Introduction

Storage components store data and perform simple data transformations, such as counting and shifting.

 Registers, counters, register files, memories, etc.

 Register: a group of binary cells (FFs) suitable for holding binary information.

 In addition to the FFs, a register may have combinational gates that control when and how new information is transferred into the register.

 Counter: a register that goes through a predetermined sequence of states upon the application of input pulses.

 The gates in a counter are connected in such a way as to produce a prescribed sequence of binary states in the register.

 Memory unit: a collection of storage cells together with associated circuits needed to transfer information in and out of storage.

 For example, SRAM & DRAM.

Registers

 A register can be viewed as a bitwise extension of a FF.

 The simplest of the storage components: $n$ inputs, $n$ outputs, and a clock signal.
All the $n$ FFs are driven by the common clock signal.

Registers are readily available as MSI circuits, it becomes convenient at times to employ a register as part of the sequential circuit. The combinational-circuit part of the sequential circuit can be implemented by any of the methods discussed in Chapters 4 & 5.

D-FFs are normally used for registers.

The register may be enhanced by asynchronous Preset and Clear (Reset) signals, which are not controlled by the clock signal.

Figure 1: A 4-bit register [Gajski].
Figure 2: A 4-bit register with asynchronous Preset and Clear [Gajski].
To be able to control when the data will be entered into a register, and for how long it will be stored there before being sent to the output, we add the Load (Enable) input to form a parallel-load register.

![Graphic symbol](a) Graphic symbol

<table>
<thead>
<tr>
<th>Present state</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td></td>
</tr>
<tr>
<td>Q₃ Q₂ Q₁ Q₀</td>
<td>I₃ I₂ I₁ I₀</td>
</tr>
<tr>
<td>0</td>
<td>No change</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

![Operation table](b) Operation table

![Register schematic](c) Register schematic

Figure 3: Register with parallel load [Gajski].
A shift register can shift the stored data right and/or left.

Figure 4: Shift registers [Gajski].
A counter is a special type of register that counts upward, downward, or in any prespecified sequence.
Figure 6: Binary up/down counter with parallel load [Gajski].
7. Storage Components

BCD Counter

*A BCD counter counts in the sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, ....*

![Diagram of BCD up-counter and up/down-counter](image)

(a) BCD up-counter  (b) BCD up/down-counter

Figure 7: BCD counters [Gajski].

Asynchronous Counter

*A asynchronous counter counts without an incrementer or decremener—its FFs are not clocked by the same signal.*

Counting without an incrementer/decrementer is achieved by **toggling** each FF at half the frequency of the preceding FF.

- FF$_i$ changes state only half as often as FF$_{i-1}$.
- FF$_i$ changes state only when FF$_{i-1}$ goes from 1 to 0, but not from 0 to 1.

A T-FF is very convenient for such an asynchronous counter design.

The **counting frequency** (speed) will be limited by the number of FFs due to the linear growth of the clock-to-output delay.

To speed up the counting process, we can use the **mixed-mode counter**.
Figure 8: Asynchronous up-counter [Gajski].
### Register Files

A register file has $2^n$ registers of $m$ FFs each.

- The registers are arranged as a 2-dimensional array of register-file cells (RFCs).
- In addition, it has read/write decoders and output driving logic.
- Writing is controlled by the Write-Enable (WE) signal.
  - At any time, we can write into only one register (row), unless it has multiple write ports.
- Reading is controlled by the Read-Enable (RE) signal.
  - At any time, we can read from only one register, unless it has multiple read ports.
- Reading from and writing into the same register at the same time normally is not allowed.

---

**Figure 9: Mixed-mode up-counter [Gajski].**
The primary advantage of a register file is regularity, which reduces routing (wiring) complexity.

Figure 10: Register file with 1 write port and 1 read port [Gajski].
Figure 11: Register file with 1 write port and 2 read port [Gajski].
Random Access Memories (RAMs)

☆ A RAM is organized as an array of \(2^n\) rows with \(m\) bits stored in each row.
  - The size of the RAM is \(2^n \times m\) bits—it has \(n\) address lines, \(m\) input data lines, and \(m\) output data lines (see Fig. 12).
  - The input data lines can be the same with the output data lines, i.e., the data lines can be bidirectional.
  - For a commodity RAM, \(16 \leq n \leq 30\), and \(m = 1, 4, 8, 16, 0r 32\).

☆ A memory cell (MC) can be considered as a clocked D latch with an AND gate and an output driver (see Fig. 13(a)).
  - For a static RAM (SRAM), MC is constructed by 6 transistors, using cross-coupled inverters to serve as a latch, and implementing the input AND gate and the output driver with one transistor each.
  - For a dynamic RAM (DRAM), MC is constructed by only 1 transistor.
    - The latch is implemented by a capacitor.
    - It needs to be refreshed periodically.
    - It has high density (therefore low cost).

☆ The RAM also has a Chip-Select (CS) input and a Read/Write Select (RWS) input (see Fig. 13(b)).
  - The RWS input sometimes is denoted as \(R' / W\).

