Automatically Generated VLSI Memory (December 2003)

Allen TANNER, Jerry TAN, and Scott HOLMES

***Abstract*—Data storage devices (memories) of various sizes are required for most interesting integrated circuit designs. This paper describes an automatic set of tools for creating read only memory (ROM) and static random access memory (SRAM) for use in small-scale integrated circuit projects. These tools are intended for use by university students learning integrated circuit design.**

***Index Terms*—Student VLSI design, SRAM generation, ROM generation, Automatic VLSI circuit generation.**

I. INTRODUCTION

Most major universities now teach Very Large Scale Integrated (VLSI) design courses as part of their engineering curriculum. These courses typically involve a significant project component, where the students design a complete integrated circuit. By any metric, designing a complete, ready-to-fabricate circuit is a major undertaking. Meaningful projects tend to cluster around specialized controllers based on ALUs with supporting functional blocks. One of the difficulties students face, is that most interesting projects require storage in the form of either read only memory (ROM) or static random access memory (SRAM). Creating circuits to implement these memory devices is a significant undertaking which may detract from the overall project.

This effort is intended to provide a set of tools and techniques to allow university students to create ROM and SRAM structures in a convenient, reliable way. With these structures in place, more interesting and meaningful student projects will be possible.

The University of Utah CS6710 Digital VLSI Design Course is licensed to use the NCSU suite of Cadence tools. The CMOS devices are designed using the AMI 0.6 micron design rules. The projects are fabricated at MOSIS.

The VLSI memories that are automatically generated are compatible with the Cadence tools, the AMI 0.6 micron process and the MOSIS fabrication rules.

In order to construct either ROM or SRAM memories, a basic storage cell must be created and surrounded with infrastructure, e.g. address decoders, word drivers, pull-ups, amplifiers and power hookups. The user selects the memory configuration (size of the SRAM as well as size

Manuscript submitted December 8, 2003. This work is submitted in partial fulfillment of the requirements of the University of Utah, CS6710, Digital VLSI Design Course, taught by Dr. Erik Brunvand of the University of Utah Computer Science Department. The work was supported by the University of Utah CADE lab.

A. Tanner is graduate student at the University of Utah, Physics Department . (telephone: 801-588-1705, e-mail: atanner@ es.com).

J. Tan, BSEE UofU 2002, is a graduate student at the University of Utah, Electrical Engineering., UT 80108 USA (telephone: 801-706-4693, e-mail: ctan001@earthlink.net). He has previously worked at Agilent Technologies as an intern constructing a CDMA Wireless Phone Test Station.

S. Holmes, BSEE NAU 2003, is a graduate student at the University of Utah, Electrical Engineering., UT 80108 USA (e-mail: seh8@dana.ucc.nau.edu). He is working of a micro fuel cell for his MS project.

and contents of the ROM) and this configuration is translated by a Java program, *makemem,* into a circuit layout and Verilog structural file. These files are imported into Cadence for use by the students in their projects.

II. GOALS

A. Usability

A student learning to make VLSI circuits is faced with the daunting task of learning and using several complex tools. These tools evolved over many years to meet the needs of the VLSI design industry. They were written by many different authors and as a result each has its own quirky interface. Typically, the tools all have a GUI based on menu windows offering a bewildering array of choices. The choices offered in these menus, often have subtle consequences that are difficult for the naïve student to predict. It is not unusual for a student to make a menu selection only to find out, many steps later, that the wrong choice was made and several hours work must be repeated.

The goal of our project is to make life easier for the students and allow them to make more meaningful projects.

Our software, called *makemem* requires that the user provide only the essential design information necessary to create the memory cells. For SRAM this information is the size of the memory array in rows and columns. For ROM both the size and contents of the array are required. The contents of the ROM are stored in a file and can be described in binary, octal or hexadecimal radices. The ROM file is often a primary design document, so descriptive headers are allowed (encouraged). Comments are also supported to provide descriptions of the ROM contents. White space between data items may be used to partition the data into readable rows and columns.

A design guide has been created that explains the construction of the ROM and SRAM memory and gives a step by step recipe.

