CMPT 250 : Week 1

1. OVERVIEW

This course is about the organization and architecture of computer systems. The term *computer architecture* refers collectively to:

- **instruction set architecture**: the computer hardware systems from the point of view of the programmer. That is, the functional behavior, representation of data, and operations supported.
- **organization**; The structural relationships among the hardware components, including such considerations as the clock frequency, the interface organization and the memory technology employed.

The course will focus on the system bus model of the von Neumann architecture. This model is based on the organization of five comonents (datapath or ALU, controller, input device, output device, and memory unit) all sharing a common bus, referred to as the *system bus*.

The key feature of the von Neumann model, and consequently its wide usage, is the storage of program instructions in memory along with the data. This permits instructions to be manipulated in the same way as data.

The use of a system bus to provide the communications link between the five modules reflects a common practice in interface design; that is, the adoption of a standard interface between components. This system bus provides signal lines to address the three main requirements:

- provision of a path for transferring data between components (i.e., a data bus)
- provision of a path for specifying source and destination addresses when storing, retrieving, reading, or writing data (i.e., an address bus)
- provision of a path for sending and receiving control signals (i.e., a control bus).

2. FORMAL SPECIFICATION OF DIGITAL SYSTEMS

Before examining the development of computer architectures, certain tools and techniques are required:

- Techniques for formally specifying and digital system are needed. The goal is a behavioral description that is unambiguous, consistent, and complete.

- An inventory of "standard" components, that by convention are assumed to be available as building blocks. Specification of the behavioural descriptions of all such components is required.
- Design strategies for transforming behavioural descriptions into structural ones; that is, a network of components.

BASIC DESIGN STRATEGY

1. Analyze the problem to identify requirements. These requirements may include any of the following:
    - inputs and outputs
    - performance constraints
    - economic considerations
    - component restrictions
    - interface constraints
    - classification (Combinational vs Sequential)
2. Recast the problem, using a formal specification model to produce a *behavioral description* of the system.
3. Apply several alternative design strategies in parallel, discarding those which cannot continue to meet the engineering constraints.
4. Formulate a *structural description* of the system, such as a logic diagram, which represents the problem as a decomposition of the problem into commonly accepted "standard" components.
5. When all components are resolved to commercial packages, circuit can be constructed.

2.1. CLASSIFICATION OF DIGITAL SYSTEMS

Combinational:

- Outputs depend only on the inputs currently being applied.
- An output can change only if an input changes.

Sequential:

- The outputs depend on both the current inputs and the content of all memory elements, collectively called the *state* of the system.
- The state of the system may also change as a result of a change in the current inputs.

- There are two types of sequential circuit:
    - **Asynchronous**: Any change to an input at any time can affect the outputs or content of memory.
    - **Synchronous**: Changes to the inputs affect the system only if they are held long enbough to be present during discrete time intervals when the system is "enabled".

2.2. FORMAL SPECIFICATION OF COMBINATIONAL SYSTEMS

A *combinational system* is one whose output at any instant depends only on the current value of the inputs.

A combinational system is defined by:

1. An Input Set (I) : The set of values that can be input.
2. An Output Set (O) : A set of values that can occur as outputs.
3. An Input/Output Function (F : I --> O) : The set of transformations performed by the system. The function F can be defined by a function table, algebraic expression, algortihm, or other notation that completely and unambiquiously captures the desired behavior (hence the term "behavioral description")

2.3. BEHAVIORAL DESCRIPTIONS

A *behavioral description* is a formal representation of a digital system that defines the inputs, outputs, and describes the functions performed by the system. It does not provide any information on the internal structure of the system.

A behavioral description is commonly provided by a "black-box" diagram with inputs and outputs labelled so that it can be used as a component in a logic diagaram. The "behavior" of the device so represented can be defined in a variety of ways:

- By an algebraic expression.
- By a function table itemizing the observed outputs for every possible input.
- By a function select table. This is a table where only the values of the control variables identify each row of the table, and the output is defined by an algebraic expression in terms of the data inputs, rather than by an explicit value.
- By an algorithm that describes a computational procedure for

obtaining the outputs. This algorithm may be expressed as an algorithmic state machine diagram (ASM), or in a hardware description language (eg. VHDL).

The choice of representation will often be determined by the information provided in the informal description of the problem. Some examples:

Example 1: "Design a system to sum three numbers from the set {0, 1}."

The entity definition for the behavioral description consists of a black-box with three inputs and 1 output. In this case, the output set is determined to be {0, 1, 2, 3}, and as both sets are finite and the number of inputs small, the function can be defined in tabular form, usually called a *function table*:

x	y	z	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	1
1	0	1	2
1	1	0	2
1	1	1	3

Example 2: The previous example is more commmonly referred to as the specification of a 1-bit full adder. If the outputs are encoded into their corresponding "natural" binary sequences:

```
0 = 00, 1 = 01, 2 = 10, 3 = 11
```

then the specification consists of two truth tables:

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Example 3: The boolean functions defined by these two truth tables are:

$$s = x \oplus y \oplus z$$

$$c = xy + z(x \oplus y)$$

Consequently, the black-box diagram together with this set of boolean equations constitutes a complete behavioral description of the 1-bit full adder.

