Basics of Digital Logic Design

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Sketch

- Combinational logic
- Field programmable gate arrays
- Sequential logic
- Asynchronous inputs
- Static random access memory (SRAM)
- Dynamic random access memory (DRAM)
- Finite state machines

Combinational logic

- No memory for remembering state
- Usually always enabled
- Some important combinational logic units
  - Decoders
  - Multiplexors
  - Encoders
  - Priority encoders
  - Comparators
- Any combinational logic can be designed using AND, OR, and NOT gates
  - CMOS: NAND, NOR, and NOT gates

Combinational logic

- Two-level logic implementation
  - Any combinational logic can be designed by having an array of AND gates in first level, an array of OR gates in the second level, and an optional NOT to get the output
    - Sum of product (SoP) representation; sum of minterms
  - First level OR and next level AND
    - Product of sum (PoS) representation; product of maxterms
  - SoP and PoS representations follow directly from the truth table
  - Example: full adder

Combinational logic

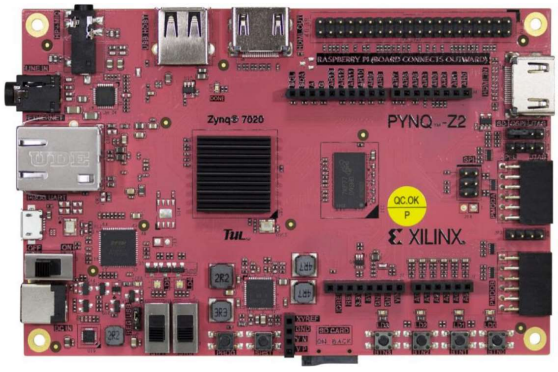
- Programmable logic array (PLA)
  - Another name for SoP implementation
  - AND plane and OR plane
  - Size of the planes?
  - Dot representation
- Implementing Boolean functions in ROM
  - Read only memory
    - PROM and EEPROM variants
  - Number of rows is exponential in number of inputs
  - Number of columns is equal to number of functions

Combinational logic

- Comparison between ROM and PLA for implementing combinational logic
  - ROM requires a full decoder (array of AND gates)
  - The AND plane size of PLA depends on the number of distinct minterms across all outputs
  - As the number of inputs grows, ROM height grows exponentially; PLA size grows slowly
- Logic minimization
  - Critical for good designs
  - Karnaugh map (K map) is useful for understanding the manual process
  - Tedious for large designs; usually automated

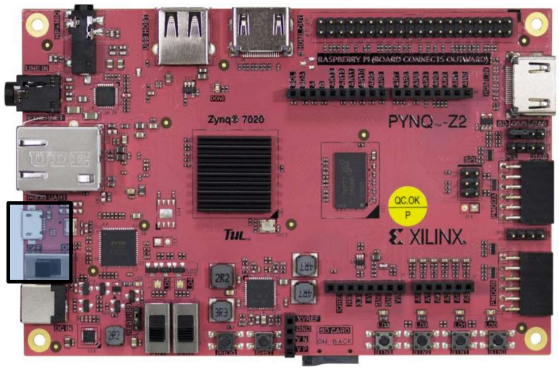
PYNQ-Z2 FPGA Board

- Contains several parts



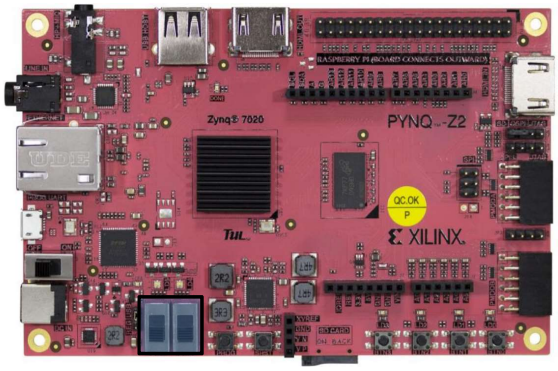
PYNQ-Z2 FPGA Board

- Power connection port and power switch



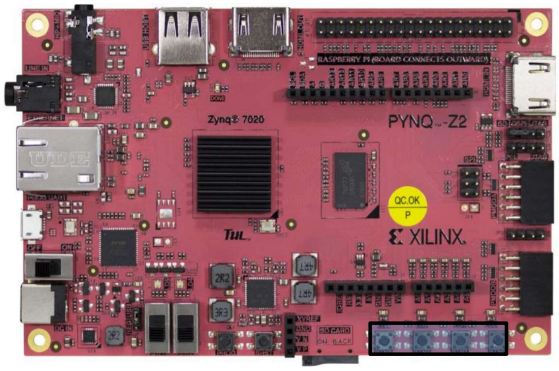
PYNQ-Z2 FPGA Board

- Switches (for giving inputs)



