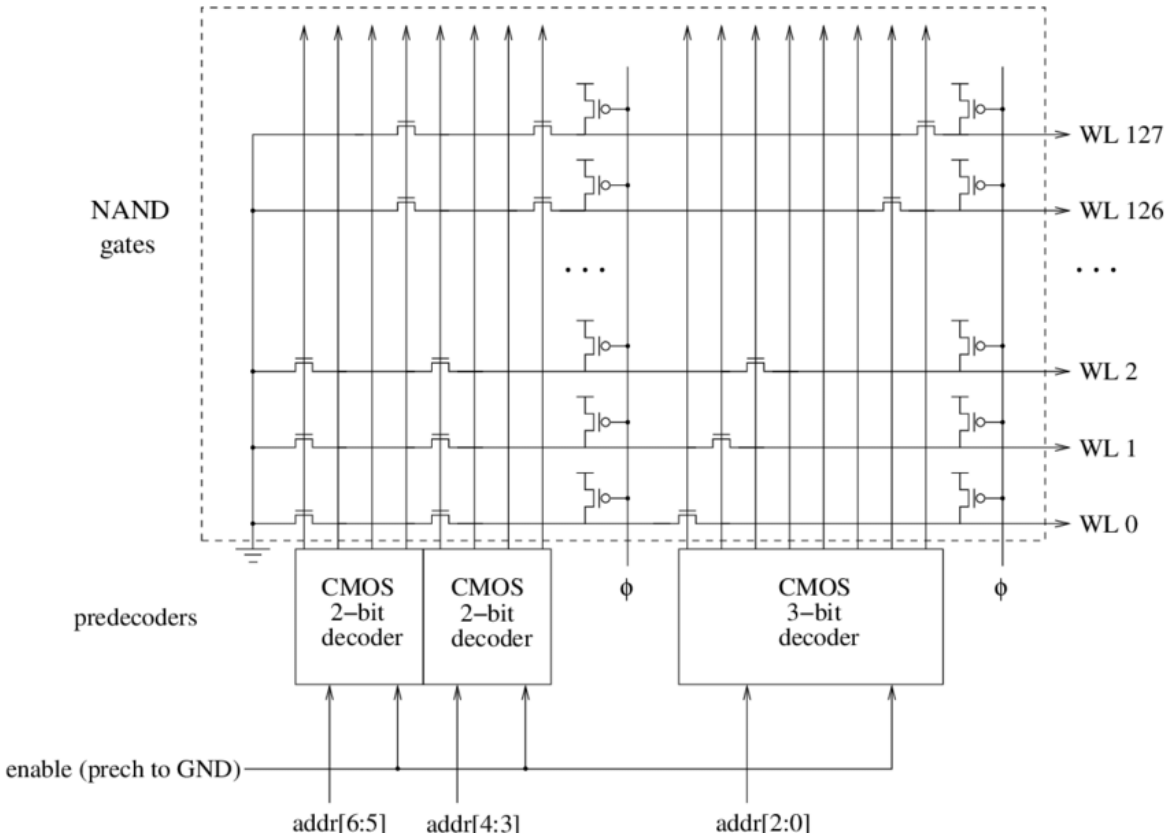

Figure 4: A 7-bit dynamic address decoder with three pre-decoders [5]

Address decoders can be static or dynamic. Static decoders are not driven by other signals and decode the input address constantly. While a static decoder is generally used because of its lower power consumption during the decoded row transitions [4] and lower area consumption, it can present a significant capacitive load to the address bus [6]. A dynamic decoder is driven by a pulse signal that dynamically enables it when the input address changes. It prioritizes speed and it can implement techniques such as self-resetting gates to reduce power consumption [6].

2.2. Read cycle operation

A read cycle is the process that a memory performs and repeats when a microprocessor requests access to the stored data. In the case of asynchronous memories, the cycle begins when the memory receives a new address and lasts until another address is received. Synchronous memories wait for a clock signal instead of reading immediately upon address change. Memory timing operation is discussed later in this chapter.

The time that it takes to change the output data from the moment a new address is received is called access time. Access time is a noteworthy metric and an essential design requirement for any electronic memory. Its complementary value, the read recovery time is the time it takes the memory to return to base output value. The read cycle will be explained further to convey how different parts of the memory are used in its duration.

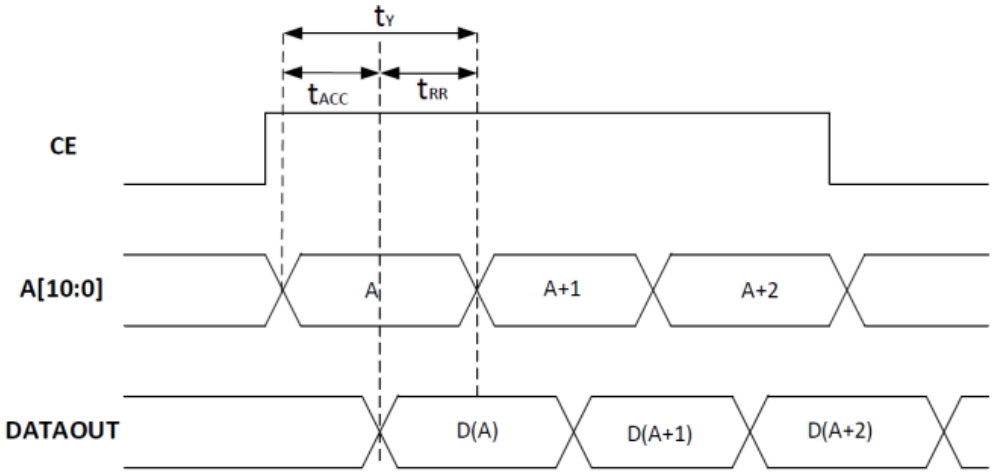

Figure 5: Access and read recovery time asynchronous diagram [7]

The cycle starts with a pre-charge process that usually takes place before receiving an address. It makes use of the fact that the extensive wiring of the bit lines behaves as multiple capacitors and holds charge for a short period of time after being charged. Therefore, all bit lines are pre-charged by the supply voltage resulting in them having the same value. This is done with a simple circuit for every bit line that consists of a single transistor, driven by a timing PC (pre-charge) enable signal, as shown in Figure 6. A PMOS transistor is used because of its good supply voltage passing capacity and consequently the ability to pass supply voltage without it dropping. [3] The dimensions of this transistor control how quickly the bit lines can be pulled up.

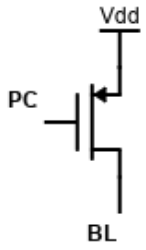

Figure 6: A pre-charge transistor

However, pre-charging can prove very power-demanding and might be unsuitable for certain low-power applications. To avoid that, pre-charging can start after receiving an address at the cost of access time and the target voltage of the bit lines can be lower than the supply voltage. Furthermore, certain selective pre-charging methods use only a part of the address to reduce the number of redundantly charged bit lines [8] in an effort to combat the power inefficiency of pre-charging. Nevertheless, this adds a certain level of complexity to the design, increasing its area, and might give way to a higher number of mismatches on the output of the ROM.

Once the address is decoded and bit lines have been charged, the relevant bit lines are disconnected from the supply voltage and connected to a sense circuit. The way a sense circuit is implemented determines word length among other things. A sense amplifier, which is commonly used to amplify and detect small signals, can be a part of said circuit. It either serves as a buffer to improve the output slew rate and set the output signal to the circuit's logic value for pre-charged bit lines or also amplifies the signal resulting from the partially charged bit line which leads to a minimized voltage span.

To achieve the minimized bit line voltage span, a current-mirror-type architecture is used with the bit lines being left unconnected in the case of logical ones on the output. In the case depicted in Figure 7, when an SAE (Sense amplifier enable) pulse closes M1, supply voltage disconnects from the gates of transistors M2 and M3. If the value on the bit line chosen by M4 is "0", M2 and M3 start conducting and the input of the output inverter starts rising together with the M2 gate voltage. After the inverter input reaches a value higher than the threshold voltage of the inverter, M2 and M3 lose conductivity and the inverter input returns to "0". This creates a short pulse of VNEG on the SENSE OUT output reflecting the value stored in the read cell. Owing to the short length of the pulse and lower voltage value on the bit lines, this sense amplifier can be highly sensitive to the presence of noise and other mismatches which can cause a wrong output signal [9].

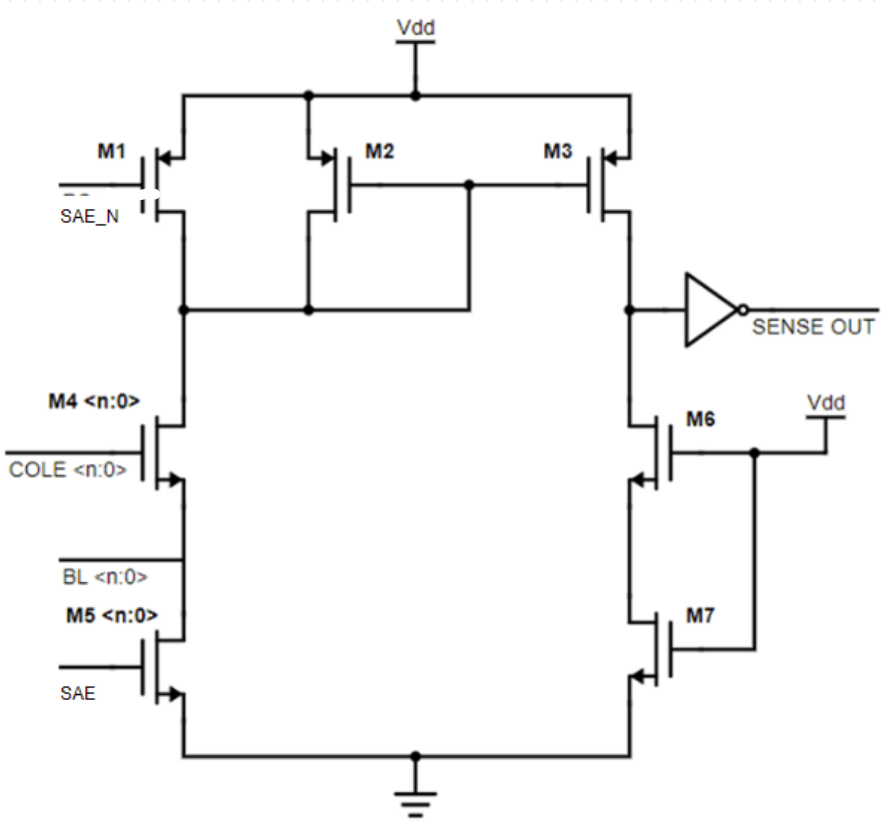

Figure 7: A Current mirror sense amplifier design with bit line select

Finally, the resulting data must remain on the output of the memory for a period of time so a microprocessor can read it. Therefore, data latches (flip-flops) are used to store the entire length of the word. The output of the latch replicates the input of the D terminal when the clock signal is high. On the falling edge of the enable signal, the latch stores the bit value of the D terminal and holds it until the rising edge starts the next cycle.

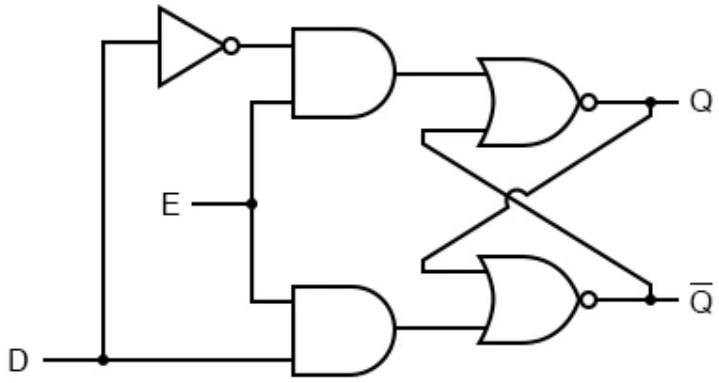

Figure 8: A typical D latch circuit [10]

2.3. Timing unit and control schemes

An electronic memory can be differentiated according to its timing operation. An asynchronous memory changes its output directly after the address input is changed. Its design eliminates the need to generate a clock signal and all the wiring associated with it. Furthermore, it helps with the time margins and clock skews over large areas in the memory designs [4]. To avoid glitches caused by its asynchronous nature, an asynchronous ROM also needs an address transition detection circuit. The address transition detector typically consists of blocks with delay elements, consisting of inverters with longer channel lengths or lower widths in series, and logic gates that detect rising or falling edges replicated for each bit of the address. Every time one or more address bits change, the address transition detector generates a pulse by pulling down the supply voltage. This pulse is further used by a timing unit. Similarly, a synchronous memory changes its output with a rising or falling edge of a clock signal which is supplied to the timing unit where it is adequately delayed for other memory blocks.

However, the choice between synchronous and asynchronous architecture has its set of criteria that depends on many design parameters of microprocessors and the communication protocols used by them. Nevertheless, the fact that asynchronous designs provide dynamically chosen frequencies leads to enhanced scalability over synchronous designs. Additionally, asynchronous designs are overall less susceptible to latency [11].

The timing unit ensures precise timing of the address decoder, pre-charge circuit, and output latch. It also deals with timing hazards, such as a change in the address input before the read cycle is completed, which can lead to two cells being read from simultaneously [4]. Although the timing unit also consists of only delay elements, logic gates, and buffers, it can vary substantially depending on the specific timing control scheme implemented.

The simplest scheme is to directly connect the timing signal to all the previously mentioned blocks, but it is clearly not optimal for larger memory dimensions because of its dependence on timing margins for reliability. The first solution is to place delay elements in series and pull the timing signals from different points of the delay chain accordingly. Thus, the delay elements are chosen to replicate the approximate delay caused by the memory size. For an even tighter approximation of the pre-charged bit line delay, which can change due to threshold and supply voltage fluctuations, a dummy column replica is widely implemented. An additional bit line is placed at the edge of the memory core with all its cells storing a value that corresponds to cell discharging, ensuring maximum delay independently on the word line chosen by the adress decoder [12]. Consequently, the bit line delay is accurately replicated with only a small amount of overhead. When a word line is activated, the dummy column begins to discharge in parallel to it and is evaluated in a sense circuit without a latch. The sense circuit must reproduce the remaining delay by using lower-width transistors and delay elements as was discussed earlier.

2.4. Simulation methods

In all integrated circuit designs, testing and verification of the design is performed. In this process, essential parameters are checked meticulously to guarantee the circuit's correct function as much as possible. In the case of a ROM, two main criteria must be verified. The memory output data must match its stored data, and the speed at which the data can be read repeatedly must meet the design specifications of the processor it will be used with.

To validate that the memory data output is correct, two tests should be carried out. Firstly, a decoding simulation that checks if only a single word line and the correct bit lines are active, for each according address, assuming the memory uses address decoders. The number of signals and addresses that need to be checked grows rapidly with the ROM size. Testing all of them manually would prove needlessly time-consuming. Secondly, the important internal timing signals, responsible for reading the output data from a latch within the right time frame, must be checked. Due to the ever-increasing speed of memories, mismatches even fractions of a ns long can prove fatal to the function of the memory.

