1. Digital System Design

A digital system performs microoperations. The most well known digital system example is a microprocessor. A microprocessor performs microoperations to run machine language instructions. Microoperations are simple operations that can be completed in one clock period, such as additions, subtractions, AND, OR, INVERT, compare, shift, rotate, read from memory, write to memory. Digital systems consist of digital circuits. That is, digital circuits are building blocks of digital systems. Examples of digital circuits include multiplexers, decoders, adders, comparators, encoders, demultiplexers, registers, counters and shift registers. All of these digital circuits are implemented by gates and flip-flops. CS 2204 covers gates, flip-flops and digital circuits as fundamental circuits and then discusses digital system design in the context of a game chip. Designing microprocessors as digital systems is a main topic of the CS 2214 Computer Architecture and Organization course.

Our earliest experimentation with digital systems started in the 18th century when first steam-operated mechanical textile machines were manufactured and continued with the electromechanical telephone exchange systems of the early 20th century. Today's digital systems are more complex than ever. For example, a microprocessor today has billions of transistors. But, there are still no formal techniques to design digital systems today. That is, there is no specific algebra or formal circuit synthesis rules. Therefore, digital system design is based on the divide-and-conquer method where a digital system is repeatedly partitioned into smaller and smaller pieces until each is reduced to a small digital circuit upon which Switching Algebra and sequential circuit synthesis rules are applied.

For today’s high-density chips, such as microprocessors, help is needed and are available for the complex digital system design process. Methods and tools have been developed, accomplishing two seemingly contradictory jobs: hiding details and simultaneously allowing the design to freely move from one level to another (simultaneous design of the levels) down to the transistor level. One such digital system design technique that borrows ideas from sequential circuit synthesis rules, is the finite-state machine (FSM) technique. A state diagram with finite number of states is drawn to show which microoperation happens when and how, thus describing both the datapath and the control unit. In this technique, a FSM state diagram describes a digital system.

Another design technique, currently used by industry is Hardware Description Language- (HDL-) based. It is needed when a complex circuit is designed. HDL statements have constructs to describe hardware events, resulting in the abstraction of hardware. Such statements are as understandable as high-level language statements. In this technique, a HDL program describes a digital system. Another HDL program describes another digital system. Most commonly used HDLs are VHDL and the Verilog HDL. Designers work on these HDL programs and delay the complex task of building the prototype until they are sure about their HDL program. Polytechnic has a number of courses that use the HDL approach, for example EE 4313, EL 5473, EL 5493 and EL 6443.

FSM and HDL approaches lend themselves to computer usage for speedy correct design. For example, verifying that the design works is done by obtaining correct results from the simulations of the FSM or HDL design on a computer. Such simulations are fast, allowing designers to start the prototype building phase quickly. For complex circuit design, HDL languages together with software tools are powerful enough to describe a digital system at the microarchitecture, logic and transistor layers, alleviating the lack of formalism in the digital system design to a certain extent.

In CS 2204, the finite state machine design technique will be used. We will concentrate mostly on the Ppm project and obtain a diagram only for the Ppm. A diagram will describe the Ppm digital system, specifying which microoperation happens when and how. CS 2214 also uses the finite state machine design technique to design a microprocessor. Below, first, a brief summary of the FSM technique and datapath construction are introduced. Then a vending machine controller design is given as a complete example for finite state machine design.
1.1. High-Level Digital System Design

Digital System design process includes a series of partitionings today. However, there are no specific rules about how to partition a digital system today, except that the first partitioning of every digital system is into data and control units. This partitioning is universal. The data unit, also known as datapath, performs the microoperations. The control unit determines the timing and sequence of the microoperations (the complex task of determining which microoperation happens when). Thus, the control unit controls the data unit. Then, one can partition in a number of different ways, each i) simplifying or complicating the design process and each, ii) decreasing or increasing the cost, speed, size, power consumption, reliability, etc. Thus, in digital system design, there are plenty of good designs, each satisfying a different set of speed/cost/size/power consumption/reliability/... goals.