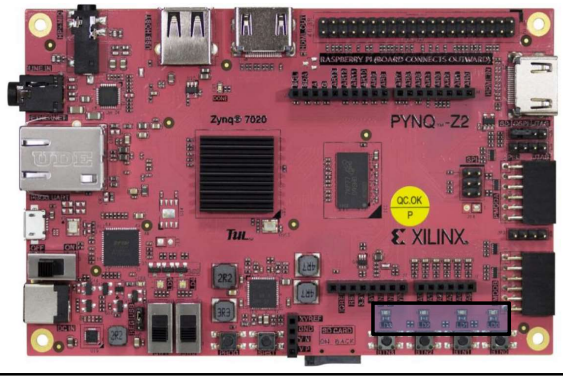
PYNQ-Z2 FPGA Board

- Push buttons (for giving inputs)



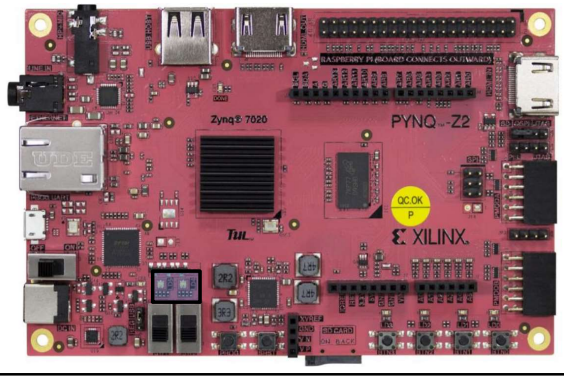
PYNQ-Z2 FPGA Board

- Monochrome LEDs (for seeing outputs)



PYNQ-Z2 FPGA Board

- RGB LEDs (for seeing outputs)



PYNQ-Z2 FPGA Board

- FPGA and processor (Xilinx Zynq Z-7020)

Xilinx Zynq Z-7020

- Formal part name ZYNQ XC7Z020-1CLG400C
- Contains a processor (called processing system) and an FPGA (called programmable logic)
- Processing system (PS)
  - 667 MHz ARM Cortex-A9 processor
  - Has two processing engines (called cores)

14

Xilinx Zynq Z-7020

- Programmable logic (PL)
  - Xilinx Artix-7 FPGA containing the following
  - A 2D array of configurable logic blocks (CLBs)
  - Each CLB has two logic slices
  - Each of the two logic slices has
    - Four six-input 64-entry look-up table (LUT) function generators (can store any six-input function)
    - Eight flip-flops
  - Each CLB has two cascadeable four-bit adders that can be configured into an eight-bit adder

15

Xilinx Zynq Z-7020

- Programmable logic (PL)
  - 13300 logic slices
  - 4.9 Mb of block RAM
    - Organized as 140 blocks of 36 Kb RAM
  - 220 DSP slices
    - Each slice has a 48-bit adder and an 18x25 multiplier

16

Field-programmable gate array

- Two-dimensional array of configurable logic/storage cells (or blocks) interconnected by programmable switches
  - Each cell/block can be programmed to carry out simple combinational functions
  - The interconnection switches can be programmed to decide how the result of one function is input to another function or stored in memory
  - Typically the programming bits are downloaded to the FPGA and the circuit is ready
    - Done in the field as opposed to in the fabrication facility; hence the name FPGA