The next parameter to be simulated in an electronic memory is access time. One of the ways to go about measuring it is by checking for a change in output data represented by a rising or falling edge and subtracting its time from the time when the input address changes. However, to check every signal of the address for the time its value changes is unnecessary and can be easily substituted for a clock signal whose rising edge indicates address change. Asynchronous memories can also be tested this way with their address input changes following a synchronous clock signal.

These parameters can be tested not only by simulating the memory schematic circuit but also by simulating the design's layout by creating a layout extraction. This process invokes a tool that creates an accurate representation of the electrical characteristics of the memory physical layout by identifying the parasitic capacitances and resistances formed by the layout features. The post-layout simulation is usually an iterative process as the layout may need to be changed if the results do not meet the design specifications [13].

However, performing a post-layout simulation for all the design parameters is unnecessary because the parasitic elements may have zero influence over them. This proves to be true for the decoding simulation, as assuming the physical layout, verified by DRC (design rules check) and LVS (layout vs schematic) checks, is correctly routed, the result of decoding will always be identical to the result from the schematic simulation.

Chapter 3: Employed software

Certain tools and software frameworks are required for the automatic generation and simulation of a ROM. Given that Cadence tools such as the schematic and layout editor or ADE (Analog Design Environment) are tightly interwoven with and run on a powerful scripting language, SKILL, it was the first choice for the development of the generator. Other tools to help with the automation of simulations were also used and are discussed further in this chapter.

3.1. SKILL

Cadence's scripting language, SKILL, is based on LISP, which is a high-level programming scripting language, but also supports a C-like syntax. SKILL is perhaps most used for quality-of-life improvements by everyday users such as defining functions, binding them to a specific key, or putting them in pulldown menus for later use. Still, there is virtually no limitation on the applications for which it can be used [14].

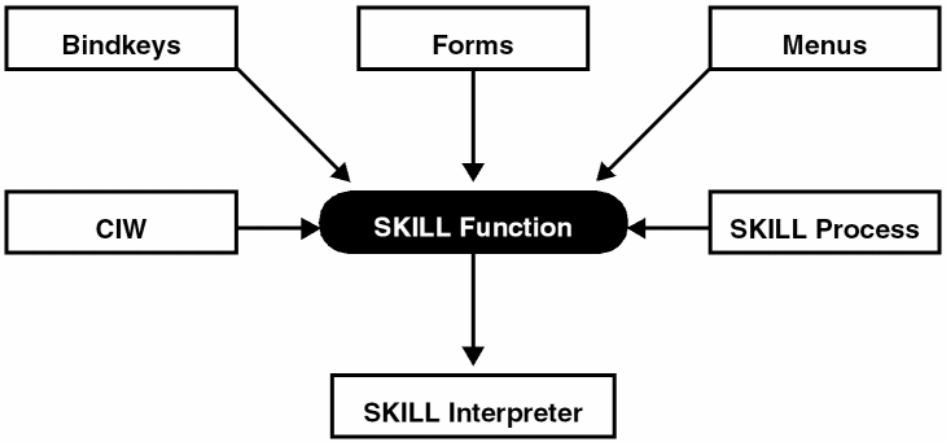

Figure 9: SKILL application diagram [14]

In Cadence, a cellview is a database object that represents a specific design or its part that can be freely opened, edited, and saved. There are multiple types of cellviews such as the typical schematic, layout, or symbol, but also configuration, among many others, that enables substitution of schematics for different types of cellviews for later simulations. An instance, also a database object, forms a part of a cellview that can represent other cellviews. Pins and terminals form the inputs and outputs of a cellview. Terminals consist of attached pins and nets while a pin represents the direction and location of the terminal. A net represents a single signal or bus of signals throughout the design.

In SKILL, programs are stored as lists. A list is an ordered collection of SKILL data objects. The data objects can have any data type, including variables and other lists. The lists can be traversed just as in LISP by accessing the contents of the address or decrement register, representing the first and the rest of the list elements. Lists are an essential part of the language and are used throughout the whole generator. [14]

For the purposes of memory generation, the memory blocks must have their schematic and layout views created. This process can be simplified and broken down into a few steps that can be followed for the whole design. Firstly, a cellview is opened or created. Secondly, instances are placed on specified coordinates and have their properties set. Next, the instances are connected by wires in a schematic or paths in a layout view where additional rectangles may be placed. Then, terminals are placed, nets created, and both are interconnected in the database. Finally, the cellview is saved and closed. Naturally, certain variants of built-in functions need to be used frequently to achieve complete ROM generation. Here is an overview of some of them.

Table 2: A list of common SKILL functions for IC design [14]

Function	Description
dbOpenCellViewByType()	Non-graphically opens a cellview or creates it if it does not exist yet.
dbCreateInstByMasterName()	Creates an instance according to the library, cellview, and name specified.
schCreateWire()	Creates a wire in a schematic cellview.
schCreateWireLabel() /dbCreateLabel()	Functions for label creation of different kinds.
schCreatePin()/dbCreatePin() /dbCreateTerm	Functions for terminal/pin creation. Both pins and terminals need to be created in a layout cellview. One terminal can have multiple pins.
dbMakeNet()	Makes or returns a net stored in the database if it already exists. Necessary for the use of other functions.
dbCreateRect()/leCreatePath()	Functions for placing various layer shapes in a layout view.
dbCreateVia() techFindViaDefByName()	These functions are used to create a via in a layout view. The via parameters are defined in a separate tech file but can be customized with input parameters. Vias are not instances and can be created only in this way.
schSchemToPinList() schPinListToSymbol()	These two functions together take a complete schematic and generate a symbol out of it. The outcome is equivalent to creating the symbol from a schematic cellview directly.
lxGenFromSource()	Automatically generates a layout cellview from a schematic cellview with terminals and instances placed. The schematic instances are placed only if they have a layout cellview themselves.

SKILL functions can have different prefixes that indicate in what environment the function can be used. The ‘db’ prefix stands for database, and functions that have it are used to access database objects through their identifiers. When a database object is loaded into virtual memory, i.e. when its cellview is open, the object and its properties can be accessed and manipulated [15]. Functions with prefixes such as ‘sch’ ‘le’ ‘lx’ and so on, manipulate opened cellviews in their respective environments and may be limited by a license for that environment.

3.2. Simulation tools

ADE can be controlled with the use of a powerful scripting tool, OCEAN (Open Command Environment for Analysis), that stands as an extension of SKILL. Among other things, OCEAN enables running simulations, without a graphical environment, repeatedly. OCEAN sessions can be run directly from the CIW (Command Interpreter Window) or from a UNIX (Uniplexed Information Computing System) shell. All OCEAN commands can be divided into three categories which are simulation set-up, run simulator, and data access [16]. OCEAN can be very convenient for users familiar with ADE because its simulation scripts can be generated directly from an ADE cellview. However, the documentation for simulators, that must be chosen and configured, is more difficult to come by. Additionally, the simulators don't have complete support for these features. So, configuring the simulator's options directly through code is not ideal as the different functions might not behave as expected.

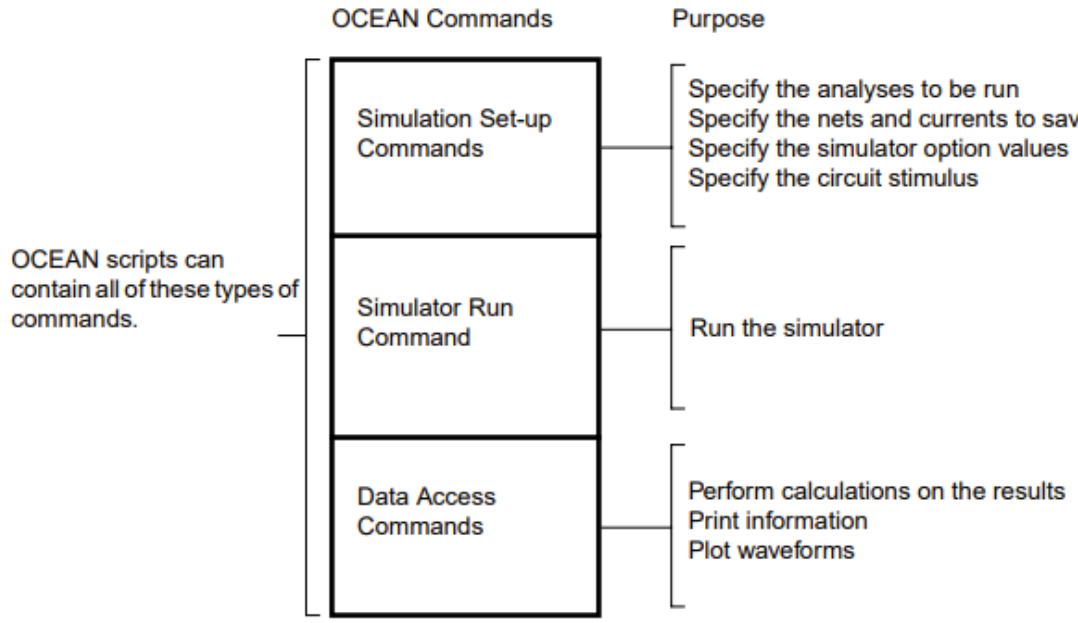

Figure 10: Types of OCEAN commands [16]

For further customization, OCEANXL, an OCEAN extension, enables the user to employ functions to directly set outputs, corners, specifications, and more. These are important to quickly determine if the simulated ROM behavior is satisfactory. An output is an expression made of functions that determine the values of different signals at a given time. The output expression results in a value that can be checked with specifications that specify the acceptable results with a maximum, minimum, or other combination of threshold values. A corner is a technique that refers to a variation of fabrication parameters to run the simulation over. They are commonly used to run the simulation over a combination of temperature and supply voltage variations which would lead to decreased performance [17].

Another tool used for the purpose of simulating is a vector generator. It takes input in the form of a text file and reliably generates signal vectors from specified parameters for commonly used simulators. These vectors can replace and act as a source in a testbench schematic when included in the simulation files. Furthermore, the vectors can also be set as expected outputs that can be checked against the results of a simulation. That is how the multitude of address inputs can be easily configured and their outputs checked with the generated vectors in the decoding simulation. An example of the generated vectors is shown in Figure 11. In this case, a change in address input is expected to bring a change in the word line signals. On the left side is the simulation time of the change in the inputs and outputs listed at the top of the figure.

```
vname    CE A<[11:0]>    DUT.RR<[31:0]>                  DUT.RL<[31:0]>
radix    1  111111111111 11111111111111111111111111111111 11111111111111111111111111111111
io       i  iiiiiiiiiiii oooooooooooooooooooooooooooooo oooooooooooooooooooooooooooooo


00000    0  000000000000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
10000    1  000000000001 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20000    1  000000000000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20004    1  000000000000 00000000000000000000000000000001 00000000000000000000000000000001
20030    1  000000010000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20034    1  000000010000 00000000000000000000000000000010 00000000000000000000000000000010
20060    1  000000100000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20064    1  000000100000 00000000000000000000000000000100 00000000000000000000000000000100
20090    1  000000110000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20094    1  000000110000 00000000000000000000000000001000 00000000000000000000000000001000
20120    1  000001000000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20124    1  000001000000 00000000000000000000000000010000 00000000000000000000000000010000
20150    1  000001010000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20154    1  000001010000 00000000000000000000000000100000 00000000000000000000000000100000
20180    1  000001100000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20184    1  000001100000 00000000000000000000000001000000 00000000000000000000000001000000
20210    1  000001110000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20214    1  000001110000 00000000000000000000000010000000 00000000000000000000000010000000
20240    1  000010000000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20244    1  000010000000 00000000000000000000000100000000 00000000000000000000000100000000
20270    1  000010010000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20274    1  000010010000 00000000000000000000001000000000 00000000000000000000001000000000
20300    1  000010100000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20304    1  000010100000 00000000000000000000010000000000 00000000000000000000010000000000
20330    1  000010110000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20334    1  000010110000 00000000000000000000100000000000 00000000000000000000100000000000
20360    1  000011000000 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
20364    1  000011000000 00000000000000000001000000000000 00000000000000000001000000000000
```

Figure 11: A vector file generated by a vector generator

This was a brief overview of the software used to develop the ROM generator. How the SKILL and OCEAN functions were used will be further discussed in Chapter 5. In the next chapter, the ROM design selected for the generator will be described.

Chapter 4: Used read-only memory design

The ROM design to be adopted by the generator is an asynchronous modular memory design. It has low power consumption at a 1.8V power supply and an operating frequency of 50MHz. A 12 ns read access time is guaranteed with a single 64-bit word output data bus and a 12-bit input address. It also has a chip enable (chip select) input which can be used to activate or deactivate the memory. Address decoding is split into two stages. The memory uses a dummy column control scheme to drive all the timing signals. The general structure of the read-only memory is depicted in Figure 12 and will be discussed in detail throughout this chapter.