B. Safety

The software produces layouts, schematics and symbols that the students will use in their project designs. It is important that these products be very robust. We have extensively tested many configurations of both ROM and SRAM. We have designed both very large configurations, e.g. 1024x4 (much too large for student projects) and very small configurations (1x2) searching for pathologies. In addition to testing at the size margins, we have tested many of the common configurations, 16x16, 32x8, 32x128, etc. In addition to these pedestrian configurations, we have tested exotic configurations such as 17x6, 9x8 trying to look in all the corners of the design where nasty bugs often lurk. It should not be surprising that there were indeed problems in several of these strange places.

We tested all of the supported address lengths from 1 to 6 address bits. For each address bit length we examined configurations with both them maximum and minimum number of rows. For instance, with five address bits any number of rows between 17 and 32 can be configured. We tested configurations with both the minimum 17 rows and the maximum 32 rows. Each of these configuration exercises used the entire process from creating files with *makemem*, importing into Cadence, verifying the layout and schematic with DRC, Extract and LVS. Many configurations have undergone analog testing. We also created a Verilog memory test for the SRAM memory.

The ROM and SRAM configurations have been thoroughly tested and are ready for inclusion in student projects. However, it is assumed that the students using either ROM or SRAM will understand the VLSI designers mantra, “What has not been tested, will not work.” and they will exercise the ROM and SRAM functions in their projects as part of normal design verification.

Given our extensive testing and the project verification that a finished design should be subject to prior to fabrication, we are convinced

that our ROM and SRAM memories can safely be included in student VLSI design projects.

C. Simplicity

The *makemem* software is intended to be very simple to use. We have a bias against the Cadence style GUI with its complex set of choices. The bias predates our exposure to Cadence but was certainly reinforced by the using the Cadence tools over a period of several months. For this reason we have chosen to use a command line interpreter (CLI) style interface.

The following is the command line required to use *makemem* to create a 32x8 SRAM

```
>makemem –s 32 8
```

Similarly, *makemem* can be used to create a ROM file:

```
>makemem –r myFile
```

The contents of the ROM are specified in the file myFile.rom which is stored in the working directory.

The size of the ROM is also stored in that file. In the course of developing and testing *makemem*, the user interface was refined to make it simple to use. The original implementation required the user to specify the ROM size and the Cadence cell names. A typical session looked like:

```
>makemem –r myFile –s 32 128 –n Smyfile Rmyfile
```

Where the –s command specified the size and the –n command specified the Cadence top cell names. We found that the information on the size of the ROM was redundant (it is encoded in the data file) and the cell names were always the same, except when we had typos (which caused lots of confusion) so the interface was changed to default to the size of the ROM specified in the file and making the cell names the same as the ROM file with a capital S and a capital R prepended. (More about these two files later.) A sophisticated user can override the defaults using command line options.

During a review of the design, it was suggested that a log file be created. The program *makemem* logs pertinent information and sends this log to the console. Rather than create a separate set of controls (or defaults with overriding controls) logging uses the piping features of the operating system. To create a log in file *myROM.log* the user can enter:

```
>makemem –r myFile > myROM.log
```

We feel this is a simpler (better) user interface, than the Cadence style which is to put a log with a mystery name in a mystery directory.

D. Portability

The tools were designed for use by the university students in a university environment. Specifically, in the NCSU Cadence, AMI 0.6 micron, MOSIS environment. Portability was not a primary requirement. However, *makemem* is written in Java and will run on both Unix and Windows (both were used extensively during the development). The library of cells that are used to make the memory structures are specific to Cadence tools and AMI 0.6 micron process but they would serve as an excellent starting point should they need to be ported to other design tools or to other fabrication targets.

III. READ ONLY MEMORY

A. Introduction

Read Only Memory (ROM) is an important ingredient in modern computer design. It finds application storing data items that do not change. For instance, many commercial microprocessors intended for control applications can be purchased with ROM containing a fixed program.

Arithmetic processing also uses ROM for tables containing seed values of important functions. These seeds are expanded to the precision required by the application.

B. ROM Array

There are many ways to construct ROM storage devices. We have chosen a simple method where each bit in the ROM is assigned a fixed location in an array of rows and columns. At each location an Nmos transistor is used to represent a "one" bit and the lack of a transistor represents a "zero" bit. Figure one shows a 4x4 ROM with an Nmos transistor at every location. This ROM would contain all "one" bits.