Often it is possible to obtain an "algebraic" behavioral description for systems too complex to define by a function table. As well, algebraic descriptions permit the application of mathematical tools to the transformation of such expressions to obtain different implementations for a given problem.

3. VHDL: A NOTATION SYSTEM FOR HARDWARE MODELS

VHDL is a language for describing electronic systems, either in terms of their behavior (hence called behavioral models) or in terms of their components and their interconnections (and so called structural models).

VHDL is the abbreviation for *VHSIC Hardware Description Language*, where VHSIC stands for "Very High Speed Integrated Circuits." Some of the features of VHDL include:

- The ability to describe the structure of a system in terms of its components;
- The ability to describe the function of a systm using an algorithmic notation, similar to procedural programming languages;
- The ability to simulate systems so described before fabrication, thus reducing the time and costs of development;
- The ability to construct designs that can be simulated, from more abstract specifications, thus permitting earlier strategic desgin decisions to be made.

The following notes provide a *very tiny subset* of VHDL. Users should refer to cited sources at the course website for more information.

3.1. ENTITIES

An *Entity* is equivalent to a black-box, and provides the same kind of information; namely the nature of the inputs and outputs of the hardware device. An example of an entity representing the 1-bit full adder is:

```
entity FA is
   port(x, y, z: in std_logic;
        s, c: out std_logic);
end FA;
```

`std_logic` is a VHDL data-type that includes the bit values 0 and 1.

The entity only defines the black-box part of the specification; a functional definition is also required. This can be done in VHDL within an *architecture* body:

```
architecture behav of FA is
begin
   proc1: process
   begin
        s <= x xor y xor z;
        c <= (x and y) or (z and (x xor y));
        wait on x, y, z;
   end process;
end behav;
```

This archtecture is defined in a single process called `proc1`. In general an architecture can include many processes. Processes provide a way of describing potentially parallel events in VHDL. Whether a process actually gets invoked during simulation depends on the list of *sensitivity* variables that are part of the first statement of a process declaration. In the above example, the process will be invoked if `x`, `y`, or `c_in` change their value. This is consistent with the idea that the outputs from a combinational system can only change if some input changes.

This architecture uses the pre-defined functions `and`, `or`, and `xor` to define the output values. The symbol `<=` defines a signal assignment statement, since a value is being assigned to the output signal lines `s` and `c`.

VHDL descriptions include two kinds of variables: *signal* variables, and *ordinary* variables. Ordinary variables can be introduced into descriptions and used for the same purposes as found in ordinary programming languages.

Signal variables, however, are unique to hardware description languages and represent the inputs or outputs of a hardware component.

The example above could be defined as follows. This version makes use of a temporary ordinary variable. Note that all the input and output variables in an entity definition are signal variables.

```
architecture behav of FA is
begin
   proc1: process
   variable tmp : std_logic;
   begin
        tmp := x xor y;
        s <= tmp xor z;
        c <= (x and y) or (z and tmp);
        wait on x, y, z;
   end process;
end behav;
```

Ordinary variables can be declared (with a `variable` statement) either locally with respect to a given process, or globally within a given architecture definition. NOtice the different assignment operator (":=") used for assigning values to ordinary variables.

Another distinction between VHDL descriptions and ordinary programs is the ability to specify a propagation delay that occurs before a given signal variable is assigned a new value. Using the same example, one can specify a propagation delay of 10 ns for `s` and 15 ns for `c` as follows:

```
architecture behav of FA is
begin
   proc1: process
   variable tmp : std_logic;
   begin
        tmp := x xor y;
        s <= tmp xor z after 10 ns;
        c <= (x and y) or (z and tmp) after 15 ns;
        wait on x, y, z;
   end process;
end behav;
```

To understand how the above description is interpreted by a simulator it is important to understand how the simulator "executes" a VHDL description:

1. All VHDL descriptions are executed under the control of an event clock. The event clock is modelled by a queue of events with the events order according to when they are to take place.

2. Initially, the event queue is empty and all variables are set to 0.

3. The simulator evaluates all processes in every architecture until a wait statement is encountered.

4. Whenever signal assignments occur, these cause a new event to be added to the event queue. The signal variable is not updated until the event reaches the front of the queue and is removed by the simulator.

5. Whenever the simulator removes an event from the event queue, it searches for every occurrence of the associated signal variable in some wait statement list. Every process that includes such a wait statement is then re-evaluated again, until the next wait statement in the process is encountered.

6. The simulator continues to remove events from the queue, while processes may continue to add events to the queue until such time as the simulation is terminated or because there are no more events left to process on the queue.