CHAPTER X

MEMORY SYSTEMS

READ MEMORY NOTES ON COURSE WEBPAGE

CONSIDER READING PAGES 285-310 FROM MANO AND KIME
OTHER USEFUL RAM MATERIAL AT ARS TECHNICA
A number of different types of memories and programmable logic devices exist.

- Random-access memory (RAM)
- Read-only memory (ROM)
- Programmable logic devices (PLDs)
  - Programmable logic arrays (PLAs)
  - Programmable array logic (PAL)
  - Programmable gate arrays (PGAs)
- Programmable sequential arrays (PSAs)
- Field-programmable gate arrays (FPGAs)
- Due to time limitations, we will only cover RAM.
Two main categories of random-access memory (RAM) exist.  
- Static memory or static RAM (SRAM)  
  - Information bits are latched such as with a latch or a flip-flop.  
  - Typical SRAM implementations require 4 to 6 transistors.  
- Dynamic memory or dynamic RAM (DRAM)  
  - Information bits are stored in the form of electric charges on capacitors.  
  - The capacitors will discharge over time.  
  - Refreshing the memory cell is required before the capacitor has discharged to much of the electric charge.  
  - Most DRAM implementations use 1 transistor and 1 capacitor.
An inefficient SRAM bit cell can be formed as follows.

Select (word line)

- How many transistors required for this design?
  - 2*4 for inverters + 2*2 for TGs = 12 transistors.
  - Very expensive in terms of silicon real estate!!!
The structure for a 6 transistor implementation of an SRAM 1-bit cell is as follows. (We will refer to this as the “6T” design)

Select (word line)

- The select, or word line, chooses the bit cell when high.
- When selected, the new $D/\overline{D}$ is latched into the feedback loop.
Of course, the previous SRAM cell structure can be drawn as follows, replacing each inverter with 2 transistors.

Select (word line)

6T design

\[ V_{DD} \]

\[ D \]

\[ \overline{D} \]
• A 4 transistor design for an SRAM bit cell is as follows.

Select (word line)

4T design

\[ V_{DD} \]

• Notice replacement of pMOS transistors with load resistors.
• This is for your own information. We won’t be testing on the 4T design.
A dynamic RAM cell stores the bit as a charge in a capacitor. This bit must be refreshed periodically (>100s of times a second).

Select (word line)

Transmission gate opens when selected to charge or discharge capacitor. This charge stores the bit.

How many transistors required for this design?

- 2*1 for TG and 2*1 for inverter = 4 transistors.
- Still expensive considering the extra refresh circuitry required!
The capacitor charging structure can be simplified as follows.

Select (word line)

Transmission gate opens when selected to charge or discharge capacitor.

This charge stores the bit.

This structure for a DRAM bit cell is what is used in practice in real implementations.

Very little chip real estate is used!!!
Having developed bit cells, either SRAM or DRAM bit cells, they can now be pieced together forming a memory unit.

What do we want to specify in the design of a memory unit?

- The number of bits.
  - This gives the total number of bits that the memory unit can store.
- The grouping of bits into words.
  - Accessing 1 bit at a time might be inconvenient, so, grouping bits into words is often done.
  - Common examples of word bit sizes are 4, 8, 16, 32, and 64.
- The number of words in the memory unit (addressable words).
  - This is a function of the word size and total number of bits.
In describing the capacity of a memory unit, the following is used:

- # addresses x word size
  - Example: 1Mx8

If a memory unit is described as 1Mx8, then it has:

- \(1M = 2^{20} = 1048576\) addresses,
- 8 bits per word at each address location,
- 8 data lines for the 8 bit words,
- 20 address lines to specify the \(1M = 2^{20} = 1048576\) addresses,
- and \((1048576)(8) = 8388608\) bits in the entire memory unit.
Some further examples of memory descriptions are given below.

- Note that the last four columns are all described with the information in the first column.
- Try to fill in the empty cells for the last two rows.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Total bits</th>
<th># of addresses</th>
<th># address lines</th>
<th># data lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Mx8</td>
<td>8388608</td>
<td>1048576</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>1Kx4</td>
<td>4096</td>
<td>1024</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>2Mx4</td>
<td>8388608</td>
<td>2097152</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>4Mx1</td>
<td>4194304</td>
<td>4194304</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>2Mx32</td>
<td>67108864</td>
<td>2097152</td>
<td>21</td>
<td>32</td>
</tr>
<tr>
<td>16Kx64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8Mx8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Below is a general block diagram for a memory unit.

![Block Diagram]

- The $k$ address lines access a word in the memory for input or output.
- To simplify drawing, we now form buses of $n$ (or $k$) lines.