☆ Both SRAM and DRAM are volatile memories, i.e., their content is lost if the power is shut down.
  - ROM, PROM, EPROM, EEPROM, and flash memories are nonvolatile.

☆ The delay time from address input to data output (\(t_2 - t_0\) in Fig. 14) is the memory access time.
  - The address/data setup time and hold time are shown in Fig. 14.

☆ We can connect several memory chips to get one of longer words (Fig. 15), or connect several memory chips to get one with more words (Fig. 16).
### Memory address

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ... 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 ... 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 ... 0 1 0</td>
<td>2</td>
</tr>
<tr>
<td>0 ... 0 1 1</td>
<td>3</td>
</tr>
<tr>
<td>0 ... 1 0 0</td>
<td>4</td>
</tr>
<tr>
<td>0 ... 1 0 1</td>
<td>5</td>
</tr>
<tr>
<td>0 ... 1 1 0</td>
<td>6</td>
</tr>
<tr>
<td>0 ... 1 1 1</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1 ... 1 1 0</td>
<td>$2^{n-2}$</td>
</tr>
<tr>
<td>1 ... 1 1 1</td>
<td>$2^{n-1}$</td>
</tr>
</tbody>
</table>

- $m$ bits

---

(a) Memory address and content

---

(b) Graphic symbols

---

Figure 12: Random-access memory (RAM) [Gajski].
Figure 13: RAM organization [Gajski].
Figure 14: RAM timing [Gajski].
Figure 15: A $16K \times 32$ RAM design using $16K \times 8$ RAMs [Gajski].
Figure 16: A $64K \times 8$ RAM design using $16K \times 8$ RAMs [Gajski].
*Push-Down Stacks*

A push-down stack (or simply stack) is a memory component with limited access—data can be accessed through only one location (i.e., the top of the stack).

- When data is to be stored, it is pushed on the stack and stays on top of others.
- When data is to be fetched, it has to be in the top position before it can be popped out of the stack.

A stack can be implemented by shift registers, with an up-down counter to detect full/empty stack as shown in Fig. 18.

It can also be implemented by a RAM—less expensive for a large stack, but need two pointers (implemented by counters) as shown in Fig. 19.

![Stack Diagram](Figure 17: Push-down stack operations [Gajski].)
Figure 18: A 4-word stack implemented by shift registers [Gajski].
Figure 19: A 4-word stack implemented by RAM [Gajski].
A first-in-first-out (FIFO) queue (or simply queue or FIFO) is a memory component with limited access—data can be written through only the head (front) of the queue and read (and removed) through only the tail (back) of the queue.

A queue can be implemented by shift registers, with an up-down counter to detect full/empty queue as shown in Fig. 21.

It can also be implemented by a RAM—less expensive for a large queue, but need two pointers (implemented by counters) as shown in Fig. 22.

Figure 20: FIFO queue operations [Gajski].
Figure 21: A 4-word queue implemented by shift registers [Gajski].
Figure 22: A 4-word queue implemented by RAM [Gajski].
Datapaths are used in all standard CPU and ASIC implementations to perform complex numerical computation or data manipulations; a datapath consists of temporary storage in addition to arithmetic, logic, and shift units.

**Example 1**
Assume we want to perform the summation of 100 numbers: \( \text{sum} = \sum_{i=1}^{100} x_i \). We can use the datapath as shown in Fig. 23 to implement the following algorithm:

```plaintext
sum=0;
for(i=1; i<=100; i++)
    sum=sum+x[i];
```

![Figure 23: Simple datapath with one accumulator [Gajski].](image)
Figure 24: Datapath with 3-port register file [Gajski].
### Figure 25: One’s-count algorithm [Gajski].

**Basic algorithm for one’s count**

1. Data := Inport
2. Ocount := 0
3. Mask := 1

**while** Data ≠ 0 **repeat**
4. Temp := Data AND Mask
5. Ocount := Ocount + Temp
6. Data := Data >> 1
**end while**
7. Outport := Ocount

**Register assignment**

<table>
<thead>
<tr>
<th>R1: Data</th>
<th>R2: Mask</th>
<th>R3: Ocount</th>
<th>R4: Temp</th>
</tr>
</thead>
</table>

**Control words for one’s counter**

<table>
<thead>
<tr>
<th>Control Words</th>
<th>Write address</th>
<th>Read address A</th>
<th>Read address B</th>
<th>ALU operation</th>
<th>Shifter operation</th>
<th>OE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>R1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>R3</td>
<td>0</td>
<td>0</td>
<td>add</td>
<td>pass</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>R2</td>
<td>0</td>
<td>X</td>
<td>increment</td>
<td>pass</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>R4</td>
<td>R1</td>
<td>R2</td>
<td>AND</td>
<td>pass</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>R3</td>
<td>R3</td>
<td>R4</td>
<td>add</td>
<td>pass</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>R1</td>
<td>R1</td>
<td>0</td>
<td>add</td>
<td>shift right</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>none</td>
<td>R3</td>
<td>0</td>
<td>add</td>
<td>pass</td>
</tr>
</tbody>
</table>