The data unit performs microoperations such as additions, subtractions, AND, OR, shift, compare, fetch an instruction, read data from memory, write data to the memory. In order to perform these microoperations, it needs to have three kinds of hardware: registers, ALUs and buses as shown below. Registers are needed to keep data temporarily, ALUs perform additions, subtractions, shifts, ANDs, ORs, etc. and buses interconnect registers and ALUs. Registers contain flip-flops to store bits. ALUs are often combinational circuits, i.e. they contain only gates. Buses are bundles of wires with additional control logic. The control unit consists of a sequencer that generates control signals based on the status signals from the datapath. It is the sequencer that determines the sequence of microoperations.

While the control unit seems to perform rather a simple job of controlling the data unit and the data unit has most of the logic, the datapath is easier to design than the control unit because of two reasons. One is that the components are not ideal. They have gate delays and fan-in restrictions and consume power. Often, the control unit circuit logically works but when it becomes operational, it does not work due to glitches created by the gate delays. The second reason is that the data unit is highly regular. That is, a data unit has pieces of hardware repeated many times. For example, an ADDer is a repetitive set of smaller addition blocks, the multiplier is similar and so are registers. Another example is that a 4-bit compare circuit can be repeated 8 times to compare 32-bit numbers. The design, test, modify, manufacture, upgrade of regular hardware is easier. It also costs less. But, the control unit is not that regular. The sequencer is implemented by either hardwiring or microprogramming or a mixture of both. Especially, hardwired control units have considerable random logic that their design is difficult.

The FSM techniques requires that we obtain a state diagram with a finite number of states. Each state specifies zero, one or more microoperations and how what they require in hardware. Thus, the FSM state diagram describes the digital system. However, the FSM technique is preferable if the number of states is small. That is, it is practical when the digital system is relatively simple. Otherwise, one would have to use the HDL technique. In CS2204, we will use the FSM technique since the Ppm digital system we implement is a simple digital system. However, the FSM diagram is such that it is easy to modify it to add more features.

In the FSM technique, the state diagram has rather a “circular” look where the same states are traced one after another as shown below. Sometimes, a few states can be different based on the conditions at the moment. One can interpret the FSM state diagram as a “graphical program” where each circle is a “program line” and the branches coming out of a state represent an “If statement.” Since the state diagram is circular, one can think of the digital system in an “infinite loop” performing same microoperations again and again. In the case of a Ppm digital system, the state diagram is traced top to bottom (except the Reset state) as the players play or skip one after another. If the players play/skip 20 times until the game is over, the Ppm will trace its state diagram top to bottom, except Reset, 20 times.
1.2. The High-Level State Diagram

The first step to design a digital system by using the FSM technique is obtaining high-level state diagram. The high-level state diagram specifies which microoperation happens when. That is it describes the Data Unit. Below, a partial picture of the high-level state diagram of an imaginary digital system is shown. Again, each circle in a high-level state diagram represents a state and each state corresponds to a clock period. In order to identify states, they are numbered as shown above: state 24, state 25, etc. There is no relationship between a state number and the clock period the state occurs. For example, state 24 above, might or might not occur in clock period 24. Only one (1) state occurs in one clock period. Assume that for the remaining portion of the handout, state 24 occurs in clock period 46. In clock period 47, state 25 will occur. There may be multiple branches coming out of a state, as in state 25 above. Only one of the states will happen in the next clock period. State 25 occurs in clock period 47, then, in clock period 48, either state 26 or state 27 will occur, but not both.

Each state has zero, one or more microoperations. State 24 has one microoperation, so does state 26:

- \( B \leftarrow PC + D \)  ; A register add microoperation in state 24
- \( A \leftarrow C \)  ; A register transfer microoperation in state 26

Often a microoperation moves a value from a register to another register as in state 26. This is called a register transfer. A typical state diagram may have many register transfers. That is why we sometimes name the microarchitecture level, the Register Transfer Level (RTL). It is the same reason why the microoperation notation used in the above figure (\( PC \leftarrow A \)) is called the Register Transfer Language.