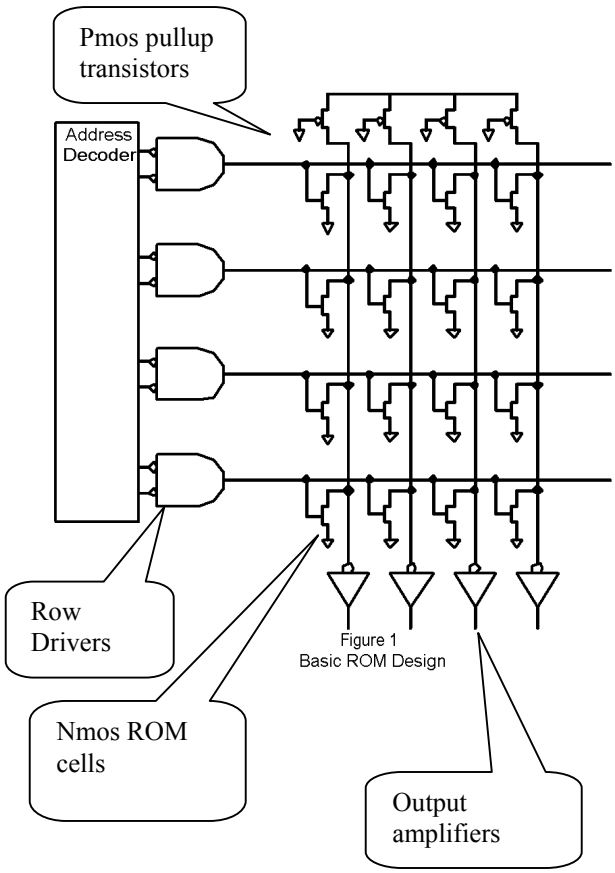


Figure 1
Basic ROM Design

C. Detailed Description

The row drivers drive only one row at a time. All the gates in a row of the Nmos ROM cell transistors are attached to the same row driver. When that row is driven, the transistors in that row each make a connection between a column and ground. This connection pulls the entire column to ground. Cells which do not have the Nmos transistors in them do not make the connection and the column is not pulled to ground. At the top of each column, is a Pmos transistor that pulls the column to Vdd. The Nmos transistors and the Pmos transistors compete for logic level of the column. Both the Nmos and the Pmos transistors are minimum size but the beta of the Nmos transistors is 2 to 3 times greater than the Pmos transistors, so the Nmos transistors win the contest and pull the entire column to ground.

The logic levels created by the contest between the Nmos and Pmos transistors are not very good, so an amplifier (inverter) is placed at the base of the each column to clean up the signal.

D. Performance

The ROM structure is very fast. The ROM timing begins when a valid address is presented to an address decoder circuit. Only four gates (two inverters, a nand-gate and the row driver nor-gate) delay the signal. The Nmos transistor witches quickly and there is a single inverter at the bottom of the column. Measurements included in the appendix indicate that the address to data out time is (hard to measure) but on the order of two nanoseconds for modest sized ROMS (32 columns). As the ROMS grow in size the capacitance of the word lines grow in significance. We have measured several large configurations to determine the effect of having many columns.

ROM configuration (rows x columns)	Read Timing Address to Output
64x32	2.2 ns
32x128	2.9 ns
4x1024	10.8 ns

The number of rows in the ROM structure also contributes to the maximum speed. However, we have limited the maximum number of rows to 64 and the variation in timing between 1 row and 64 is very slight, so the variation due to the ROM height is not considered when calculating ROM timing

E. Address Glitches

Our ROM does produce glitches (short duration and possibly ill-formed pulses) at the output. These arise when address transitions cause unintended word decodes. For instance, the transition from 01 to 10 must either pass momentarily through 00 or 11. The fast decoder circuits will respond and produce short pulses on the word decode lines 00 or 11. These glitches are unavoidable (commercial ROMS also produce them) without complex timing schemes which would defeat the purpose of providing an easy to use ROM device.