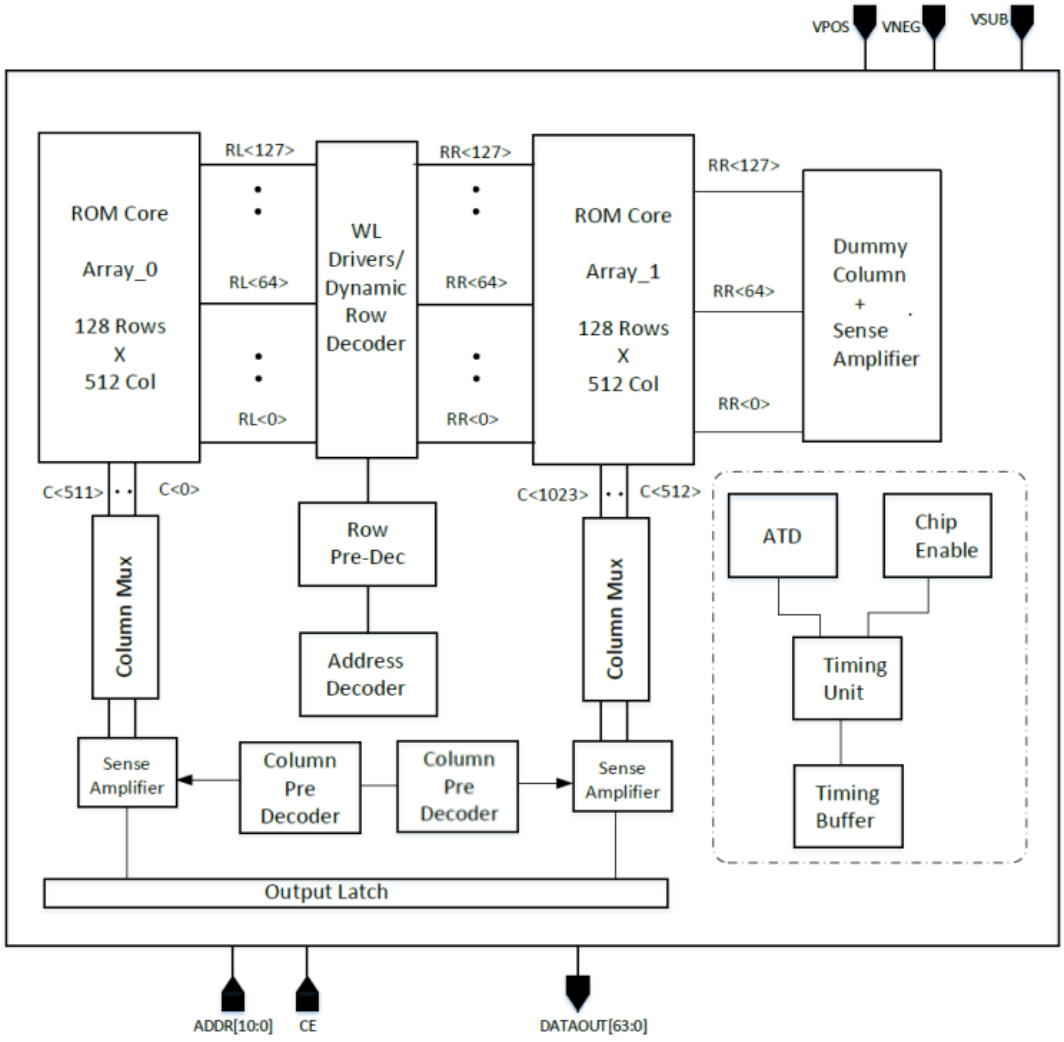

Figure 12: The generator ROM block diagram [7]

4.1. Memory core and word structure

One row of the memory core consists of 16 words and is organized into blocks of 16 cells. This pattern is repeated until the number of blocks is equal to the word length of the memory. These cells are arranged such that the block of 16 is formed by one bit from each word of the same position, and only one of them is selected at a time by the column decoder. This cell arrangement is called interleaving. The blocks are arranged in an ascending order where the first block represents the first data output bit, the second represents the second data output bit, and so on.

The data is stored in the form of metal interconnects allowing simple modification and reprogramming of an existing design. The drain of a given transistor simply needs to be connected to the according column for the value stored resulting in "0" and unconnected for it being evaluated as "1". When the drain is connected to the column and the row driving the transistor gate is active, the pre-charged column is connected to ground and the column's voltage starts discharging into it, creating a drop in value. This affects all the transistors that are connected to their column. Figure 14 depicts a row selection with a connected (M0 with C<0>) and unconnected (M1 with C<1>) transistor, also visualized in Figure 13. The transistors corresponding to the input address are later selected from the row in the sensing circuit with the use of a column multiplexer.

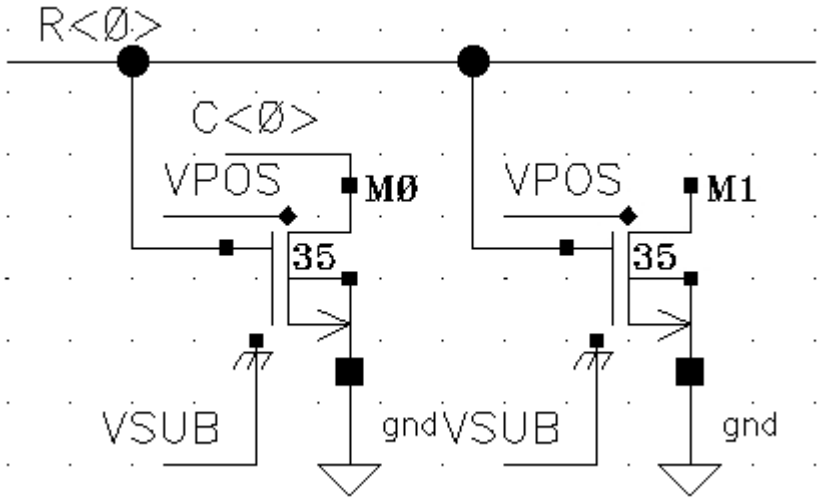

Figure 13: Two ROM core cells with different values stored

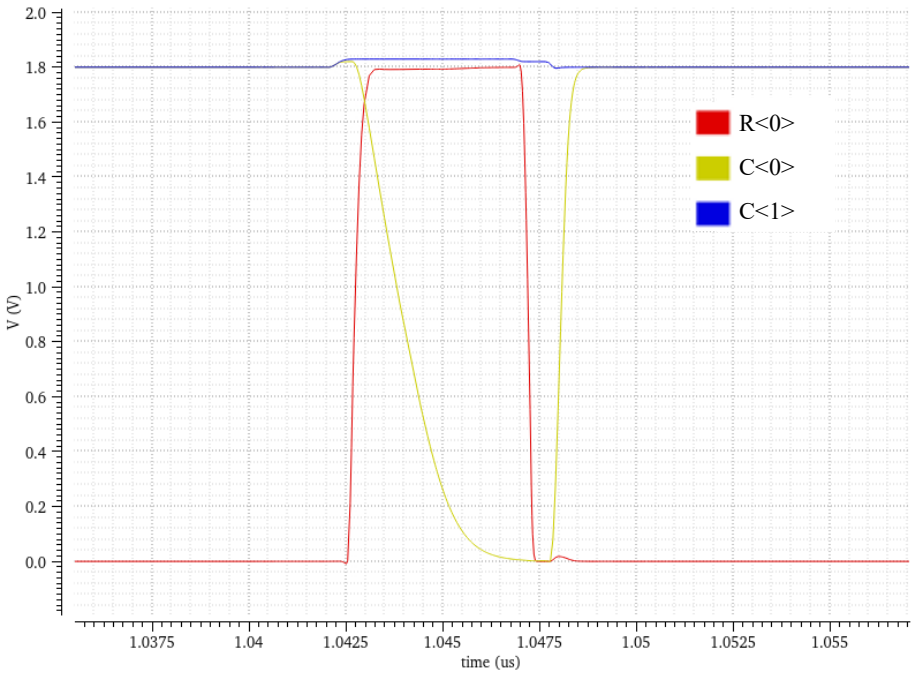

Figure 14: Memory row selection

4.2. Address decoding scheme

Given that the address bus is 12-bit wide, and each row contains 16 words, the column decoder needs N address bits to access all of them. Since $2^n = 16$, the column decoder will use 4 address bits. That leaves the last 8 bits for the row decoder with the maximum number of rows being 2^8, resulting in 256 rows. These address bits are first pre-decoded on address transition into 24 signals that lead directly into the already mentioned decoders, where they are decoded into the appropriate number of signals while ensuring only one of them is active at a time. When the address or CE signals change, they are prone to stabilize only temporarily, as shown in Figure 15, which could lead to unexpected timing signal behavior and later cause wrong output data if not accounted for.

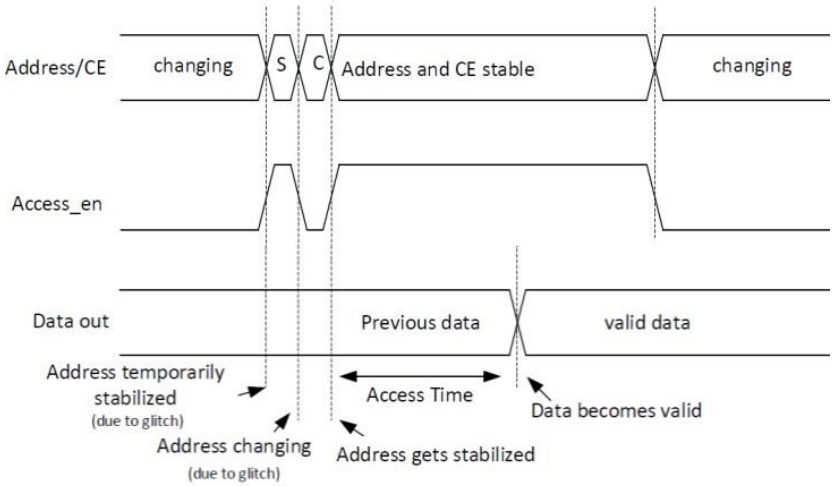

Figure 15: Address transition detector timing diagram [7]

The address transition detector signal is triggered on any transition on the address inputs through a set of XOR gates, each with a delay element on one of the inputs. The output of the XOR gates is connected to a transistor gate. Any change on the address lines causes the transistor to open, pulling the VPOS supply voltage from the address transition detector output, and generating a pulse the duration of which is determined by the delay element. The time the individual address bits change varies slightly, and so the length of the resulting pulse is extended until the address is stable. One branch of this circuit, that detects address transition on a single address bit, is depicted in Figure 16. The whole circuit has 12 branches that are all connected to the ATDRaw bus.

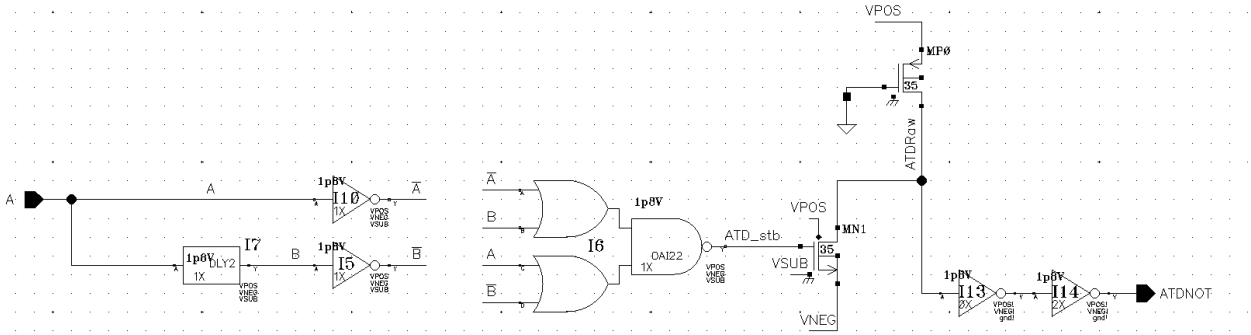

Figure 16: One branch of the address transition detector

The pre-decoders take two address inputs and decode them into four output signals, only one of which can be "0", which is later evaluated as active. When the Chip select signal (CSA) is "0" or, put differently, reading of the memory is not enabled, not one of the signals is active and no rows and columns are selected.

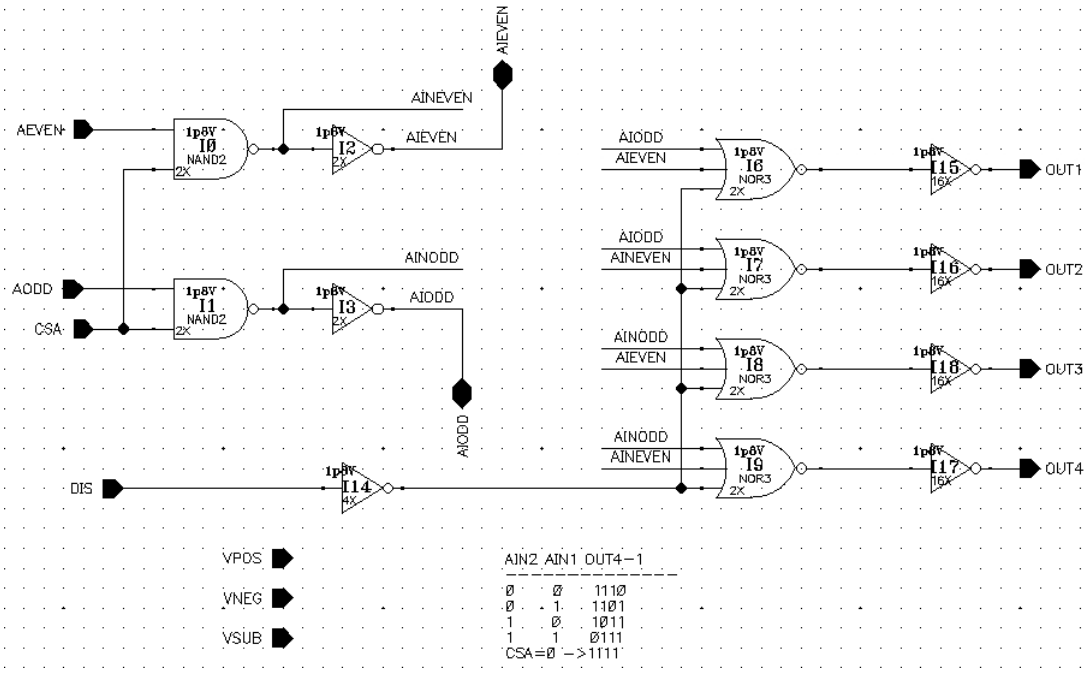

Figure 17: A single pre-decoder subcircuit with a truth table

How the pre-decoder output signals are used is best seen in the row and column decoder circuits, where each of these signals leads into multiple p-channel MOSFETs (Metal-Oxide-Semiconductor Field-Effect Transistors). For a specific word line and bit lines to be active, a given combination of the pre-decoder output signals must be filled with only "0's". If we take all the outputs of the pre-decoder and label them with letters, we get signals grouped in fours such as A<4:1>, B<4:1>, and so on. As mentioned above, rows have 8 address signals assigned and columns have 4, so that leaves A to D for the row decoder and E with F for the column decoder. If we take these signals and arrange them in a truth table fashion, such that their value corresponds to the letters, we can see which combination of "0's" activates which word line and bit lines. A portion of these truth tables is shown in Tables 3 and 4.

The row decoder schematic is shown in Figure 18. The A, B, and C signals function as described, but D also acts as an enable signal, due to it being controlled by a pulse generated by the timing unit, for up to a maximum of 64 rows simultaneously. The output is split into two signals that lead to both of the memory core halves. The MP10 transistor improves the rising edge of the output signals. The column decoder circuit is identical to the row decoder, apart from input signals belonging to the pre-decoder. D is replaced by F and A by E, while B together with C and the two left-over p-channel transistors are not present. Simultaneously it only has one COLE (Column Enable) output.