If a state has more than one microoperation, they happen in parallel. Thus, the order of writing these microoperations in a state does not matter. For example, state 25 has two microoperations that happen in parallel:

- \( MDR \leftarrow M[B] \)  ; A memory read microoperation in state 25
- \( PC \leftarrow A \)  ; A register transfer microoperation also in state 25

Each state takes at least one clock period, unless there is an arrow pointed at itself, then it can take more than one clock period.

What the digital system does is continuously going through a sequence of states (tracing circles) based on conditions at the moment.

The state diagram has a circular look.

Each circle is a state.
The process of determining which branch to take is testing. In the above picture, the rightmost bit of register MDR is tested to take a branch: MDR[0]. This test is performed in state 25, in parallel with the memory read. The testing is done by the control unit, while the datapath transfers the value from the memory to the MDR. Thus, the control unit checks the current value of MDR, not the value read from the memory at the same moment. The testing determines which one of the microoperations will happen in the next clock period (clock period 48). In order to test MDR[0], this leftmost bit is input from the datapath to the control unit as a status signal. If an instruction is tested in a clock period, we call it instruction decoding. Above, we might or might not be decoding an instruction.

1.3. The Datapath Design

As stated earlier, the datapath performs microoperations for which it has to have registers, buses and ALUs. The transfers from a source to a destination in the state diagram are typically carried out by buses in a digital system. Otherwise, there must be direct connections between every source and destination pair which is very expensive. This is the reason for buses. Typically, a datapath contains at least three internal buses to interconnect the registers and ALUs. The figure below has three internal buses: ABUS, BBUS and OBUS. The Memory Data Bus (MDB) and the Memory Address Bus (MABUS) connected to the memory are external buses.

A bus is a shared set of lines. A bus has two or more sources and at least one destination. ABUS above has three sources (PC, A and C) and two destinations (an ALU input and OBUS). A bus is constructed by connecting the sources to a multiplexer (MUX) whose output is the bus. Select signals of the MUX connect a source to the bus. SelectABUS lines connect source PC to ABUS if they are 00. They connect source A to ABUS, if they are 01. If they are 10, they connect C to ABUS. The SelectABUS signals are control signals and generated by the control unit. If a
MUX has more than two sets of data inputs, it will have more than one select line. All the buses above have more than two sets of data lines input. For example, OBUS has eight sets of data inputs and so has three select lines. Note that, we draw control signals as angled lines to make sure we know they are generated by the control unit.

A bus gets a value as soon as it is “connected” a value, which happens in the beginning of the clock period the bus is used. For example, the ABUS is connected PC in state 24. ABUS gets the value of PC in the beginning of clock period 46 as shown below. It is important to note that a bus does not remember what was connected to it in the past. For example, in clock period 47, the ABUS does not remember the value of PC connected to it in clock period 46.

The reason why we use a multiplexer to form a bus is that we use regular gates and FFs (registers are totem-pole registers) as opposed to tri-state or wired-OR gates and FFs. It is important NOT to short-circuit the outputs of totem-pole gates and FFs as done below. A MUX solves the problem.

A datapath has registers to keep (remember) values for the future. Storing a value on a register is very different from connecting a value to a bus. A register which is transferred a value in clock period “x” will actually get the new value in the beginning of the following clock period: “x + 1.” For example, register B in state 24 (clock period 46) is transferred a new value which is (PC + D). The B register takes on the new value in the beginning of clock period 47 (in state 25), not in clock period 46 (not in state 24).
Similarly, register MDR and PC take on their new values in the beginning of clock period 48 (one of states 26 or 27), not in clock period 47 (not in state 25). Thus, the MDR[0] bit that is tested in state 25 is not the new bit, but the old one. The reason why a register takes on (is stored) a new value in the beginning of the next clock period is that flip-flops of registers are clocked at the end of the clock period.

The reason why flip-flops (FFs) are clocked at the end of a particular clock period is that they are **edge-triggered** FFs. In CS2204, we will frequently use negative-edge triggered FFs: they need negative edges to store values.