F. Additional features

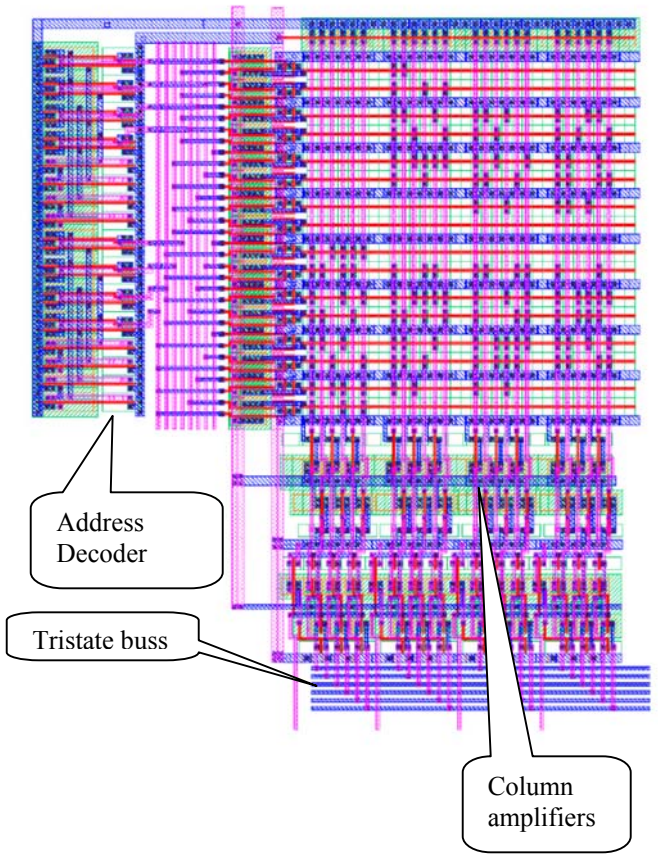
The ROM devices have two additional features that may be very useful.

A ROM can be equipped with a set of pass-transistor gates to implement a tristate output buss. The size of this bus can be specified separately from the size of the ROM and so that a ROM with many columns can have those columns grouped into a smaller size (easier to handle) buss. For instance a 128 column ROM can either produce 128 separate outputs or it can be configured to produce 8 output bits under the control of 16 tri-state enable inputs. This feature can either be specified in the ROM contents file or as a CLI option.

The ROM layout is produced in two pieces. One is called the structure and the other is the dockable ROM array. The intent of this feature is that the user will specify the size of the ROM (and thus the structure) early in the design cycle. This structure will include the address decoders, the row drivers, the Pmos pull-ups and the output amplifiers. This structure will most likely not change and it can be incorporated into the final layout. The contents of the ROM are another matter, they are stored in the Nmos transistor array which is created separately by *makemem*. This will allow the user to change the ROM contents very late in the design process to make final adjustments without disturbing the final layout.

G. Limitations

The basic Nmos cell is 2.7µ wide by 6µ tall. These dimensions force the nor-gate word drivers to be 6µ tall and the column amplifiers to be 2.7µ wide. Figure 2 shows a 4096 bit ROM configured in 16 rows and 24 columns. The columns are further grouped into an 6-bit tristate bus. Larger ROM configurations are but MOSIS imposes sever size limitations. The maximum number of rows currently implemented is 64.



Any number of rows (1-64) and any number of columns can be generated. However the width of the tristate busses must be an even number. That is, if you try to make a 5 bit tristate buss, makemem will issue an error and terminate.

If the number of columns is not evenly divisible by the tri-state group size, makemem will make as many groups as it can and put the remainder in the last group. For instance, 30 columns in groups of 8 will produce 3 complete groups of 8 and a final group of 6. The final group will be filled from the low order bit. This will all work fine provided the user does not try to do anything with the bits that do not exist.

IV. SRAM MEMORY

A. Introduction

Static Random Access Memory (SRAM) is widely used in integrated circuits. All of the popular microprocessors have elaborate on-chip caching schemes for improved performance. All of these caches use SRAM. Fifo and other high performance storage devices also tend to be based on SRAMs or SRAM technology. SRAM is more complex than ROM and tremendous design efforts have gone into making it smaller, more reliable and faster.

B. SRAM cell

The primary focus of SRAM design is to make a read/write storage device with as few transistors in as small an area as possible. Exotic processing and other tricks are used to accomplish this. In fact, very serious companies (Intel, Micron and others) who have at one point in their history been primary SRAM suppliers eventually have decide that making SRAMs is too big a distraction from their main business and have exited the market. Industry lore has it that once an engineer has gone over to the dark side (SRAM or even worse, DRAM) they never come back.

