Chapter 3
Digital Logic Structures

Combinational vs. Sequential

**Combinational Circuit**
- does not store information, always gives the same output for a given set of inputs
- *example*: adder always generates sum and carry, regardless of previous inputs

**Sequential Circuit**
- stores information, output depends on stored info (state) plus input
- so a given input might produce different outputs, depending on the stored information
- useful for building “memory” elements and “state machines”
  - *example*: ticket counter

R-S Latch: Simple Storage Element
- R is used to “reset” or “clear” the element – set it to zero.
- S is used to “set” the element – set it to one.
- If both R and S are one, output could be either zero or one.
  - “quiescent” state -- holds its previous value
  - if a is 1, b is 0, and vice versa
Clearing the R-S latch

- Suppose we start with output = 1, then change R to zero.

Output changes to zero.

Then set R=1 to "store" value in quiescent state.

Setting the R-S Latch

- Suppose we start with output = 0, then change S to zero.

Output changes to one.

Then set S=1 to "store" value in quiescent state.

R-S Latch Summary

- R = S = 1
  - hold current value in latch
- S = 0, R=1
  - set value to 1
- R = 0, S = 1
  - set value to 0
- R = S = 0
  - both outputs equal one
  - final state determined by electrical properties of gates
  - Don’t do it

Gated D-Latch

- Two inputs: D (data) and WE (write enable)
  - when \( WE = 1 \), latch is set to value of D
    - \( S = \text{NOT}(D), R = D \)
  - when \( WE = 0 \), latch holds previous value
    - \( S = R = 1 \)
A register stores a multi-bit value.
- We use a collection of D-latches, all controlled by a common WE.

When \( WE = 1 \), \( n \)-bit value \( D \) is written to register.

Representing Multi-bit Values
- Number bits from right (0) to left (\( n-1 \))
  - just a convention -- could be left to right, but must be consistent
- Use brackets to denote range:
  \( D[l:r] \) denotes bit \( l \) to bit \( r \), from left to right

\[ A = 0101001101010101 \]

- May also see \( A<14:9> \), especially in hardware block diagrams.

Now that we know how to store bits, we can build a memory -- a logical \( k \times m \) array of stored bits.

Address Space:
- number of locations (usually a power of 2)

Addressability:
- number of bits per location (e.g., byte-addressable)
More Memory Details

- Not the way actual memory is implemented!
  - fewer transistors, denser, relies on electrical properties
- But the logical structure is very similar.
  - address decoder, word select line, word write enable
- Random Access Memory: 2 different types
  - Static RAM (SRAM)
    - fast, used for caches, maintains data when powered
  - Dynamic RAM (DRAM)
    - slower but denser, storage decays, must be refreshed
- Non-Volatile Memory: ROM, PROM, Flash

Memory Bandwidth

- Bandwidth is the rate at which memory can be read or written by the processor.
- Approximately equal to the memory bus size times the speed at which the memory is clocked.
- Examples of bandwidth (from Wikipedia):
  - Phone line, Modem, up to 5.6KB/s
  - Digital subscriber line, ADSL, up to 128KB/s
  - Wireless networking, 802.11g, up to 17.5MB/s
  - Peripheral connection, USB 2.0, 60MB/s
  - Digital video, HDMI, up to 1.275GB/s
  - Computer bus, PCI Express, up to 25.6GB/s
  - Memory chips, SDRAM, up to 52GB/s

Looking Ahead: C Arrays

- Similar to Java arrays
  ```c
  // integer array
  int iArray[3] = {1, 2, 3};
  printf("iArray[2]: %d", iArray[2]);

  // float array
  float fArray[2] = {0.1f, 0.2f};
  printf("fArray[1]: %f", fArray[1]);

  // character array
  char cArray[4] = {'a', 'b', 'c', 'd'};
  printf("cArray[3]: %c", cArray[3]);
  ```

Looking Ahead: C Strings

- Array of chars with null termination
  ```c
  // string: static allocation
  char *string1 = "Hello World\n";
  printf("string1: %s", string1);

  // string: dynamic allocation
  char *string2 = (char *)malloc(13);
  strcpy(string2, "Hello World\n");
  Note that the programmer is responsible for making sure string has enough memory!
  ```
Looking Ahead: C Arrays and C Pointers

- Array name is a pointer to array

```c
int iArray[2] = {1234, 5678};
printf("iArray[0]: %d", iArray[0]);
printf("iArray[1]: %d", iArray[1]);
printf("&iArray[0]: %x", &iArray[0]);
printf("&iArray[1]: %x", &iArray[1]);
printf("iArray: %x", iArray);
iArray[2] = 0; // out of bounds!
```

Looking Ahead: C Functions

- Can pass by value or reference

```c
// by value (copies value)
float f1(int i, float f);
// by reference (copies pointer)
float f2(float *f);
```

- Function cannot change values passed by value

```c
f1: i = 10; // changes the copy
```

- Function can change values passed by reference

```c
f2: *f = 1.2; // changes actual value
```