When we want to store a value on a register in a state, say, state 25 which occurs in clock period 47, a negative edge is supplied to the clock (C) input and a 1 is supplied to the Clock Enable (CE) input of the register in state 25 as shown on the left.

The StoreMDR signal is active, 1, when we want to store on MDR, that is, in state 25. The AND of the clock signal and the StoreMDR signal is StoreMDRpulse that results in a **negative edge**. The negative edge happens at the end of the 47th clock period, storing the Memory Read Bus (MRBUS) value in the beginning of the 48th clock period.

Note that the MDR register is in the datapath. The data input to MDR is from the memory: MDB, carries the content of a memory location. The StoreMDR signal is generated by the control unit of the digital system and is supplied to the datapath.

The memory operation happens as follows: In the beginning of the 47th clock period, the system activates its read signal to the memory called MemRead. The memory uses the Memory Address Bus (MABUS) as the address to access the location. Before the end of the clock period the memory delivers the content of the location which is stored at the end of clock period 47. Note that the value of MRBUS in the beginning of clock period 47 changes (is not stable) as the memory is in the process of accessing the location. Eventually, the location is accessed and the value becomes stable.

Finally, we will assume that we can read or write a memory location in one clock period as seen on the left. Similarly, we will assume we have hardware fast enough to add two registers in one clock period as implied by
The finite state diagram above is called the **high-level state diagram**. In the diagram, the data movements are not written in terms of buses. Because when we start the design, we typically obtain a state diagram without buses. We just show register transfers. Then, we decide about the busing structure, such as how many buses and which register is a source on which bus. Below, we convert the high-level state diagram to the one with buses.

Note that we use an **arrow** if the destination is a register and an **equal** sign if the destination is a bus or wire. As stated above, all microoperations in a state happen at the same time. This means, all the connections and transfers happen simultaneously in a state, not sequentially. So, we can write the connections and transfers in a state in any order, it does not matter.

How many buses and what is connected to them are based on the speed, cost, power, size, etc. goals. For example, the more buses, the higher the speed and the cost. This is because, with more buses, we can have more data transfers in parallel, not in sequence, saving time. But, with more buses, there are more wires and control logic and so the cost is high. In addition, the placement of registers on buses is not straightforward.

For example, in state 24 we add PC and D which means PC and D cannot be on the same bus to do the addition. They must be connected to different buses. In state 27, we add C and D, meaning that they must be on separate buses to do the addition.

### 1.4. Low-Level State Diagram

Next step is obtaining a state diagram where the microoperations are described in terms of control signals. That state diagram is the **low-level state diagram**. Below, we show the low-level state diagram of the above system. These control signals are determined after the datapath design is complete and when the control unit is designed.
It is the control unit that has to generate these control signals and so its design depends on this control signal determination very much. However, the control unit has to receive status signals from the datapath as well. The status signals are also determined during the control unit design.

1.5. The Control Unit Design

The control unit controls the data unit. It determines which microoperation happens when. It also determines which datapath components are involved for that microoperation. The control unit has status signals as inputs and control signals as outputs. The circuit in the control unit is called \textit{sequencer.} It is the one that determines the sequence of microoperations, hence the name, sequencer. The sequencer goes through steps, known as \textit{states.} The sequencer may “jump over” steps also. But, it knows exactly which step it is in by using a \textit{state register.} That is, the state register indicates the state the digital system is in at any moment. For example, if the state is 24 at the moment, then the value of the state register is 24.

Based on the state and status signals, control signals are generated for the data unit and the next (state) value for the state register. The sequencer circuit is often an “\textit{irregular}” circuit whose design is quite complex. The irregularity is due to the large amount of \textit{random logic} (gate networks and flip-flops).