The basic SRAM cell uses six transistors in the configuration shown in figure 3.

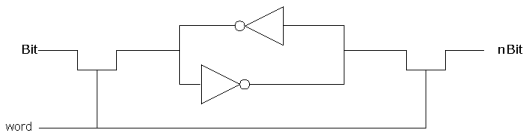


Figure 3
SRAM cell

Data is stored in the latch made by the pair of inverters and it is accessed by the Nmos transistors on the left and right of the cell.

C. SRAM Array

An SRAM cell is used to store each bit of the memory. These cells are organized like the ROM cells into columns and rows. Figure four shows the rows are controlled with word decode lines and that the columns have a pull-up at the top and an amplifier at the bottom. This amplifier is more sophisticated than the simple inverter used by the ROM array.

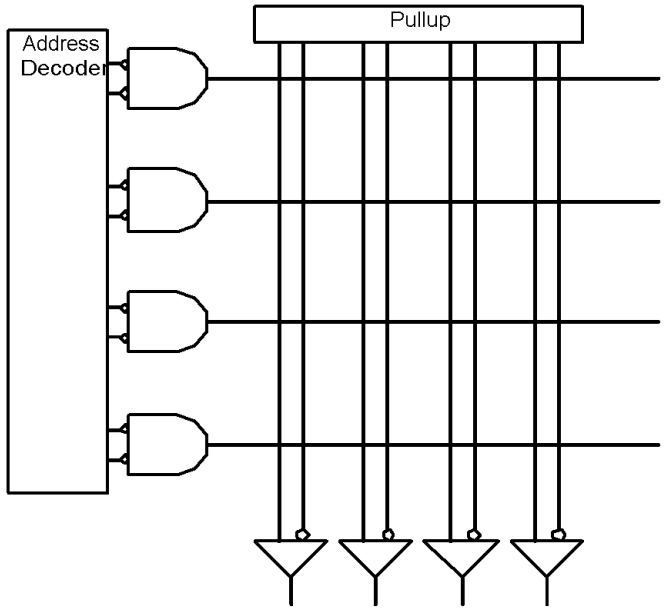


Figure 4
SRAM Design

The SRAM has two modes of operation, reading and writing.

1) SRAM Write Operation

The SRAM writes into the latch through the Nmos transistors on the left and right sides of the cell. The problem is that the coupled inverters will resist state changes. Their resistance is lowered by making them small (wimpy) and by overpowering them with a hefty write driver in the column amplifier.

2) SRAM Read Operation

The SRAM reads the value stored in the latch using a differential amplifier. The wimpy transistors in the SRAM cell do not produce good logic levels because 1) they have funny sizes (so we can write into them), 2) they have to pass through Nmos transistors which have non-ideal

(resistive) behavior when passing a "one" value and 3) they may be driving a significant capacitive load. The poor output levels that they produce must be translated into valid logic levels and a special amplifier is required to do the translation.

Each SRAM cell produces a differential pair of outputs, Bit and nBit. When the cell is storing a "one" the voltage on Bit is greater than the voltage on nBit. Conversly, when the cell is storing a zero the voltage on Bit is less than the voltage on nBit. The differences between the Bit and nBit signals are sensed with a differential amplifier of standard CMOS design which converts the small differences into reliable logic levels.

3) Address Glitches

It is important that only one SRAM cell be read at a time. If two cells are addressed at the same time, the Nmos transistors in both cells will short the latches in the two cells together and data in one or both of the cells may be changed.

The difficulty with SRAM is the same as with ROM. A simple addressing scheme produces glitches. Complex schemes, involving read timing can avoid the glitches at the cost of unwanted complexity for the user.

The solution is to design the SRAM cell to be "glitch tolerant." This involves sizing the transistors in the cross coupled inverter in proportion to the transistors in the Nmos access transistors. If the Nmos transistors conduct poorly, they will isolate their internal latches from glitches produced by other cells on the Bit and nBit lines during address changes. This requires that the write driver in the column amplifier be even heftier and that the differential amplifier be more sensitive. Fortunately, hefty drivers and sensitive differential amplifiers are easy to construct.

As this paper goes to press, we are continuing the investigation of the address glitch tolerance of our SRAM design. Very short addresses (in the 1 to 3 nanosecond range) appear to upset the SRAM cells. Waveforms of these upsets are included in the technical appendix.

