EE 4432 – VLSI Design

Layout and Simulation of a 6T SRAM Cell

Mat Binggeli

October 24th, 2014
Overview

The objective of this report is to describe the design and implementation of a 6-transistor SRAM cell. The basic operation and constraints of static RAM will be discussed, along with transistor sizing for device stability. The design will be covered using a symbolic schematic, as well as a physical device layout (both generated using Electric VLSI Design System). Finally, the read and write operations will be confirmed by simulation (using LTspice).

Basic Design

Historically, many different SRAM designs have been used (from 4T to 12T), but this report will focus exclusively on the standard 6T design. This type of RAM is one of the most common, due to its low leakage and compactness. A downside of the 6T SRAM is the need of more external circuitry to perform read and write operations, but when many memory cells are used with only one read and write driver for the whole grid, this is a good tradeoff.

The basic purpose of a memory cell is to hold a single bit of data, and this can be accomplished statically (without the need for refreshing) by using a pair of inverting gates. Since an inverter is the smallest CMOS gate, it is ideal for use in the core of an SRAM cell. In order to read from and write to this inverter pair, access transistors are also needed. Fig. 1 shows how these components are connected together and labels the different transistor pairs. The inverter pair contains pmos pull-up transistors $P_1$ and $P_2$, above nmos pull-down transistors $D_1$ and $D_2$ (also called driver transistors). The access transistor $A_1$ and $A_2$ connect nodes $Q$ and $Q_b$ (which contain the stored bit and its complement) to the bit lines.
**Operation**

The two modes of operation of the 6T SRAM cell, **read** and **write**, each require a different set of procedures to work. These steps are listed in Fig. 2.

---

**READ:**

1. Charge both *bit* and *bit_b* HIGH  
2. Let both *bit* and *bit_b* float  
3. Raise the *word* line

*Bit* will now contain the data value

---

**WRITE:**

1. Charge *bit* HIGH  
2. Let *bit* float  
3. Pull *bit_b* down to ground  
4. Raise the *word* line

*Q* will now contain a HIGH value  
*(swap *bit* & *bit_b* to write a LOW value)*

---

*Fig. 1 – Basic 6T SRAM cell.*

*Fig. 2 – Steps for 6T SRAM operation.*
Constraints

An important aspect of the 6T SRAM is the relative sizes of its transistors. By analyzing the read and write operations above, it’s shown that certain size relationships between the transistor pairs must be maintained for proper function.

In the first case (the read function), the bit-line that is connected to the Q-node which is LOW will be pulled down to ground. When this happens, the charge stored in the line goes through an access transistor (A) and a driver transistor (D). This current causes a slight increase in the voltage of the Q-node, which could potentially flip to HIGH if the increase was large enough. To prevent an unwanted flip, D must be stronger than A (the driver must be wider or shorter than the access transistor).

In the second case (the write function), the bit-line that is pulled down to ground must be able to drop the voltage of an adjacent HIGH Q-node, enough that the node flips to LOW. Since the opposite P transistor connects this Q-node to Vdd, it opposes the flip. Therefore, P must be weaker than A for the write to be successful.

These two constraints (D > A and A > P) give the basic necessary relationships between the transistor pairs for correct operation. D transistors must be the strongest, and P transistors must be the weakest, with A transistors in-between. In the following schematic and layout, this is accomplished by using drive transistors of size 8/2 (8 wide and 2 long), access transistors of size 4/2, and pmos transistors of size 3/3.
**Schematic and Layout**

The following figures show a 6T SRAM cell created in the Electric VLSI Design System. Fig. 3 contains a symbolic schematic along with an icon for the device. Fig. 4 shows the corresponding physical layout, and Fig. 5 gives this same layout in a 3D rendering that is tilted to better show the metal layers.