The sequencer is implemented by using \textit{hardwiring} or \textit{microprogramming}. Hardwiring generates control signals by \textit{networks of gates and flip-flops}: for each control signal, there is a network of gates and flip-flops. Networks are needed also to generate next state signals. Overall, hardwiring leads to a \textit{tremendous} number of gates. Since networks of gates and flip-flops are \textit{random logic}, their development is very time consuming. The design, test, modify, manufacture and upgrade of huge amounts of random logic is monumentally difficult:
The advantage of hardwiring is that it is faster than microprogramming. If one wants the fastest possible sequencer for a digital system, hardwiring is used. Hardwiring is also more expensive. In the early days of computing, only hardwiring was known and used until microprogramming became practical in the 1960s. In the 1970s, hardwiring became more of a choice of supercomputing, as machine language instruction sets became complex. Only microprogramming was able to handle the complexity for other types of computers.

A microprogrammed sequencer generates control signals by using a special memory, a control memory, in the control unit, with little accompanying random logic as shown below. Microprogramming simplifies the control unit development. The control memory is stored bits representing values of the control signals for each state. There are also bits indicating what the next state is. The content of the control memory is referred to as control program or microprogram. Each location of the control memory keeps a microinstruction.

The control memory can be read-only, meaning the microprogram is never changed. If the control memory is a read-write memory, it can be changed, even at run time! As mentioned above, hardwiring is faster than microprogramming, but more expensive!

We conclude the Digital System Design section by listing three seemingly unimportant points about it. When we deal with digital systems, we need to remember that

i) We cannot short circuit outputs of gates and flip-flops unless they are tri-state.

ii) Registers take on new values in the clock period after their write control signal is turned on

iii) Buses take on new values in the clock period their connection signal is turned on

2. An 8-bit Multiplier

We design a digital system that calculates an 8-bit multiplication by means of successive additions. We will design it by using the finite state machine (FSM) technique. The design is done through seven steps as discussed in class and in the lab.

1) The black-box view and the textual input/output relationship
The Multiplier is input two 8-bit Unsigned Binary numbers, K and M. The result is a 16-bit Unsigned Binary result. We use 16 bits in order to avoid frequent overflows. The black-box view and input/output relationship are as follows
2) Obtain the operation diagram from the black-box view and the textual input/output relationship

![Operation Diagram]

- **K and M** are 8-bit unsigned binary numbers
- **R** is 16-bit unsigned binary number

**Textual Input/Output Relationship**

When **Start** becomes 1, store **K** and **M**. Obtain the multiplication by adding **K** by **M** times. Then, output the result on **R** and raise **Valid** to 1.

3) Obtain the high-level state diagram from the operation diagram

![State Diagram]

One would continue with the design of the multiplier, by completing the remaining steps. That is, the Datapath design, the low-level state diagram, deciding about hardwiring and the control unit are done.

Students are asked to do the remaining steps by themselves. Below, we will give the complete design of another digital system, a vending machine controller.
3. A Vending Machine Controller

We will design a vending machine controller as a digital system by using the finite state machine (FSM) technique. The design is done through seven steps as discussed in class and in the lab.

1) **The black-box view and the textual input/output relationship**
   A vending machine, by means of its controller, delivers gums and chips, both costing 35 cents. One can input only Dimes (10 cents) and Nickels (5 cents). There is **no** Coin Return button.

   ![Diagram](image)

   **Inputs**
   - D : Dime is input
   - N : Nickel is input
   - G : Gum is selected
   - C : Chips is selected

   **Outputs**
   - **Amount shown on two 7-segment display** :
     1) Value of gum and chips (35 cents) or
     2) The coin input so far
   - DG : Deliver Gum
   - DC : Deliver chips
   - RetCoin : Return 5 cents

   **Textual Input/Output Relationship**
   After receiving the necessary amount (35 cents or 40 cents) and the selection is made, deliver gum or chips and if necessary return 5 cents

2) **Obtain the operation diagram from the black-box view and the textual input/output relationship**

   ![Operation Diagram](image)
3) Obtain the high-level state diagram from the operation diagram

4) Obtain the datapath from the high-level state diagram

These three signals are control signals generated by the Control Unit

These two signals are status signals used by the Control Unit
5) Obtain the low-level state diagram from the high-level state diagram and datapath

6) Decide about hardwiring versus microprogramming

The control unit is one of the two major parts of a digital system:

Other digital systems/Input/Output devices

<table>
<thead>
<tr>
<th>Registers</th>
<th>ALUs</th>
<th>Buses</th>
</tr>
</thead>
<tbody>
<tr>
<td>status signals</td>
<td></td>
<td>control signals</td>
</tr>
<tr>
<td>Sequencer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We will decide whether we will use hardwiring or microprogramming to implement the control unit!