$n$ data output lines

$k$ address lines

Read

Write

$2^k$ words

$n$ bits per word

$n$ data input lines

$n$ data output lines
To conserve pins, the following layout is more common in practice.

- The data lines are both input and output lines (not simultaneously).
- This is done by using tristate buffers to form a tristate bus (or sometimes referred to as a three-state bus).

```
  k address lines → k
  Read/Write → R/W
  Enable → CS

Memory unit
2^k words
n bits per word

n data lines
(input and output)
```
The bit cells are arranged in matrix. (more efficient!)

Row and column decoders access specific bit cells.
In read mode:

- Row decoder “activates” all bit cells in that row.
  - Each bit cell in the row outputs their stored bit.
  - Column decoder takes the bit from only one column of the activated row.

In write mode:

- Row decoder “activates” all bit cells in that row.
  - Each bit cell in the row effectively outputs their stored bit.
- Column decoder selects the appropriate column and writes the input bit.
  - SRAM: This writing is done by “overpowering” what is being read by the bit cell with a stronger voltage/current.
  - DRAM: This writing is done by recharging the capacitor for writing a 1 or discharging the capacitor for writing a 0.
• Now include $k$ address lines, 1-bit data line, Enable, and $\sim$Read/Write.
Things to note about the previous implementation.

- The \( k \) address lines are split into two parts (not necessarily equal parts)
  - \( p \) of the \( k \) address lines are sent to the row decoder.
  - \( k-p \) of the \( k \) address lines are sent to the column decoder.
- There is only one bit line for input/output, D.
  - If we desire multiple bits, say \( n \) bits, for each address location, the entire structure must be duplicated \( n \) times.
- The enable line, or often called chip select (CS), either allows the transmission gates to be closed or prevents them from being closed. This makes it so that the D can be in one of three modes;
  - reading, writing, or high impedance.
- The \(~\text{read/write}\) line controls which transmission gate is closed.
In general, given a certain size memory chip, such as a 1Mx1 memory chip, we would not know how the internal matrix is configured.

For a **1Mx1 memory chip**, we know it has **20 address lines** (for our purposes in any case, there are exceptions in the real world). Any combination of address lines for the row and column decoder could be used to form the matrix.

- Example: **10 row** address lines and **10 column** address lines for a **1024x1024 matrix** of bit cells.
- Example: **12 row** address lines and **8 column** address lines for a **4096x256 matrix** of bit cells.
- Example: **5 row** address lines and **15 column** address lines for a **32x32768 matrix** of bit cells.
As noted, the described internal bit cell matrix structure accesses only 1 bit at a time.

Multiple bits to form an $n$-bit word could be accessed in a few different methods.

- One method is to have the **column decoder** select a set of $n$ columns **simultaneously** to form the word.
  - This only works if the entire word is stored in one row. Hence, there should be a multiple of $n$ columns in the bit cell matrix.
  - For instance, the given bit cell matrix with 8 columns could easily have words of size 1, 2, 4, or 8.
- Another method, though arguably very similar, is to **duplicate** the entire **bit cell matrix $n$ times** to form the $n$-bit word.
The following illustrates these approaches for accessing a 4-bit word.

- On the left, **four column bits** from the **second row** are selected to form the **4-bit word**.
- On the right, the **first column bit** from the **second row** for 4 duplicates of the bit cell matrix are accessed to form the **4-bit word**.
• What if we don’t have the right type of memory chips to build a desired computer system?
• Must learn to use combinations of existing memory chips to form a memory system according to specifications.
• Example: We want a $1M \times 8$ memory system but can only cheaply buy $1M \times 4$ memory chips. What to do?
  • Become Bill Gates so you can afford it.
  • Get a better job.
  • Go back to storing your valuable information on little pieces of paper in your pocket. (be careful of washing machines!!!)
  • Design a memory system that uses $1M \times 4$ memory chips but logically forms a $1M \times 8$ memory system.
• Build a 1Mx8 memory system using 1Mx4 memory chips.
  • First identify the specifications for the 1Mx4 memory chips and the desired 1Mx8 memory system?

<table>
<thead>
<tr>
<th>Memory</th>
<th>Total bits</th>
<th># of addresses</th>
<th># address lines</th>
<th># data lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Mx8</td>
<td>8388608</td>
<td>1048576</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>1Mx4</td>
<td>4194304</td>
<td>1048576</td>
<td>20</td>
<td>4</td>
</tr>
</tbody>
</table>

• As the table shows, the # of addresses and # of address lines are the same for both. So, we do not have to change that.
• The number of data lines for the 1Mx8 are double that of the 1Mx4 as well as the total # of bits stored in the memory.
  • Therefore, we require two 1Mx4 memory chips arranged with the same address lines and concatenated data lines.
Forming the described 1Mx8 memory system with 1Mx4 chips, the following 1Mx8 memory system can be designed.