*Fig. 3 – 6T SRAM schematic and icon.*

*Fig. 4 – 6T SRAM physical layout.*

*Fig. 5 – 3D rendering of 6T SRAM cell.*
These designs passed a Design Rules Check (DRC) according to the MOSIS standards. A Network Consistency Check (NCC) was also performed, and the Metal-3 arc shown on the bottom of Figure 3 was added just to pass the NCC. This arc connects the two ground rails in order to match the single ground shown in the schematic and is not necessary when a complete SRAM array is made.

**Memory Arrays**

It can be seen in Figures 3 & 4 that the labeled connections are all attached to metal lines that extend the entire length or width of the cell. While these long rails are unnecessary in a single SRAM cell, they are very useful when creating a large memory array. In an SRAM structure with multiple columns (bits) and rows (words), the rails will connect to adjacent cells. Neighbors to the left and right will share the same *word* line and *Vdd*. They will also overlap on the *gnd* lines, so that there will be one *gnd* line for each cell rather than the two shown. Neighbors to the top and bottom will actually be mirrored so that each looks upside down to the other. This allows two vertical neighbors to overlap on the *Vdd* line, although they must not vertically overlap on the *word* line since each row will have its own *word* signal. Top and bottom neighbors will also connect on the vertical rails, sharing the same *gnd*, *bit*, and *bit_b* lines.

The use of vertical and horizontal rails makes sense when considering the way that memory arrays are accessed. For example, when writing a word to a memory address, each bit of the word is connect to a corresponding *bit* line (and inverted for a *bit_b* line) while the *word* line that corresponds to the desired memory address is driven HIGH. In this way, every cell in a column will receive the same value on its *bit lines*, but only the single cell that is also connected to the selected *word* line will write the data value.
Simulation – Layout Modifications

When simulating an SRAM cell, it is desirable to know the values of Q and Q_b because they contain the stored data and show whether a write is successful. Because of this, the schematic and layout were both modified in order to provide Q and Q_b connections which can be observed in SPICE simulations. The resulting schematic and layout are shown in Figures 6, 7, & 8.

Fig. 6 – Modified SRAM schematic and icon.

Fig. 7 – Modified SRAM physical layout.

Fig. 8 – 3D rendering of modified SRAM cell.
Simulation – SPICE Code

As shown in Fig. 2, performing \textbf{read} and \textbf{write} functions on the SRAM cell requires a special set of procedures, and being able to accomplish this in a SPICE simulation is a definite challenge. Perhaps the biggest difficulty is setting up a mechanism to simulate the floating \textit{bit} and \textit{bit\_b} nodes. SPICE throws out an error whenever a node is floating (for good reason), so special tricks have to be used to create this kind of situation.

\textbf{Voltage-Controlled Switches}

The method that is used in the following code (Fig. 9) takes advantage of voltage-controlled switches to connect the bit lines to either \textit{Vdd} or \textit{gnd} at specified times. In this way, either \textit{bit} or \textit{bit\_b} can be connected to \textit{Vdd} at a specified time between operations and then disconnected just before a \textbf{read} or \textbf{write}, allowing it to momentarily float (since the \textit{word} line is LOW). The same kind of switch can also connect either bit-line to \textit{gnd} before a \textbf{write} operation, leaving it tied to \textit{gnd} during the write, and then disconnect once the operation is finished. Using four of these voltage-controlled switches (connecting \textit{Vdd} and \textit{gnd} to each of the two bit-lines), the procedures for both the \textbf{read} and the \textbf{write} functions can be performed.