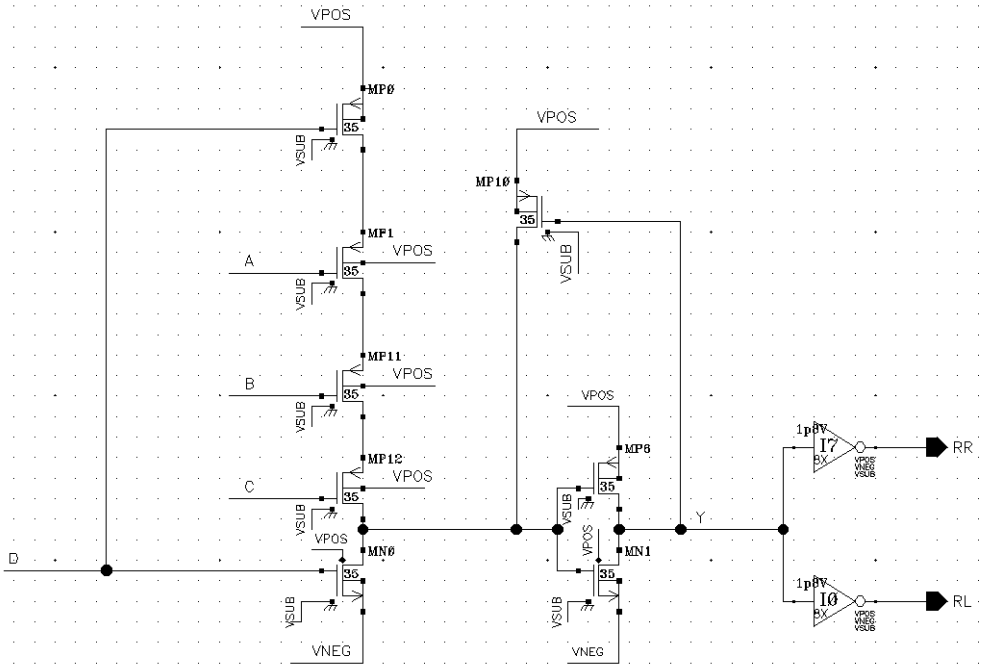

Figure 18: A single row decoder subcircuit

Table 3: Row decoder truth table

D<4:1>	C<4:1>	B<4:1>	A<4:1>	Word line (row)
D<1>	C<1>	B<1>	A<1>	0
D<1>	C<1>	B<1>	A<2>	1
D<1>	C<1>	B<1>	A<3>	2
D<1>	C<1>	B<1>	A<4>	3
D<1>	C<1>	B<2>	A<1>	4
D<1>	C<1>	B<2>	A<2>	5
: : :	: : :	: : :	: : :	: : :
D<1>	C<1>	B<4>	A<4>	15
D<1>	C<2>	B<1>	A<1>	16
D<1>	C<2>	B<1>	A<2>	17
: : :	: : :	: : :	: : :	: : :
D<1>	C<4>	B<4>	A<4>	63
D<2>	C<1>	B<1>	A<1>	64
D<2>	C<1>	B<1>	A<2>	65
: : :	: : :	: : :	: : :	: : :
D<4>	C<4>	B<4>	A<3>	254
D<4>	C<4>	B<4>	A<4>	255

Table 4: Column decoder truth table

F<4:1>	E<4:1>	Bit line (column)
F<1>	E<1>	0
F<1>	E<2>	1
F<1>	E<3>	2
F<1>	E<4>	3
F<2>	E<1>	4
F<2>	E<2>	5
: : :	: : :	: : :
F<4>	E<3>	14
F<4>	E<4>	15

4.3. Sensing circuit

COLE signals are further used in the sensing circuit where they act as the address inputs of a multiplexer. MN4 transistors act as the column multiplexer, choosing between the 16 input columns from the memory core according to the input address. All the columns are pre-charged by the MP3 transistors before data reading. When a row is selected, the cell transistors containing a "0" connect the column signals to ground and begin discharging them. The column signals chosen by the column multiplexer are then driven through a sense amplifier in the form of an inverter. With the improved slew rate, the voltage can drop well below the threshold voltage of the latch on the sense amplifier output in time with the LATE (Latch Enable) signal from the timing unit. Finally, the output value is buffered again. Given that the output is a single bit, the subcircuit shown in Figure 19 is repeated the number of times equal to the word length of the memory, forming a complete sense circuit that is controlled by two different signals from the timing unit. The SLEEP signal is unused in this configuration.

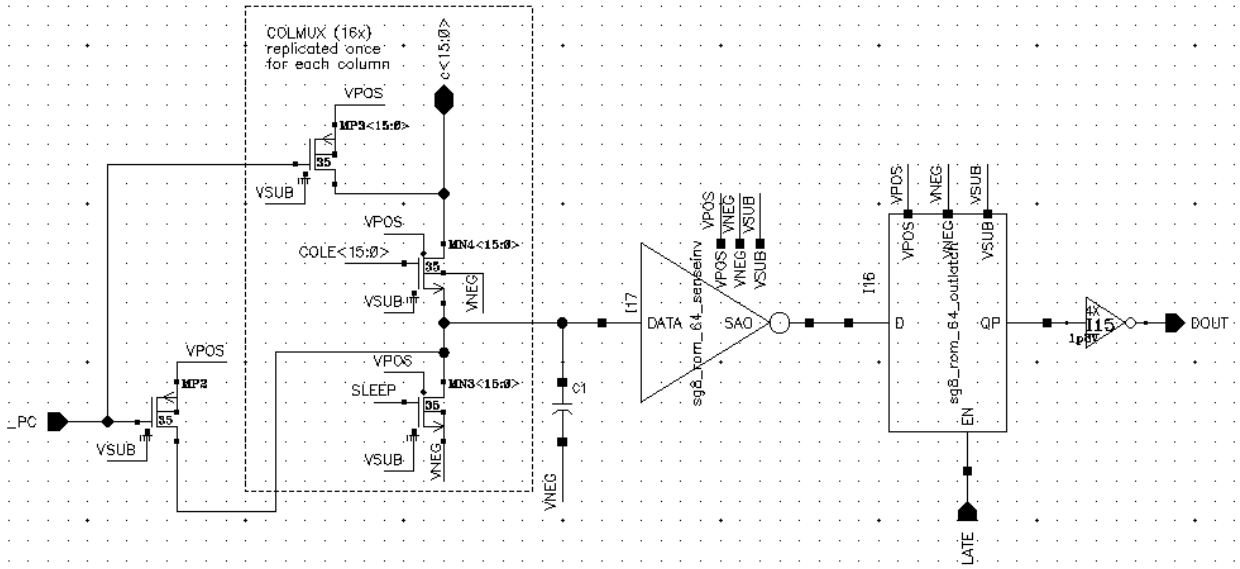

Figure 19: A Sense circuit branch

4.4. Timing unit

The timing unit uses the dummy column timing scheme that was described in Chapter 2. The output from the dummy column SAOD (Sense Amplifier Output Dummy) together with ATDNOT (Address Transition Detector Negative Output) and CS (Chip Select) signal form the inputs of an RS circuit. All of the output signals are buffered and delayed setting the right width and slew rate of their pulse.

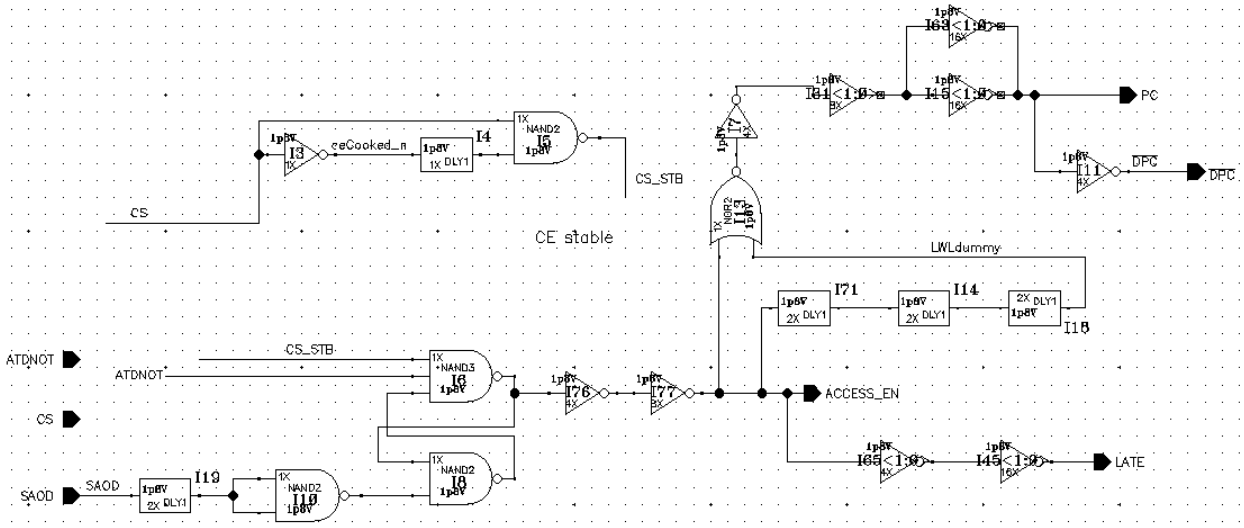

Figure 20: The timing unit circuit

When the chip select signal is active and the ADTNOT pulse reaches “0”, the ACCESS_EN (Access enable) output is set to the supply voltage until after the SAOD (Sense amplifier output dummy) input reaches the supply voltage. The ACCESS_EN pulse serves as the base signal that the other timing signals are derived from. ACCESS_EN also drives the earlier mentioned D<4:1> signals. The negative DPC (dummy pre-charge) pulse drives the dummy column, from which comes the delayed SAOD pulse. PC is later buffered and controls the actual sense circuit. LATE ensures that the moment when the latch starts holding is within the correct time frame of data being available on the output of the sense amplifier, as shown in Figure 22. Though the rising edge of the LATE signal is relatively early, no bugs will be caused by it apart from the output data always returning to “1” approximately 2 ns before the actual output value is latched. Figure 21 shows the relationship between timing signals relative to an address change that is represented by the rising edge of a CLK (Clock) signal used for simulation purposes.

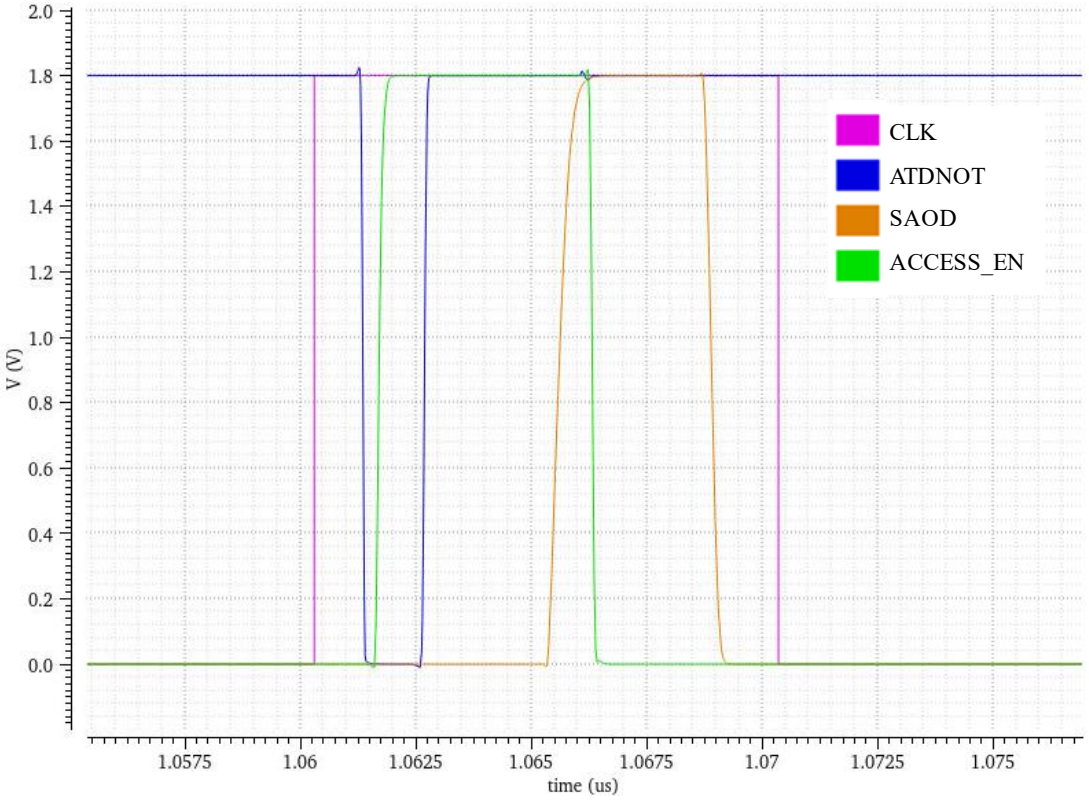

Figure 21: Timing unit internal signal operation

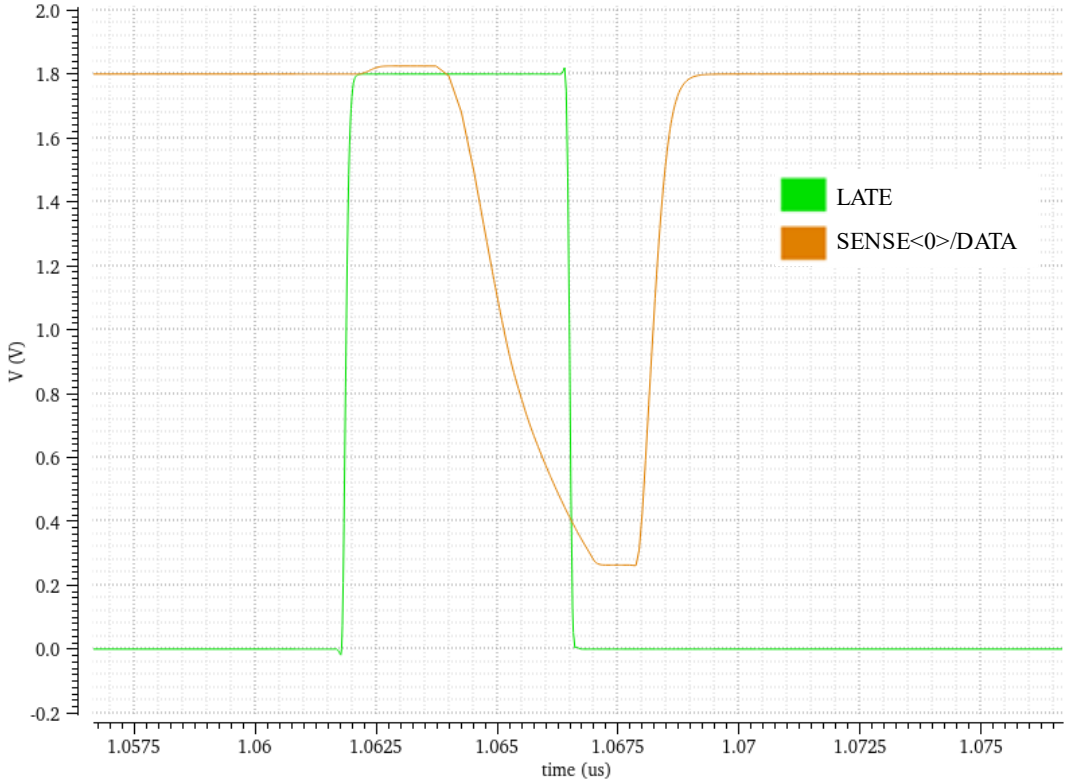

Figure 22: Latch entering hold for the output data

4.5. Layout design

This ROM layout is a 4-metal design using 180nm semiconductor process technology. It is modular in nature, as scaling the design systematically will not require the creation of new cell layouts if the certain specifications set by the nature of the read-only memory are met. These specifications are mentioned in Chapter 5 while this chapter focuses on the memory layout itself.