D. Performance

1) Read Timing

Read timing in the SRAM is similar to the ROM. A valid address is presented and a short time later the contents of the SRAM cell will appear at the Q output. The primary difference is that the SRAM is more sensitive to the number of rows because the Nmos transistors associated with Bit and nBit are not designed to drive those signals poorly.

SRAM configuration (rows x columns)	Read Timing Address to Output
4x16	2.2 ns
4x32	3.2 ns
16x16	2.7 ns
4x1024	37.9 ns

2) Write Timing

Writing an SRAM requires a little care. The column write driver must not drive the Bit and nBit lines before the addresses stabilize. Figure 5 shows the waveform required for writing into the SRAM.

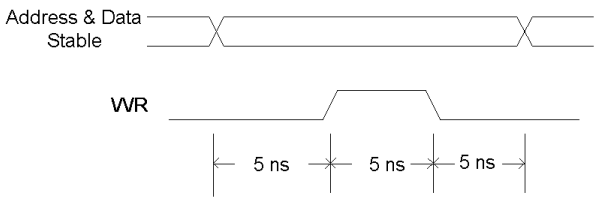


Figure 5
Write Timing

Shorter times are possible but they will depend on final layout signal quality.

E. Verilog functional view

Verilog test benchs are vital tools in the design process. They allow the user to extensively test the logic of the circuits under design. The difficulty with Verilog is that is assumes a switch model of CMOS transistors. This works fine for the ROM where the only accommodation to Verilog is to use a resistive Pmos transistor in the

column pullup. However, the SRAM uses a differential amplifier to clean up the Bit and nBit signals. This differential amplifier definitely does not fit the Verilog switch model. Cadence provides a solution. If a cell has a "functional" view defined it is used in place of the schematic for the Verilog simulation. This is intended to allow designers to use devices like differential amplifiers in schematics and to describe their operation in terms that Verilog can handle.

We created functional views of the SRAM cell and the column amplifier and changed the character of the signals between the cell and the amplifier.

Schematic view	Functional view	Use in functional view
nBit	Data_IO	Data path
Bit	Write	Write enable
Word	Word	Word address decode

Functional cellviews of the SRAM cells

```
// Verilog HDL for
// "memCells", "SRAM_samp" "functional"
// This functional view works with
// the functional view of the SRAM cell
// Bit is used to transmit WE
// and nBit is used a data path
module SRAM_samp (Q, Bit, nBit, RE, WE, D);
    output Q;
    inout Bit;    inout nBit;
    input RE;    input WE;    input D;

   bufif0(nBit, D, RE);

   assign Q = nBit;
   assign Bit = WE;

endmodule
```

o be inserted in a design and tested with a Verilog test bench.

```
// Verilog HDL for
// "memCells", "SRAM_down" "functional"
//  This functional view allows the SRAM cell to work
// correctly with verilog simulations
// which do not handle the analog nature of the device
// The functional view is automatically substituted by
// Cadence when it is simulating
//
module SRAM_down (Bit, nBit, word);
   inout Bit;  // used as WR from the SRAM_amp cell
```

Functional cellview of the column amplifier.

F. Size

The basic SRAM cell is 27µ wide by 6µ tall. This size was chosen for compatability with the ROM address decoder and word drivers. The result is that a 32x8 memory is approximately 300µ wide by 250µ tall and 32x16 is 600µ wide by 250µ. These seem to be very nice sizes for projects using MOSIS.

V. JAVA

The software for this project was written in the SunSoft programming language, JAVA, at the suggestion (insistence?) of the instructor, Erik Brunvand.

Java is the product of SunSoft. A public domain integrated development environment called NetBeans is available. NetBeans provides an identical user interface for Java software development on both Windows machines and on Unix machines.

Nine classes were created to provide all the functions necessary to support automatic creation of ROM and SRAM memory devices. These classes are described in the following paragraphs:

1) makemem.java

This is the main class. The program starts executing this class and it invokes the other classes as needed.

2) Process_args.java

This class processes the command line and reads the file containing the ROM contents, if necessary.

3) Verilog.java

The Verilog class creates a file with a ".v" extension containing a Verilog structure file. The modules used in the structure file were all created with the memCells library.
The ".v" file is imported into Cadence schematic and symbol cellviews using the CIW Import Verilog utility.