A\textsubscript{19}:A\textsubscript{0} → 1MX4

Enable \quad \sim R/W → 1MX4

D\textsubscript{0} \quad D\textsubscript{1} \quad D\textsubscript{2} \quad D\textsubscript{3} → 1MX4

D\textsubscript{4} \quad D\textsubscript{5} \quad D\textsubscript{6} \quad D\textsubscript{7} → 1MX4

CS \quad R/W \quad CS \quad R/W
• Build a 2Mx4 memory system using 1Mx4 memory chips.
  • First identify the specifications for the 1Mx4 memory chips and the desired 2Mx4 memory system?

<table>
<thead>
<tr>
<th>Memory</th>
<th>Total bits</th>
<th># of addresses</th>
<th># address lines</th>
<th># data lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>2Mx4</td>
<td>8388608</td>
<td>2097152</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>1Mx4</td>
<td>4194304</td>
<td>1048576</td>
<td>20</td>
<td>4</td>
</tr>
</tbody>
</table>

• As the table shows, the # of data lines are the same for both.
• There is an extra address line in the 2Mx4 giving double the # of addresses and double the # of bits
  • Therefore, we require two 1Mx4 memory chips arranged where one address line is used to decode which 1Mx4 chip is enabled.
The following implements the desired 2Mx4 using 1Mx4 memory.

- Notice that $A_{20}$ is used to activate one of the 1Mx4 memory chips.
• Goal is to abstract the memory model so that a programmer of the system does not need to know the physical layout of the memory.

• Below is an example $2^{32}$ addressable memory map of byte. This, for instance, could be implemented with a **4Gx8** memory system.

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Value (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0x43</td>
</tr>
<tr>
<td>0x00000001</td>
<td>0x78</td>
</tr>
<tr>
<td>0x00000002</td>
<td>0x12</td>
</tr>
<tr>
<td>0x00000003</td>
<td>0x04</td>
</tr>
<tr>
<td>0xFFFFFFFF</td>
<td></td>
</tr>
</tbody>
</table>

...
How should a multiple byte word be stored?

The two most common orderings are

- **Big endian**
  - The address is of the most significant byte location.
  - Sun Workstations and Macs use this ordering.
- **Little endian**
  - The address is of the least significant byte location.
  - Intel x86 architectures use this ordering.

Origins of the terms big and little endian.

- *Gulliver’s Travels*: Feud between the two mythical islands, Lilliput and Blefescu, over the correct end (big or little) at which to crack an egg.
• An example of a 4-byte word stored in big endian order
  
  • 0x12345678 would be stored as

  | 0x10010000 | 0x12 |
  | 0x10010001 | 0x34 |
  | 0x10010002 | 0x56 |
  | 0x10010003 | 0x78 |

  12345678

• An example of a 4-byte word stored in little endian order

  • 0x12345678 would be stored as

  | 0x10010000 | 0x78 |
  | 0x10010001 | 0x56 |
  | 0x10010002 | 0x34 |
  | 0x10010003 | 0x12 |

  78563412
Let’s assume we have a memory system with 32 bit words using little endian byte ordering.

- `.word n_{10}`
  - Store the value $n_{10}$ in memory as a 32 bit word binary value.
- `.byte n_{10}`
  - Store the value $n_{10}$ in memory as a byte value (8 bits).
- `.asciiz "string"`
  - Store the ASCII string in memory with a null character.
- `.space n_{10},`
  - Skip the next $n$ bytes.
- `.align k`
  - Force loader to go to next $2^k$ byte boundary.
One example assembly memory directive set could be as follows.

Assume loader starts at 0x10010000

```
.data
.word 24
.byte 7
.ascii "help"
```

Notice null character
Continuing with the previous example.

```
.align 2
.word 186548547
.byte 52
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10010008</td>
<td>0x70</td>
</tr>
<tr>
<td>0x1001000C</td>
<td>0x43</td>
</tr>
<tr>
<td>0x10010010</td>
<td>0x1E</td>
</tr>
<tr>
<td>0x10010010</td>
<td>0x0B</td>
</tr>
<tr>
<td>0x10010010</td>
<td>0x34</td>
</tr>
</tbody>
</table>
Continuing with the previous example.

```
.align 2
.byte 16
.space 3
.byte 67
```

<table>
<thead>
<tr>
<th>0x10010010</th>
<th>0x34</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10010014</td>
<td>0x10</td>
</tr>
<tr>
<td>0x10010018</td>
<td>0x25</td>
</tr>
</tbody>
</table>

Notice the difference between `align` and `space`.