Matching is a very important part of every layout design for ensuring that properties of multiple transistors are almost identical. To achieve it, dummy (unconnected) cells are placed between the blocks of 16 cells. The dummy cells act as dividers in such a way that the activated cells in the middle and on the sides of the memory have similar layouts and are not influenced by their differences. Furthermore, the dummy cells make space for metal paths carrying negative voltage to the column multiplexers and sense amplifiers located below the memory core.

To shorten the path length between the row decoder and memory cells and to reduce the time delays associated with them, the memory core is divided into two halves, allowing the row decoder to be placed in the center. Below are the pre-decoders, the column decoder, the timing unit, and the address transition detector. At the sides of the address transition detector are buffers, used by the PC timing signal that is driving the sense circuit. The dummy column is located at the right edge of the memory core with its sense amplifier, using the space at the bottom right under the sense circuit. Address inputs, chip-enable, and the data outputs are all located at the bottom of the memory. The inputs are in the center and each bit of the output data is located right under its corresponding latch output.

As the memory becomes larger, it becomes more susceptible to differences in the supply voltage caused by resistance of the increasingly large supply paths. To distribute the power evenly and to reduce the resistance of the supply paths, power rings are used together with vertical stripes to carry both positive and negative voltage through the whole design. Power rings surround the whole layout with a metal layer and the stripes carry the voltage into the design itself where possible, for example through the dummy cells.

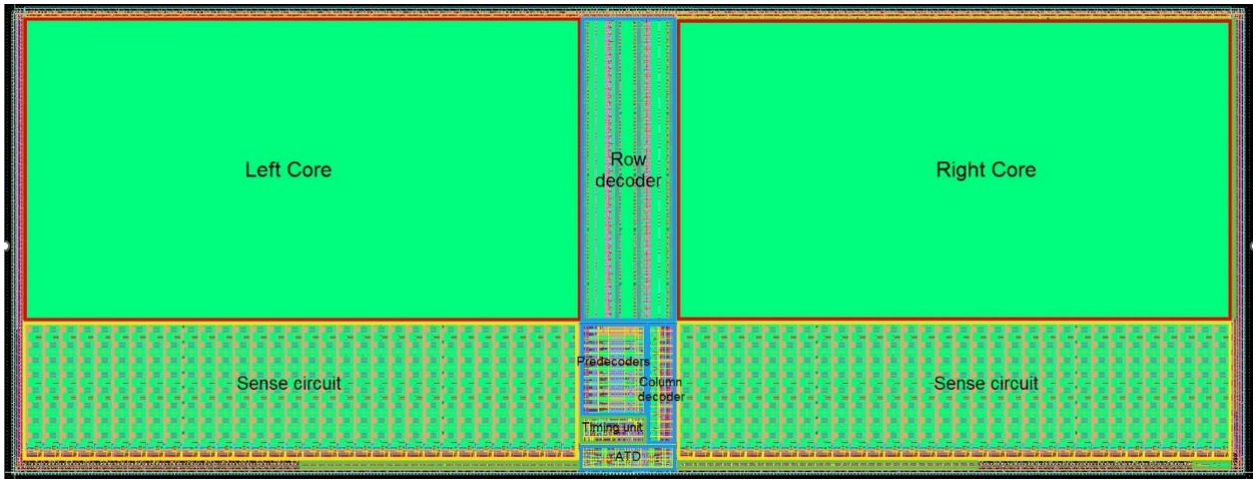

Figure 23: The original ROM layout

The dimensions of the design are dependent on the number of rows and columns. Assuming the same address and decoding structure for different sizes of the memory, the differences in the layout, caused by their changed number, are mostly limited to the row decoder, sense circuit, and memory core. Given that the different parts of the design are separated and connected to each other in a way that supports scaling, it is ideal for the extending and shrinking of the design. The automation of scaling the design is described in the next chapter together with the other functions of the ROM generator.

Chapter 5: Read-only memory generator design

The ROM generator can generate scaled schematic and layout views of a ROM for specified dimensions, simulate the memory, verify its correct function with access time and decoding simulations, and enable programming of the generated design with a text file. A functional model for use in digital simulations can also be generated. Additionally, instructions on how to use the generator can be accessed with the "Help" button. Generation and the other functions can be launched independently of each other, assuming the different conditions for each of them are met. The constraints of the dimensions of the generated ROM are determined by the design of the ROM which is described in Chapter 4. These constraints are caused both by the general design of the ROM and the way the layouts of different cells connect to each other.

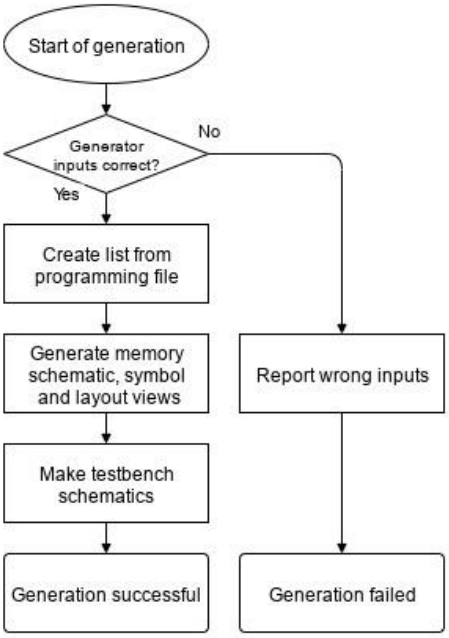

Figure 24: ROM generation diagram

The number of rows spans from as low as 8 to 256. Though it is inefficient to create a low-capacity memory design, the possibility of it is implemented. The row minimum is set because of the asymmetry of the row decoder layout whose parts cycle through a bottom middle and top branch. That is also the reason why the number of rows must be divisible by 4, for the ROM to generate successfully. The upper limit is caused by the bit count of the address as discussed earlier in Chapters 2 and 4. The lowest number of columns is also limited by the ROM layout. Without completely restructuring the bottom of the memory layout for the purpose of small dimensions, the lowest possible number of columns is 128 which results in words 8 bits wide. Besides that, the number of columns must be divisible by 16 for the memory to have the same word length on each address and function correctly. Other generator inputs include the paths to a file used to program the memory and to the ROM generator top script. Additionally, the option of specifying a suffix for key cellview names is provided.

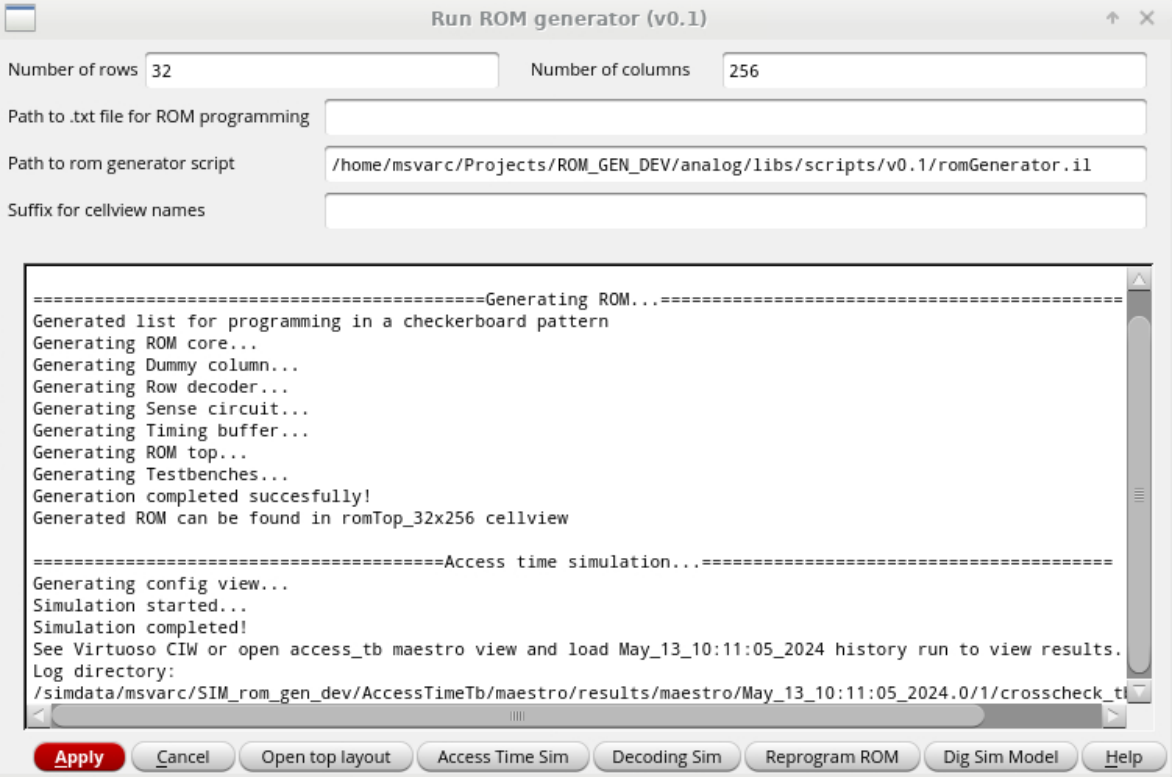

Figure 25: Generator UI

The generator also has a log that relays information about the different processes under the input string fields. Although the information is technically also located in the CIW, it is quickly lost among the other messages and warnings printed there. The log tracks the progress of the generator with a small number of commands instead. A message is printed out into the log when a new block is being generated, simulations started etc.

In this chapter, how the generator works will be thoroughly described. The chapter is divided into parts, the order of which does not necessarily follow the order of operation of the generator, but better conveys the different functions of the generator.

5.1. Read-only memory programming and reprogramming

The first function called, following the launching of the ROM generator, is the makeAddrList function that creates a list from a file that is filled with binary values, and later used to connect the drain of transistors to the bit lines. The file used for ROM programing is expected to be a text file filled with ones and zeros that signify how all the cells are to be programmed to the generator. In other words, a "1" in the text file represents a logical "1" on the output and the same applies for any "0's". The position of the characters directly corresponds to the position of the transistors in the generated memory core. Nevertheless, the dimensions of the text file don't have to be the same as the memory dimensions, and any missing characters are treated as a "0". Still, if any characters pass the dimensions specified to the generator by the user, the programming process results in an error that alerts the user rather than ignoring the extra value with a warning.

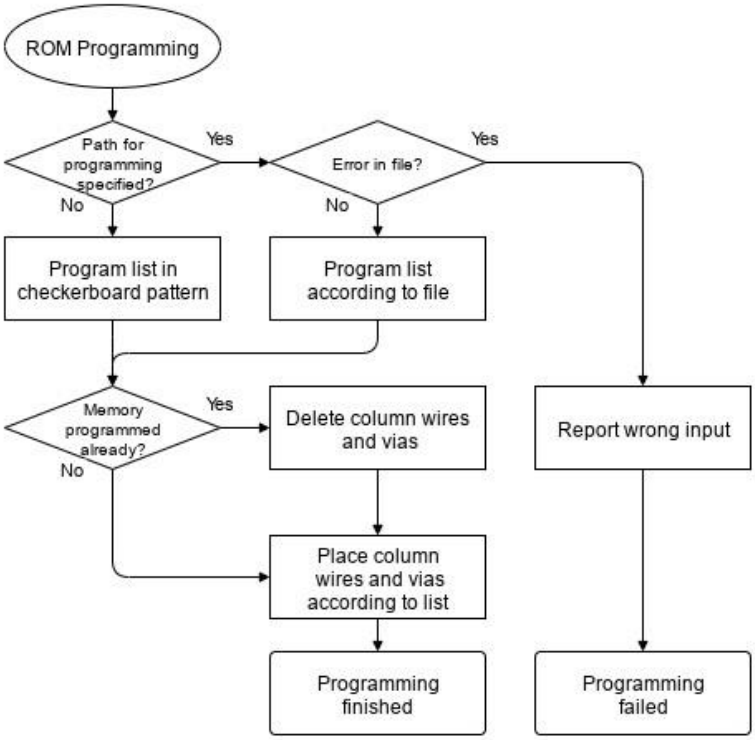

Figure 26: ROM programming diagram

The generator opens the existing file and starts creating two lists, one for each half of the memory core, that are nested in another list. The values of the lists are binary (t and nil) and correspond to the ones and zeros in the file. If no file path was specified, the lists are generated with alternating values resulting in a checkerboard-like memory core. The schematic and layout of the core are then generated according to this list, following its pattern.

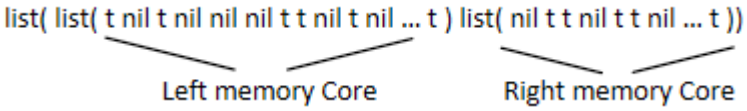

Figure 27: Programming list arrangement

The combination of these two methods allows for two different approaches to memory generation. On the one hand, it is possible to program the memory quickly alongside generating it. On the other hand, generating a memory with the default checkerboard pattern and reprogramming it later brings more accurate results from the access time simulation for reasons that will be discussed later in this chapter.

When reprogramming the memory, the generator first finds and deletes the wires connected to the drains of the memory core transistors and replaces them with a new pattern taken from the file used for programming. This is done by filtering wires in the cellview by their labels. Afterwards, the layout cellview with metal interconnects is recreated according to the pattern and the reprogramming is completed.

5.2. Schematic and layout generation

With the programming list assembled, the generator completes the task of creating the whole ROM schematic and layout design described in Chapter 4. Creating all instances used in the design would be needlessly complex while adding no benefits to the generator's function. Therefore, the generator uses a cadence library that stores predefined cells that are used in multiple parts of the design. These cells include invertors, delay elements, memory cells, cell dividers, decoders, sense subcircuits, and so on.

The functions called after makeAddrList are createRomCore, createRomDummyCol, createDyRowDecx, createSenseLayout, createTimingBuffer, createRomTestbenches, and createRomTop; each generating their respective part of the memory. However, apart from createRomTop, these functions do not cover all parts of the ROM because the other blocks do not change with the memory dimensions and thus do not need to be generated. Instead, they are placed with the createInstByMasterName function. As such, these pre-made cells must be saved in the generator library for the ROM to generate correctly. This approach helps improve the generation time while also reducing the complexity of the generator. These blocks are the pre-decoders, the column decoder, the address transition detector, the timing unit, and the sense circuit for the dummy column.

While generating a schematic with SKILL, the coordinates of instances and wires connected to them have to match exactly. Rather than checking and inputting these coordinates by hand, the generator makes heavy use of a function that calculates the exact position of the center of an instance terminal and, connects a short wire with a label to it in a direction depending on which side of the instance is the instance terminal located. The side of the instance is determined by comparing coordinates of the instance terminal bounding box and the bounding box of the instance. In this manner, every instance and terminal can be connected through wire labels only. However, this approach proves too demanding to be used for schematics with a huge number of instances and wires like the memory core. Subsequently, the createRomCore function uses a more optimized method to connect the array of transistors to word lines and bit lines by specifying the wire and instance coordinates.