\textbf{Realistic Modeling}

There are still other devices that need to be included in the SPICE code, however. If a simulation is run using just the method described above, unpredictable and impossible results can occur. This is because a floating node is not a realistic model. In reality, if a section of metallization is connected through a switch to \textit{Vdd} and then allowed to float by the opening of that switch, it will momentarily hold a charge. This charge will then dissipate over a certain period of time. In SPICE, the ability to hold a charge can be modeled by a capacitor, and a resistor can model the dissipation. As seen in Fig. 9, a resistor and a capacitor were added to each of the bit lines. Their values are purposely large, since they are modeling leakage rather than actual devices.
PWL Control

With the switching devices created and a realistic SPICE model, the next step is to construct control devices for timing the operations. Timing the bit-lines is achieved by creating four voltage sources, each connected to the voltage control terminals of one the switches described above. By specifying the voltage of each of these control sources with a piece-wise linear (PWL) function, the four switches can be independently controlled and made to turn on and off at specified times. Another voltage source with a PWL function is also used to control the word line.

Timing Sequences

With all of the control devices now described, all that’s left is to decide what operations to perform and to generate the appropriate timing sequences. For this SRAM cell, there are two possible operations (read and write) and two possible values (HIGH and LOW). Therefore, a simulation that performs four operations can demonstrate the correct function of all possible actions that the SRAM needs to execute. For this project, the following simulation sequence was created: write HIGH, read (should output HIGH), write low, read (should output LOW).

First, the timing of the word line was setup similar to a clock signal with a 10ns period. For the first five nanoseconds, the word line is LOW. Between 5ns and 6ns, it ramps up from LOW to HIGH (from 0V to 5V). The word is HIGH from 6ns to 9ns, then it ramps down to LOW in the last nanosecond, and this repeats. Since the word line must be raised (HIGH) in order for a read or write to occur, the operation begins on the rising edge between 5ns and 6ns. Therefore, the bit-lines are prepared in the nanosecond before word begins to rise. For example, in the first period (0ns to 10ns) a write HIGH operation is performed. As described in Fig. 2, preparing the bit-lines means that bit needs to be charged HIGH and then allowed to float, while bit_b needs to be pulled to gnd and left there. To achieve this, the switch that connects bit to Vdd is closed at 4ns, then opened at 5ns, while the switch that connects bit_b to gnd is closed at 4ns and left closed through the write. Similar bit-line switching
(depending of the operation being performed) also occurs at 14ns, 24ns, and 34ns – just before the word line rising edges at 15ns, 25ns, and 35ns. The word line falls to LOW for the last time at 40ns, completing the last read operation, and the simulation transient analysis ends at 45ns. These timing values are all shown in Fig. 9.

```
# Connect 5V to vdd
vdd vdd 0 DC 5

# Physical modeling necessary for floating nodes
R1 bit 0 1meg
R2 bit_b 0 1meg
C1 bit 0 100f
C2 bit_b 0 100f

# Define switches to connect each bit line to vdd and gnd
Sw_bit_HI vdd bit SbH 0 switch
Sw_bit_LO bit 0 SbL 0 switch
Sw_bitB_HI vdd bit_b SbBH 0 switch
Sw_bitB_LO bit_b 0 SbBL 0 switch

# Set precise timing of switches to perform read and write operations
vSw_bit_HI SbH 0 PWL(0n 0 3.9n 0 4n 1 4.9n 1 5n 0 13.9n 0 14n 1
+14.9n 1 15n 0 33.9n 0 34n 1 34.9n 1 35n 0)
vSw_bit_LO SbL 0 PWL(0n 0 23.9n 0 24n 1 30.9n 1 31n 0)
vSw_bitB_HI SbBH 0 PWL(0n 0 13.9n 0 14n 1 14.9n 1 15n 0 23.9n 0
+24n 1 24.9n 1 25n 0 33.9n 0 34n 1 34.9n 1 35n 0)
VSw_bitB_LO SbBL 0 PWL(0n 0 3.9n 0 4n 1 10.9n 1 11n 0)

# Set precise timing of word line for 4 separate 10ns operation cycles
vword word 0 PWL(0n 0 5n 0 6n 5 9n 5 10n 0 15n 0 16n 5 19n 5 20n 0
+25n 0 26n 5 29n 5 30n 0 35n 0 36n 5 39n 5 40n 0)

# Use predefined voltage-controlled switch model
.model switch Vswitch()

# Perform transient analysis over period that includes 4 operation cycles
.tran 0 45n

# Include models for the nmos and pmos transistors
.include "C5_models.txt"
```