4) Layout.java

The Layout class creates a file with a ".gds" extension containing the two dimensional layout of the memory circuit. The ".gds" file is imported into a Cadence cellview using the CIW Import CIF facility.

5) AddrGen.java

This class creates the address decoder for both the SRAM and the ROM. Cells from the memCells library are placed and connected with and metal one and metal two wiring.

6) cifscan.java

The cifscan class reads the *memCells.gds* file from the working directory. This file contains all of the cell layouts required for both SRAM and ROM generation. The *memCells.gds* file is created using the Cadence Export CIF facility. (The users will not have to do this export.) The original source for export is a cellview called memCells. This cellview contains one instance of each of the cell layouts. The Export CIF facility assigns a number to each cell and that number must be used for subsequent references to the cell. The difficulty with these numbers is that they may change as new cells are added to the library. Early in the project the cells numbers were used as identifiers in the Java methods but they had to be changed every time a new assignment was made. Fortunately the Export CIF facility also includes the original cell name in the *memCells.gds* file. Cifscan reads the *memCells.gds* file and builds a table of the names and the assigned numbers. This allows the other classes to refer to the cells by their names, which do not change as updates are made.

7) IO_pads.java

This method writes a Cadence SKILL file with the extension ".il" containing instructions for placing the IO pads in the final layout. This feature saved a lot of development time because it makes the creation of complete layouts almost completely automatic.

8) helps.java

This class contains several input and output methods use as utilities by the other classes.

Listings of all the classes are provided in the appendix.

VI. Additional Features, Future Directions & Improvements

The tools developed for this project are complete and ready to use. However, there are improvements that could be made.

1) Increased Addressing

The current addressing is limited to 64 rows or fewer. This can be increased by making larger address decoder cells and integrating them into the *makemem* classes. The cells and modifications that would be required to support 128 and 256 rows are:

1. 4 input nand-gate which should be called NAND4.
2. 4 to 16 address decoder which should be called addr4.
3. A new decoder cell called LA3toA4

made from the existing addr3 cell and the new addr4 cell. This cell would decode seven address bits.

4. A new decoder cell called LA4toA4 made from two copies of the new addr4 cell. This cell would decode eight address bits.
5. The *makemem* classes *Layout.java* and *AddrGen.java* would have modified to reference and locate the new address decoder cells. *Process_args.java* currently checks that 64 or fewer rows are specified. This check would have to be changed.

2) Smaller cell size

The ROM and SRAM cells are fairly small given the process constraints. The ROM cells could be decreased in height from 6.0µ to 4.8µ by sharing the ground connection between cells. Of course, the address generators would all have to be shrunk to match this finer row pitch. The cells also used only one of the polysilicon layers available. Using the other might allow the SRAM cell to be made smaller.

3) SRAM tristate outputs

The ROM is equipped with tristate outputs and the SRAM is not. The ROM layouts use the tristate feature to multiplex wide column widths onto smaller busses. This is not as important in the SRAM because the SRAM cells are ten times larger than the ROM and it seems to us that very wide SRAM layouts are less likely to be useful.

4) Cifscan and friends

Makemem was written to allow the automatic generation of VLSI memories. It can be adapted to the generation of other VLSI structures. The *makemem* classes contain methods for creating busses that could be easily adapted to other uses. The *CifScan* class seems to us to be particularly useful. It allows a library of cells to be created and then accessed by name externally. We modified (simplied) the design of *makemem* once we realized the power of this capability.

5) IO_Pads

IO_Pads is a very convenient class. It creates named pads in a layout. A utility built around this class would allow pad rings to be more quickly integrated into final projects. A simple main class could be created that would read a user file of pin numbers and pin names and use IO_Pads to create a SKILL code file to put pins into the pad ring.

VII. CONCLUSION

The Automatically Generated VLSI Memory tools are ready to be used by the university students. In fact, two project in the current semester have already used the tools.

The first project used a 32x64 ROM with an 8 bit tristate grouping to store a sine wave for a pattern generator. Integrating the ROM into their project took less than an hour. In fact, the sine wave generator was originally billed as the most difficult part of the project and it became one of the easiest.

The second project is a videoscore board that is designed to support basketball games. It uses both ROM and SRAM and demonstrates a simple video generation scheme that may be useful to future VLSI students.