In layout generation, the coordinates of paths, rectangles and vias are strictly restricted by design rules and a cost-efficient custom design. Consequently, every part of the layout is placed on specific coordinates following the pattern of the original ROM design. On top of building the layout from the ground up in the according scale, different-sized memories required small changes in the design. For example, smaller memories have a slightly larger timing buffer. The additional inverters were fitted into the lower left corner of the layout in order to further reduce the lower limit of columns that the generator can make. Furthermore, the placement of tub diodes that connect the N-well and P-well layers was changed for different dimensions of the design.

Typically, upon completing a layout design, DRC and LVS checks are run to verify the integrity of the layout. Though they are an essential part of layout design, tools, menus, and functions that are used to invoke them are often proprietary and differ from the standard Cadence toolkit, making their integration into the generator reliant on access to them. The generator, however, does not need to run these checks automatically as they were performed beforehand in its development. Still, the checks can be easily run manually, after the generation is completed, when needed. The generator supports this by having a dedicated button in the user interface that can open the top layout cellview. This button is mostly meant to be used as a convenient way to manually run LVS, with a layout extractor, needed for a post-layout simulation.

5.3. Access time simulation

With the testbenches and the layout extract ready to be used, the next step is to start testing the generated ROM. The testbench for the access time simulation makes use of a 12-bit ADC (Analog to Digital Converter) that conveniently converts an input voltage value, representing the input address in decimal form, to 12 bits on the memory input. In this way, it sets 3 input addresses, two in the middle of the memory, sharing the same row, and one on the second to last row of the memory. The addresses are set successively after a 1us startup, with the chip select signal enabled, following a 20ns clock cycle.

Due to the early timing of the output latch, a “1” appears roughly 2 ns earlier on the output than the actual output data value. This is shown in Figure 28 on two DATAOUT bit signals. One of them changes from “0” to “1” and the other stays at “0”. Consequently, the access time can be calculated correctly only from changes to “0” on the output. This way, the generator checks for falling edges for all data output bits following the 20 ns time frames. If a falling edge is detected, a time when the value of the output data signal reaches half the positive voltage is measured and saved. Then a clock signal, corresponding to the address changes, is taken from an unconnected source output, and similarly, the time of the signal rising edge is measured. Afterwards, the time of the output data falling edge is subtracted from the time of the clock signal rising edge, and the result is saved as a single access time value. After all the output data falling edges are measured, the maximum value measured becomes the memory access time.

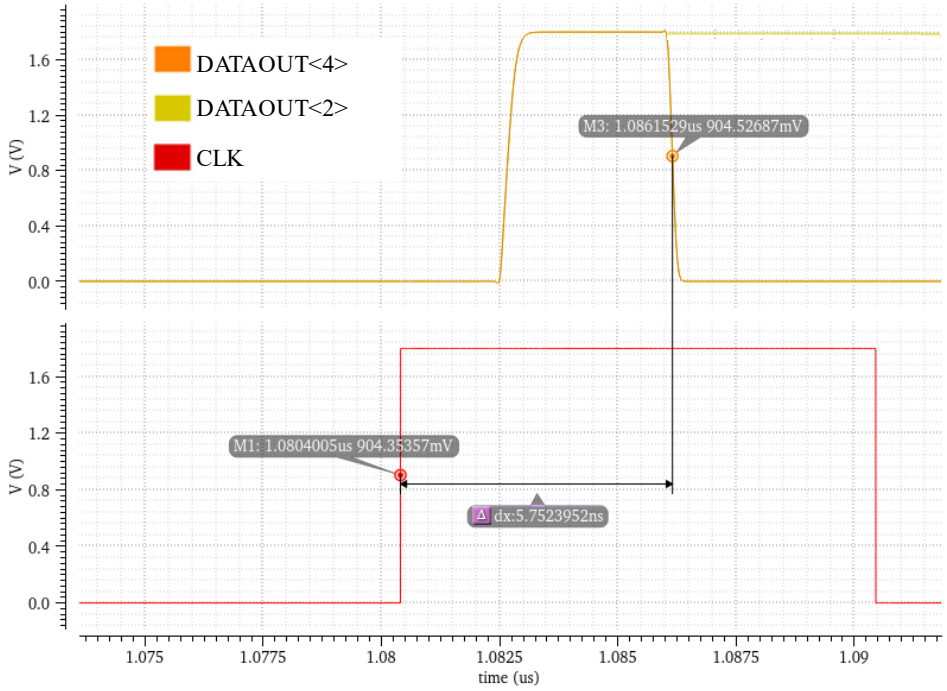

Figure 28: DATAOUT on address change with access time measurement

Though a certain number of passes to "0" is guaranteed to be on the output with any address change due to the amount of them determined by the memory size, more accurate results will always be reached while simulating the memory programmed in a checkerboard pattern. With the value stored in one address being the same for all DATAOUT bits, the generator takes the addresses that result in "0" on the output and measures the access time with them.

Regardless, upon clicking the "Access sim" button, the generator must first set up the simulation. Continuing from the manual generation of the extracted layout view of the ROM, the generator now creates and configures a config view. The config view is used to configure the testbench and bind the ROM extracted view to the symbol located in the schematic. When running a simulation of the config view, the results are determined from the memory layout. The generator uses hierarchy database functions that enable creating the view and setting the library, view, and stop lists. The resulting config view is shown in Figure 29 [16].

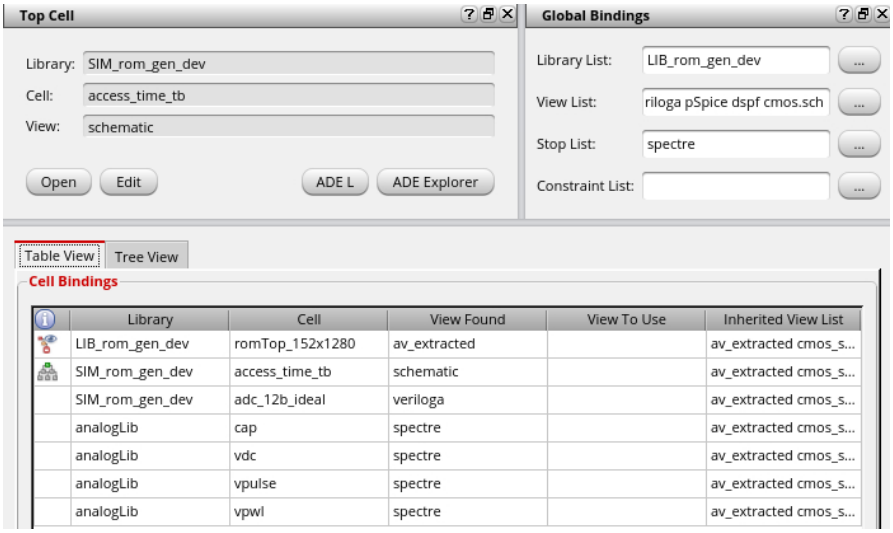

Figure 29: A generated config view

Subsequently, an ADE view is generated and set up with the use of OCEAN functions. The functions used include specifying the design, simulator, model, definition files, creating design variables, setting up the simulator options, corners, output expressions, and their specifications. The choice of simulator is limited by its compatibility with the used OCEAN functions, discussed in more detail in Chapter 6. The simulation is run right after the setup is completed, and the results are shown in the command interpreter window, when the simulation is completed. Simultaneously, the results are saved together with the simulation setup as a history setup that can be loaded in the ADE view afterwards. The date and time of the simulation run is saved and included in the name of the history setup for easy discernment of multiple runs. To customize the simulation, a new ADE view must be manually created as a history setup does not save all data from the simulation to conserve disk space.

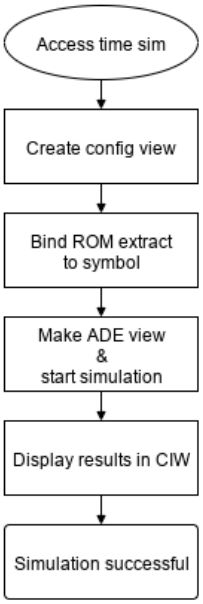

Figure 30: Access time simulation diagram

5.4. Decoding simulation

In the decoding simulation, the testbench has no sources. Instead, the wires that would lead to them are left unconnected and the sources are replaced by the vector generator. The vectors follow the pattern shown in Figure 11 with the lengths of the vectors changed to represent the generated memory. First, the row decoders are tested. The address input is changed consecutively, ignoring the first 4 bits, until it reaches the 256th row which is the maximum of the address. Simultaneously, both the left and right row signals (RL and RR) are checked against the address. A single active bit represents the row that is supposed to be active. Next, to test the column decoder, the row is set back to the first one, and the first 4 bits of the address start changing while the COLE signal is compared against them analogously. Any unexpected outputs are saved into a simulation error file by the simulator.

The simulation can be run right after the ROM generation is completed. When the "Decoding sim" button is used, the ROM generator first makes a generation file for the vector generator. The vector generator is then run through an invoked process that can run commands from a UNIX shell. Next, a run time for the simulation is read from a log file of the vector generator and the simulation is set up and run akin to the access time simulation. One difference is the inclusion of the vector file into the simulation. Finally, the simulation error file is checked, and the result is printed in the CIW. If the file is empty, the simulation was successful and if not, the path to the error file is conveyed.

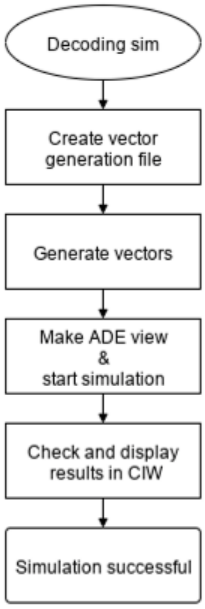

Figure 31: Decoding simulation diagram

5.5. Functional model for digital simulations and file conversion

Functional models for digital simulations replicate the basic functions of the circuit based on an HDL (Hardware Description Language). In the case of an asynchronous ROM, only two things are to be achieved. The functional model must be able to access a file containing data equivalent to those programmed in the actual memory, and it must put the data on its output when the address input is changed. Although the model can have other, more sophisticated capabilities, this provides a baseline that can be expanded if the need to do so arises. The model, shown in Figure 32, is generated as a Verilog file when the "Dig model" button is used. Figure 33 shows a manually written testbench for the memory model that compares the output data with an expected value from the memory file.

```
1
2  `timescale     1ns/1ns
3
4  module sg8_rom_8x128 (ADDR, DATAOUT, CE);
5
6    input  [11:0]        ADDR;
7    output  [7:0]        DATAOUT;
8    input                        CE;
9
10   parameter MEMORY_FILE = "addrFile.txt"; // file with memory content
11
12   reg [11:0]   i_addr;
13   reg  [7:0]   mem [0:127];
14   reg  [7:0]   data_word;
15
16 ////////////////////////////////////////////////////
17  // FUNCTIONAL MODEL
18  ////////////////////////////////////////////////////
19
20   initial
21     begin
22       $readmemb(MEMORY_FILE, mem);
23     end
24
25   // ROM input registers
26   always @(ADDR)
27     begin
28        i_addr <= ADDR;
29     end
30
31   assign DATAOUT = CE ? mem[i_addr] : 8'h0;
32
33 endmodule
34
```

Figure 32: Functional memory model for digital simulations

```
module tb_ROM;

  reg [11:0]    i_addr;
  reg           i_ce;
  reg  [7:0]    data_word;

  // Memory instance
  sg8_rom_8x128 u_rom   (.ADDR(i_addr), .DATAOUT(data_word), .CE(i_ce));

  task expectv (input expected_out);
    if (data_word!==expected_out) begin
      $display("DATA NOT MATCHING");
      $display("%b", data_word);
      end
    else begin
      $display("DATA MATCHES");
      $display("%b", data_word);
      end
  endtask

  initial
    begin
      i_addr = 12'b000000000000;  // Start at address 0
          i_ce   = 1'b1;             // Enable the ROM

      #1;
      expectv(8'b00000000);  // Check output value

      #1;
      i_addr <= 12'b000000000001; //Set adress to 1

      #1;
      expectv(8'b00000001);     // Check output value

      // Other adress values for checking...

    end

endmodule

```

Figure 33: Testbench of the memory model

The memory file used by the functional model is expected to be filled by groups of binary values separated by whitespace or newline symbols. The width of the binary value group corresponds to the width of the output data bus. Files of this type are commonly used, and various proprietary file generators and converters exist and are often employed by designers. Given that these generators are already capable of generating such files, no file converters were developed as part of the ROM generator. The same applies to the earlier described file used in ROM programming.

Chapter 6: Generator results

With the generator set up, ROMs of different sizes are generated for thorough testing. Most importantly, the generator behaves as expected, consistently creating functioning ROM designs that pass DRC and LVS checks. Their functionality is further verified by applying the simulation methods described in Chapter 5. The results gained from these simulations are shown and examined throughout this chapter.

6.1. Simulator constraints

SPECTRE, Cadence's SPICE-class simulator, is the first choice for use in tandem with seldom utilized Cadence tools whose compatibility with other simulators is hardly documented. Expectedly, every utilized OCEAN or OCEANXL function works as anticipated with SPECTRE, and the first simulation completes successfully without any errors. However, upon running a subsequent simulation of any kind, the simulator inexplicably freezes on netlist generation and returns no error messages. This bug occurs every time until the Cadence Virtuoso software is restarted and happens even if the code is generated directly from an ADE view and run without modification.

Another available simulator for the purposes of the generator is AFS (analog FastSPICE) provided by SIEMENS. Though it has its own design environment in development, the framework of non-graphical and automated control is not yet implemented. As such, it is unfit for use by the generator. Still, AFS can also be run by the ADE and consequently controlled by OCEAN commands. Unlike in the case of SPECTRE, subsequent simulations run with AFS finish successfully. Despite that, one crucial OCEAN function used to include the vector file, created by the vector generator, in the decoding simulation is not supported and therefore unavailable for use. This makes AFS unable to complete the decoding simulation with the use of OCEAN and OCEANXL commands.

With that said, the decoding simulation, DRC and LVS always pass as expected, signaling an error-less generation of both schematic and layout views. AFS will be used in the automatic access time simulation and SPECTRE will be used for the decoding simulation. While using two different simulators on one IC design is in general a bad practice as they can return varying values, the only relevant output from the decoding simulation is whether it passed or not. As such, the use of a different simulator has no effect in this case apart from enabling smoother function of the generator.