Fig. 9 – SPICE code for SRAM simulation.
Simulation - Results

Using the SPICE code above, simulation results were similar for the schematic and the layout. Figures 10 and 11 show output for the full 45ns simulation. For clarity, Fig. 10 shows just the input signal voltages: bit, $bit_b$, and word. Fig. 11 shows the resulting node voltages that hold the data bit: $q$ and $q_b$. Fig. 12 zooms in to show all of these voltages during the write HIGH operation, and Fig. 13 zooms in to the following read. Similarly, Fig. 14 displays the write LOW, and Fig. 15 gives the subsequent read.

Fig. 10 – Full simulation, input signals.

Fig. 11 – Full simulation, $q$-nodes.
Fig. 12 – Write HIGH simulation.

Fig. 13 – Read HIGH simulation.
Fig. 14 – Write LOW simulation.

Fig. 15 – Read LOW simulation.
Conclusions

The simulation results above demonstrate that this 6T SRAM cell design operates correctly for all four necessary functions: write HIGH, write LOW, read HIGH, and read LOW. As Fig. 12 shows, the write HIGH function is successful in flipping the bit, changing $q$ from LOW to HIGH. It’s also shown that $q$ reaches a voltage within 10% of HIGH at about 0.2ns after the word line finishes rising.

Similarly, Fig. 14 shows that the write LOW function is successful, flipping the bit again and changing $q$ from HIGH to LOW. Here, it’s worth noting that although Fig. 14 is virtually identical to Fig. 12 (just with the bit-lines and the q-nodes reversed), during the write LOW $q$ drops to 0V, and it’s there before the word line finishes rising. While this may make it might seem like writing a LOW is faster than writing a HIGH, looking at $q_b$ shows that it still takes 0.2ns after the word line finishes rising to come within 10% of 5V. Whatever data value is being written, the same period of time is needed for the cell to become stable.

To determine the success of the read functions, it’s necessary to analyze the bit-lines as well as the q-nodes. For the operation shown in Fig. 13, $q$ is HIGH so the read should result in bit staying HIGH. In fact, bit does stay HIGH, while $bit_b$ drops to LOW, as it should. However, another important criteria for a successful read is that the operation does not cause an unwanted flipping of the bit. In this case, $q_b$ does raise in voltage slightly, but it does not go above 0.5 V. The bit remained stable while the value was read, so the read was a success. Fig. 15 shows an identical situation with the bit-lines and q-nodes reversed. Here, $q$ is LOW, so the read should result in bit dropping LOW. It does drop to 0V while the bit remains stable, so this read was a success, as well. It can be noted for the read function that the dropping bit-line doesn’t fall within 10% of 0V until about 2ns after the word line finishes rising.
Considering the operation delay seen in these simulations, it appears that the **read** function takes about ten times as long to reach stability as the **write** function. This is assuming that the cell data is stable when the q-nodes are within 10% of their final values, and that the bit-lines are stable to read when their voltages are within 10% or their final values. It is important to note that the longest delay (from the dropping bit-line in the **read** function) will change if the values of resistance and capacitance shown in Fig. 9 are changed. However, trying to set ideal values for these components is a trade-off.

Having a higher value for RC (the time constant) makes the dropping bit-line take longer to discharge, so **read** delay is increased. Having a lower RC value makes the dropping bit-line fall faster (decreasing **read** delay), but it also causes the holding bit-line to fall faster. The holding bit-line is supposed to stay near HIGH during the **read**, but it will drop to an invalid logic level if the RC value is low enough. A low RC value also makes the floating bit-line weaker during the **write** function, which increases the **write** delay. Therefore, a good RC value will balance **read** and **write** delay, while holding the bit-line long enough to send the **read** output.