A Load signal needs to be generated to load the state register when needed, i.e. in a specific clock period.
We observe that the control unit has six status signals and eight control signals:

Since the vending machine controller is a simple digital system, we decide to use hardwiring.

7) Implement the control unit based on the low-level state diagram and using hardwiring

a) We separate the decoder from the other logic in the cloud:
b) We partition the cloud of gate and FF networks to two parts:

![Diagram of state machine]

- Status signals
- Control signals
- Current State
- Next State & Load
- 3-bit Register
- Decoder
- 3-to-8 Decoder

A Load input is needed to load a value to the state register when needed, i.e. in a specific clock period.

c) We have to decide if we want to use a counter or a register to keep track of the state. The decision will depend on the state transitions. We observe that we do not trace the states sequentially as 0, 1, 2, 3,.... We trace them by skipping states often. Therefore, it will not help us if we use a counter. We will use a register. It will be a 3-bit register since we have seven states. Since it is a 3-bit register, the decoder is a 3-to-8 decoder:

![Diagram of decoder]

- Y7
- Y6 -> S6
- Y5
- Y4 -> S5
- Y3
- Y2 -> S4
- Y1
- Y0 -> S3

A Load input is needed to load a value to the state register when needed, i.e. in a specific clock period.

d) We will generate control signals for which we will use gate networks. In order to get the gate networks, we have to obtain expressions:

![Diagram of gate and FF networks]
Control Signal Generation

Store is 1 when it is state 1 or 3

\[
\text{Store} = S_1 + S_3
\]

Dime is 1 when it is state 1

\[
\text{Dime} = S_1
\]

Clr is 1 when it is state 0

\[
\text{Clr} = S_0
\]

Coin is 1 when it is state 1 or 2 or 3 or 4

\[
\text{Coin} = S_1 + S_2 + S_3 + S_4 + S_5 + S_6 = \overline{S_0}
\]

5centRet is 1 when it is state 5 and Gt is 1 or state 6 and GT is 1

\[
\text{5centRet} = (S_5 + S_6) \cdot \overline{GT}
\]

OpenG is 1 when it is state 5

\[
\text{OpenG} = S_5
\]

OpenC is 1 when it is state 6

\[
\text{OpenC} = S_6
\]

Enable is 1 when it is state 1 or 3, same as Store!

\[
\text{Enable} = \text{Store}
\]

Finally, we will generate the next state signals for which we will use gate networks. In order to get the gate networks, we have to obtain expressions:
Next State Generation

a is 1 when it is state 1 and LT is 0 or state 3 and LT is 0 or state 4

\[ a = (S1 + S3)\overline{LT} + S4 \]

b is 1 when it is state 0 and N is 1 or state 1 and LT is 1 or state 2 and N is 1 or state 3 and LT is 1 or state 4 and C is 1

\[ b = S0N + S1LT + S2N + S3LT + S4C \]

c is 1 when it is state 0 and D is 1 or state 0 and N is 1 or state 2 and D is 1 or state 2 and N is 1 or state 4 and G is 1

\[ c = S0D + S0N + S2D + S2N + S4G \]

Load is 1 when it is state 0 and D is 1 or State 0 and N is 1 or state 1 or state 3 or state 2 and D is 1 or state 2 and N is 1 or state 4 and G is 1 or state 4 and C is 1 or state 5 or state 6

\[ \text{Load} = S0D + S0N + S1 + S3 + S2D + S2N + S4G + S4C + S5 + S6 \]
The full design is as follows:

Datapath

Control Unit

These three signals are control signals generated by the Control Unit.

These two signals are status signals used by the Control Unit.