6.2. Access time

With the choice of simulator completed, memories with different ROM dimensions are generated. The dimensions were chosen to accurately cover most ranges. Step size increases with ROM size, as seen in Table 5, because of the likelihood of use of the generated ROM. The table shows the sizes of the memories in bits. The colored fields pertain to the memories that were generated and simulated while the rest of the fields belong to memories that have their access time later estimated based on the values from the generated memories. This estimation is made due to the amount of time that would be spent on their generation and simulation. The blue fields belong to memories with core halves that lean closer to square-like shapes and include both the largest and smallest memories to be simulated. They are expected to have lower access times than the rest of the memories with similar sizes. On the other hand, the orange fields represent memories with less optimal dimensions. Table 6 shows the resulting access time measured from the simulated memories in nanoseconds.

Table 5: Size reference [b] of generated and estimated memories

Columns \ Rows	8	32	56	80	104	128	152	176	200	224	256
128	1024	4096	7168	10240	13312	16384	19456	22528	25600	28672	32768
192	1536	6144	10752	15360	19968	24576	29184	33792	38400	43008	49152
256	2048	8192	14336	20480	26624	32768	38912	45056	51200	57344	65536
320	2560	10240	17920	25600	33280	40960	48640	56320	64000	71680	81920
384	3072	12288	21504	30720	39936	49152	58368	67584	76800	86016	98304
448	3584	14336	25088	35840	46592	57344	68096	78848	89600	100352	114688
512	4096	16384	28672	40960	53248	65536	77824	90112	102400	114688	131072
640	5120	20480	35840	51200	66560	81920	97280	112640	128000	143360	163840
768	6144	24576	43008	61440	79872	98304	116736	135168	153600	172032	196608
896	7168	28672	50176	71680	93184	114688	136192	157696	179200	200704	229376
1024	8192	32768	57344	81920	106496	131072	155648	180224	204800	229376	262144
1280	10240	40960	71680	102400	133120	163840	194560	225280	256000	286720	327680
1536	12288	49152	86016	122880	159744	196608	233472	270336	307200	344064	393216
1792	14336	57344	100352	143360	186368	229376	272384	315392	358400	401408	458752
2048	16384	65536	114688	163840	212992	262144	311296	360448	409600	458752	524288

Table 6: Access time [ns] of generated memories

Columns \ Rows	8	32	56	80	104	128	152	176	200	224	256
128	4.240					5.015					5.701
192	4.298	4.499									
256	4.338	4.540	4.729								
320	4.381	4.580	4.766	4.896					5.544		
384		4.622	4.809	4.936	5.065						
448			4.845	4.979	5.104						
512				5.018	5.142	5.272					5.958
640					5.225	5.354					
768						5.429	5.555				
896	4.755					5.513	5.644	5.767			
1024							5.715	5.857	5.979		
1280							5.876	6.002	6.139	6.270	
1536			5.524					6.155	6.294	6.424	6.589
1792									6.452	6.577	6.746
2048	5.463					6.217				6.740	6.916

These values are plotted in relation to memory size in Figure 34. To approximate other memories that follow the diagonal dimensions, a guideline defined by a power function was plotted. The suboptimal orange values hint at how one small dimension can make the ROM slower overall, but further analysis of the presented data is required. To accurately determine how access time changes with ROM size, the measured access time values from Table 6 were used. Firstly, the differences in access time between the blue generated memories, sharing either the number of rows or columns, were compared to determine changes caused solely by one or the other. An average value for the smallest possible change was calculated by taking neighboring memories and scaling the access time difference between them to 4 rows and 16 columns respectively. Comparison of the average changes implies that scaling with columns is approximately linear and consequently better suited for estimating the rest of the memory access time values from Table 6.

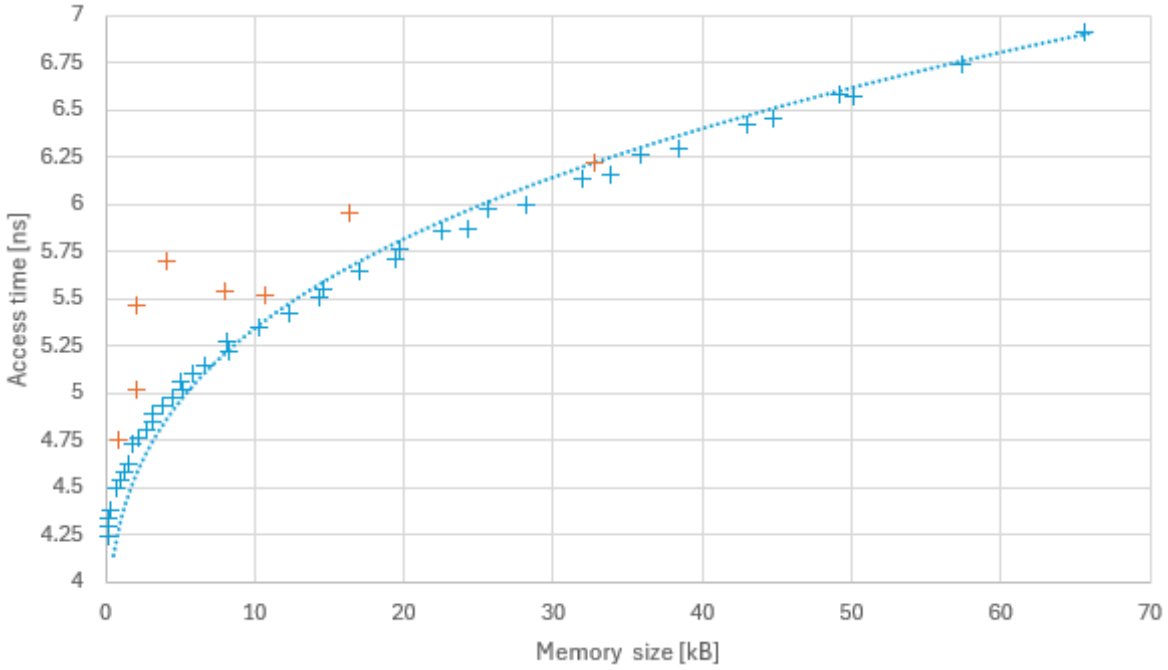

Figure 34: Access time values of generated memories

Table 7: Average change in access time per 16 columns

Rows	Average change per 16 columns [ps]
8	11.750
32	10.250
256	9.667
80	10.167
104	9.875
128	10.042
152	10.021
176	9.958
200	9.854
224	9.792
256	10.219
All values	10.143

Table 8: Average change in access time per 4 rows

Columns	Average change per 4 rows [ps]
192	33.500
256	32.583
320	28.611
384	24.611
448	21.583
512	21.167
640	21.500
768	21.000
896	21.167
1024	22.000
1280	21.889
1536	21.819
1792	20.979
2048	22.000
All values	23.848

With the average access time change value determined, the next step was to scale it back up to the rest of the memory sizes. Taking only changes from column amount into account, the estimates are based on the earlier measured diagonal values. Table 9 shows all access time values, including values for memory sizes that were not generated, in green. The approximation proves fairly accurate when it is compared to the values measured from unoptimal memory dimensions, where the highest difference equals 26 ps for the memory with 128 rows and 2048 columns. Figure 35 depicts the estimated values together with the earlier simulated ones. This combination forms an area of possible access time values that contains ROMs able to be created by the generator.

Table 9: Access time [ns] with estimated values

Columns \ Rows	8	32	56	80	104	128	152	176	200	224	256
128	4.240	4.458	4.648	4.774	4.903	5.015	5.149	5.280	5.422	5.540	5.701
192	4.298	4.499	4.688	4.815	4.943	5.069	5.190	5.321	5.463	5.580	5.755
256	4.338	4.540	4.729	4.855	4.984	5.110	5.230	5.361	5.503	5.621	5.796
320	4.381	4.580	4.766	4.896	5.024	5.150	5.271	5.402	5.544	5.661	5.836
384	4.422	4.622	4.809	4.936	5.065	5.191	5.312	5.442	5.573	5.702	5.877
448	4.462	4.663	4.845	4.979	5.104	5.231	5.352	5.483	5.614	5.743	5.917
512	4.503	4.703	4.886	5.018	5.142	5.272	5.393	5.524	5.654	5.783	5.958
640	4.584	4.784	4.967	5.099	5.225	5.354	5.474	5.605	5.736	5.864	6.021
768	4.665	4.865	5.048	5.180	5.306	5.429	5.555	5.686	5.817	5.945	6.102
896	4.755	4.947	5.129	5.261	5.387	5.513	5.644	5.767	5.898	6.027	6.183
1024	4.836	5.028	5.210	5.343	5.468	5.594	5.715	5.857	5.979	6.108	6.264
1280	4.998	5.190	5.372	5.505	5.631	5.756	5.876	6.002	6.139	6.270	6.427
1536	5.161	5.352	5.524	5.667	5.793	5.919	6.038	6.155	6.294	6.424	6.589
1792	5.323	5.515	5.686	5.829	5.955	6.081	6.201	6.317	6.452	6.577	6.746
2048	5.463	5.677	5.849	5.992	6.118	6.217	6.363	6.480	6.614	6.740	6.916

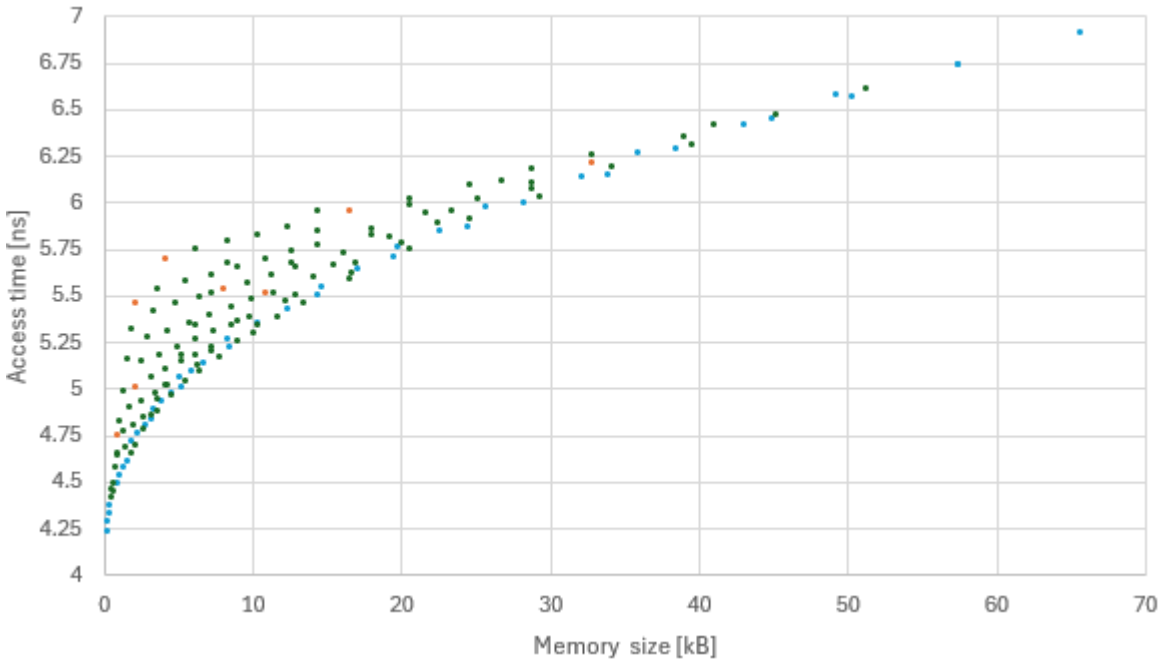

Figure 35: Access time values of generated and estimated memories

Some generated memories were also simulated across multiple corners to see how their behavior changes in critical temperatures, with high and low power supply voltages, and with different speeds of NMOS and PMOS transistors. Specifically, memories with a likely-to-be-used width of the output data bus (16-bit, 32-bit, and 64-bit) were chosen. The memories still show the programmed data on their output when read, even with a 10% supply voltage deviation under a range of temperatures from -40 °C to 150 °C. The values in Table 10 show that undervoltage and high temperatures have the largest influence on access time. With slow PMOS and NMOS corners this results in up to 78% increase of access time for the largest memory.

Table 10: Corner run access time [ns] values

Temperature [°C]	Nominal	-40	150	-40	150	-40	150	-40
Supply voltage [V]	Nominal	1.62	1.62	1.98	1.98	1.62	1.62	1.98
PMOS	Nominal	Slow	Slow	Slow	Slow	Fast	Fast	Fast
NMOS	Nominal	Slow	Slow	Slow	Slow	Slow	Slow	Slow
56x256 (16-bit) [ns]	4.728	7.003	8.299	4.83	6.081	5.675	6.767	4.157
128x512 (32-bit) [ns]	5.272	7.809	9.357	5.382	6.833	6.347	7.629	4.625
200x1024 (64-bit) [ns]	5.979	8.818	10.66	6.108	7.776	7.208	8.682	5.235
Temperature [°C]	-40	150	-40	150	-40	150	-40	150
Supply voltage [V]	1.62	1.62	1.98	1.98	1.62	1.62	1.98	1.98
PMOS	Slow	Slow	Slow	Slow	Fast	Fast	Fast	Fast
NMOS	Fast	Fast	Fast	Fast	Fast	Fast	Fast	Fast
56x256 (16-bit) [ns]	5.029	5.989	3.786	4.67	4.178	4.92	3.299	3.988
128x512 (32-bit) [ns]	5.599	6.712	4.199	5.223	4.646	5.512	3.645	4.448
200x1024 (64-bit) [ns]	6.362	7.635	4.752	5.912	5.261	6.254	4.121	5.024

6.3. Layout area

Another important parameter is the area of the generated memory layout. Unlike the size of the memory, the layout area doesn't scale directly with rows and columns. A considerable portion of the layout, see Chapter 5, stays identical for different memory sizes while only a portion of the layout scales directly with the memory dimensions. In this way, the already generated memories can have their layout areas measured and subsequently, the rest can be determined by calculating the area change caused by the difference in rows or columns. Adding one bit of output data equates to the sum of the widths of 16 cells and one cell divider. Similarly, 4 rows, the smallest possible change in height, take up the height of 4 cells. These values equate to 13.19 µm and 5.2 µm respectively. The area of the memories from the earlier shown tables is calculated from the smallest memory based on these values.

Table 11: Calculated memory area layout [μm^2]

Columns \ Rows	8	32	56	80	104	128	152	176	200	224	256
128	29317	35806	42295	48784	55273	61762	68251	74740	81229	87718	96370
192	36754	44889	53024	61159	69294	77429	85564	93700	101835	109970	120816
256	44191	53972	63753	73535	83316	93097	102878	112659	122441	132222	145263
320	51628	63055	74483	85910	97337	108765	120192	131619	143047	154474	169710
384	59065	72138	85212	98285	111359	124432	137506	150579	163652	176726	194157
448	66502	81222	95941	110661	125380	140100	154819	169539	184258	198978	218604
512	73939	90305	106670	123036	139402	155767	172133	188499	204864	221230	243051
640	88813	108471	128129	147787	167445	187103	206760	226418	246076	265734	291945
768	103687	126637	149588	172538	195488	218438	241388	264338	287288	310238	340838
896	118561	144804	171046	197288	223531	249773	276015	302258	328500	354742	389732
1024	133436	162970	192505	222039	251574	281108	310643	340177	369712	399246	438626
1280	163184	199303	235422	271541	307660	343779	379898	416017	452136	488255	536413
1536	192932	235635	278339	321042	363746	406449	449153	491856	534559	577263	634201
1792	222680	271968	321256	370544	419832	469120	518407	567695	616983	666271	731988
2048	252428	308301	364173	420045	475918	531790	587662	643535	699407	755279	829776

The layout area values from this table are not perfectly accurate as the fact that the memory layout is not a perfect rectangle is ignored as layout blocks have their areas planned and reserved mostly in rectangle shapes. Layers such as N Well, buried layer implants, and epitaxial growth layers surround the whole layout. Their minimum overhang is defined by DRC rules and adds to the dimensions of the calculated area. Assuming the generated ROM is used with a different layout design for other ICs, the effective area could be further reduced by a small amount with the elimination of the overhang. What follows is multiple generated ROM layouts with their dimensions shown beside them.

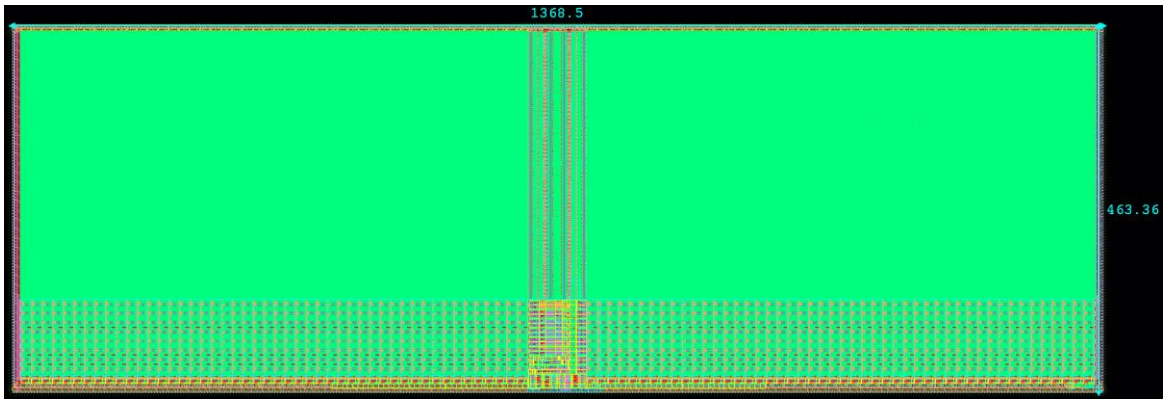

Figure 36: romTop_256x1532 layout

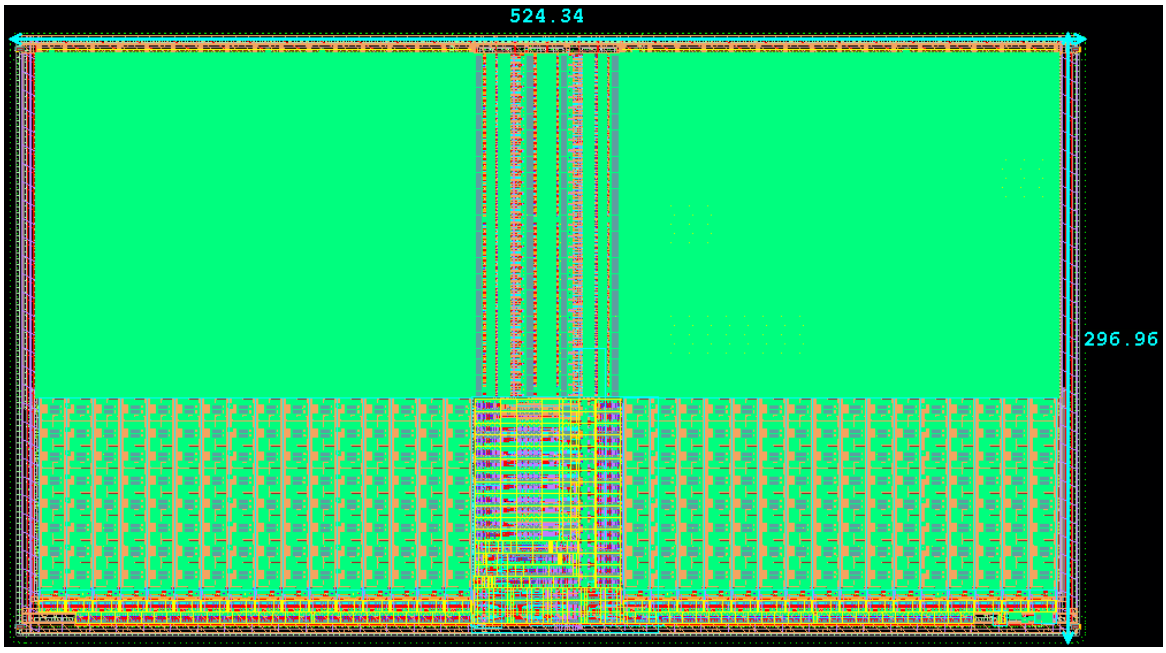

Figure 37: romTop_128x512 layout

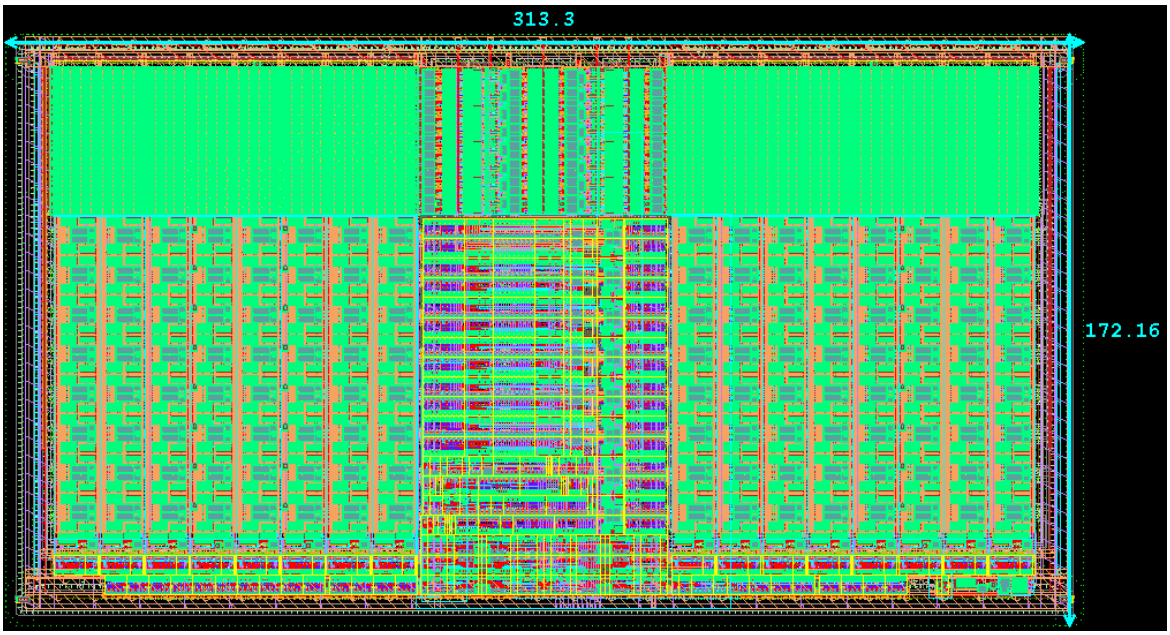

Figure 38: romTop_32x256 layout

6.4. Generation time

Another useful parameter to consider is the time it takes the generator to perform the steps of the generation process. Optimization, although not essential for this application, still has the benefit of improving ease of use. Simulations and layout integrity checks are not included in this analysis as they are independent of the design of the generator and rely on available licenses instead.

The most problematic part of generation is the memory core where the number of transistors can reach hundreds of thousands. The generation time can be improved by not placing parts individually where possible. One such possibility is making a separate layout cellview for a single row that is repeatedly inserted into the memory core instead of individual transistors. However, this cannot be the case for ROM programming because of the required connection of individual bit lines to the transistor drains, described in Chapters 4 and 5. Furthermore, checking errors and saving the schematic views also takes up a significant portion of the generation time that cannot be reduced. With this taken into consideration, the generation times range from approximately 39 seconds to 1 hour and 31 minutes. These values belong to the smallest and largest memory generated from Table 5.

Chapter 7: Conclusion

The main purpose of a memory generator is to save time by essentially skipping parts of the design process. The generator created in this project is based on a proven ROM design. The SKILL programming language was used in the development of the generator and enabled the integration of a graphical user interface into Cadence Virtuoso. The graphical interface includes the options to generate a memory of specified dimensions and program it, open the top layout of the generated memory to make a layout extract, run a post-layout access time simulation from the layout extract, run a decoding simulation, and generate a Verilog functional model for digital simulations, replicating the function of the memory. While using the generator, the time spent on designing a ROM with new dimensions is consequently reduced to a fraction of what it would be following a typical IC design workflow.

The generator was developed with only one ROM design and technology to conserve the efficiency of the design in terms of area and speed as much as possible. However, parts of the generator including the developed functions, simulation scripts, and user interface can remain identical for other potential designs. Thus, possible future additions of further ROM designs into the capabilities of the generator would require much less development time if the framework of this generator were to be used.

Created memories were simulated from the whole size range of the generator. Nominal simulations show that the access time of generated memories ranges from 4.2 ns to 6.9 ns. Based on the simulations a guideline and graph of estimates were plotted to show what the access time value will be for all memory sizes. Furthermore, multiple corner simulations were performed under which the memories continued to function with up to 78% increase in access time. With this amount of data, a tool able to accurately determine memory access time, before it is generated, could be developed in the future.

An important aspect to consider is the cost-effectiveness of the layout designs in smaller memories. Though the amount of money potentially saved, resulting from the time saved on designing the memory, is not negligible, the generated memory always decodes 12 address bits even if not all of them are used. The unused address decoders can take up a sizeable portion of layout space, especially with smaller memory sizes. Similarly, the sense circuit includes a number of transistors not used in the final design. Consequently, one possible area of improvement lies in saving more space by designing additional and more efficient layout views for these blocks and integrating them into the generator. However, the current layout design of the whole ROM is still very compact, following the minimum parameters of DRC rules. For example, a 25.6 kB memory with a 64-bit data out bus has an area of 369712 μm^2.

Other possible improvements include various additions to the graphical user interface. Although customization of the automated simulations could be achieved in this way, the number of menus and fields needed to provide the various choices would rapidly increase if meaningful options were to be realized. Still, adding the options to run the set-up access time simulation across different sets of corners might prove useful.

In summary, the developed generator is a complete and functional tool satisfying all objectives of this project with space for improvements via possible future additions. These improvements include reducing memory layout area for smaller dimensions, graphical user interface additions, a tool able to estimate access time from input dimensions, and perhaps most importantly an expansion of the generator for other memory designs.

Bibliography:

[1] LAN, Blue; STAR, Sung and JACQUES, Baudier. An MLC ROM With Inserted Redundancy and Novel Sensing Scheme. Online. Design & Reuse. P. 1. Available at: https://www.design-reuse.com/articles/36336/mlc-rom-inserted-redundancy-sensing-scheme.html.

[2] INTEGRATED CIRCUIT ENGINEERING CORPORATION. ROM, EPROM, and EEPROM Technology. Online. Available at: https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/rom-eprom-eeprom-technology.pdf.

[3] SINGH, Jawar; MOHANTY, Saraju and PRAHDAN, Dhiraj, 2013. Robust SRAM Designs and Analysis. Springer.

[4] PAVLOV, Andrei and SACHDEV, Manoj, AGRAWAL, Vishwani (ed.), 2010. CMOS SRAM Circuit Design and Parametric Test in Nano-Scaled Technologies. Springer.

[5] A Low Power 200 MHz Multiported Register File for the Vector-IRAM Chip, 2001. Online, Technical Report. Berkeley: University of California, EECS Department. Available at: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2001/5461.html.

[6] SAMSON, Giby and ANANTHAPADMANABHAN, Nagaraj. Low-Power Dynamic Memory Word Line Decoding for Static Random Access Memories. Online. Journal of Solid-State Circuits. Vol. 43, no. 11, p. 2524-2532. Available at: https://doi.org/10.1109/JSSC.2008.2005813.

[7] Allegro Microsystems [datasheet]. SG8 Asynchronous ROM Datasheet 2k of 64-bit words. 2020

[8] TURI, Michael and DELGADO-FRIAS, José, 2008. High-Performance Low-Power Selective Precharge Schemes for Address Decoders. Online. IEEE Transactions on Circuits and Systems II: Express Briefs. Vol. 55, no. 9, p. 917-921. Available at: https://doi.org/10.1109/TCSII.2008.923435.

[9] WICHT, B.; NIRSCHL, T. and SCHMITT-LANDSIEDEL, D., 2004. Yield and speed optimization of a latch-type voltage sense amplifier. Online. IEEE Journal of Solid-State Circuits. Vol. 39, no. 7, p. 1148-1158. Available at: https://doi.org/10.1109/JSSC.2004.829399.

[10] The D Latch. Online. In: All About Circuits. Available at: https://www.allaboutcircuits.com/textbook/digital/chpt-10/d-latch/.

[11] STEVENS, Kenneth; GOLANI, Pankaj and BEEREL, Peter, 2011. Energy and Performance Models for Synchronous and Asynchronous Communication. Online. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. Vol. 19, no. 3, p. 369-382. Available at: https://doi.org/10.1109/TVLSI.2009.2037327.

[12] AMRUTUR, B. and HOROWITZ, M., 1998. A replica technique for wordline and sense control in low-power SRAM's. Online. IEEE Journal of Solid-State Circuits. Vol. 33, no. 8, p. 1208-1219. Available at: https://doi.org/10.1109/4.705359.

[13] LI, Tong and KANG, Sung-Mo, 1998. Layout extraction and verification methodology for CMOS I/O circuits. Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175). Available at: https://doi.org/10.1109/DAC.1998.724485.

[14] CADENCE DESIGN SYSTEMS, INC. Cadence SKILL Language User Guide IC6.1.8. Online. Available at: https://support.cadence.com/.

[15] CADENCE DESIGN SYSTEMS, INC. Cadence Functions Reference. Online. Available at: https://bpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/0/367/files/2016/03/Cadence_functions_reference.pdf.

[16] CADENCE DESIGN SYSTEMS, INC. OCEAN+Reference. Online. Available at: https://www.eecis.udel.edu/~vsaxena/courses/ece697A/docs/OCEAN+Reference.pdf.

[17] LOW POWER 16-CHANNEL DATA SELECTOR FOR BIO-MEDICAL APPLICATIONS, 2014. Online. International Journal of VLSI design & Communication Systems (VLSICS). Vol. 5, no. 6, p. 1-8. Available at: https://aircconline.com/vlsics/V5N6/5614vlsi02.pdf.