17

Field-programmable gate array

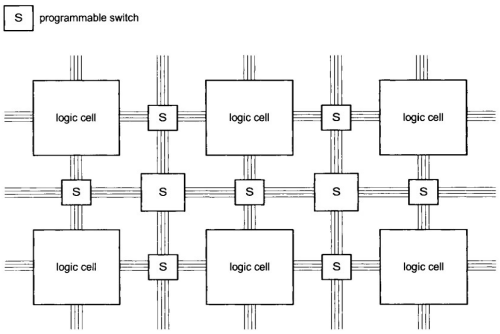


Image source: Chu. FPGA Prototyping by Verilog Examples

18

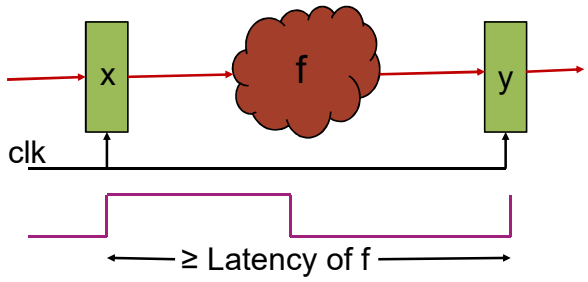
Sequential logic

- Computation proceeds in discrete steps
  - Computation in a step may depend on the states in the previous step
    - Requires storage or memory
  - A signal must define what a step is
    - This is the clock signal
  - Edge-triggered logic changes the states on a clock edge
    - Input to a state storage is sampled on the clock edge
    - After the propagation delay through the state element, the input appears on the output
    - Rising edge (posedge) or falling edge (negedge)
  - Sequential logic doesn't compute, it helps implement storage

Why sequential logic

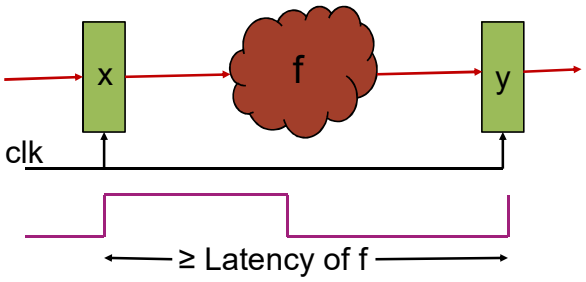
- Results produced by combinational logic needs to be stored
- Also, inputs to combinational logic typically come from storage
  - Computation begins with stored inputs and ends with stored outputs
- Output of the combinational logic gets sampled at the clock edge (rising or falling)
  - The output must be valid by now
  - Needs to be stored before output changes
  - Dictates a lower bound on clock cycle time
    - Latency of combinational logic

Why sequential logic



Why sequential logic

synchronous computation



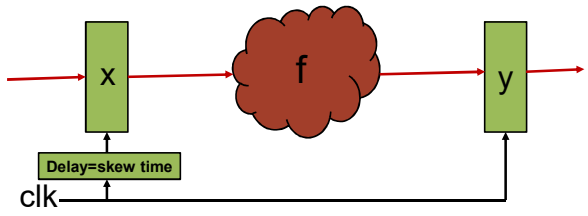
Storage/State elements

- Usually known as memory elements
  - Latches, flip-flops, register files
  - Latches are usually level-sensitive
    - SR latch (set-reset latch) does not have a clock input
      - Cross-coupled NOR gates
  - Flip-flops are edge-triggered
    - Always has a clock input
    - Setup time and hold time for input data
    - Propagation delay to output (clock-to-Q delay)
    - Cycle time is at least the sum of propagation delay, combinational logic delay, setup time, and clock skew
      - Reduce clock skew by careful clock routing
    - Edge-triggering requires more logic; can be slow

Storage/State elements

- Handling clock skew
  - Lengthening the cycle time by the skew time (as discussed in the previous slide) guards against the situation where clock at y arrives before x
    - Since the cycle time has been lengthened by skew time, the input to y would be ready by within cycle time – skew time
  - Lengthening the cycle time alone is not enough if the clock at y arrives after x
    - In this case, an error occurs if skew time is more than propagation delay + min delay through f + setup time
    - The error is that y will not be able to store the result of the last computation
    - Edge at x needs to be delayed by adding logic

Delayed clock to handle skew



Need to identify the paths where such delay should be inserted. This is done in addition to lengthening the cycle time by skew time.

Storage/State elements

- Level-sensitive sequential logic
  - A changing input when the clock is high affects the output
  - Tricky to design because of race through in an array of combinational computations separated by latches
  - Use of two-phase clocks
    - Divide combinational logic into two parts
    - Clock one part with one phase and another with the other phase
    - Alternating phases in the array

Register file

- Array of registers
  - A register can be built with an array of flip-flops
  - Read ports and write ports
    - Input busses for passing register numbers to be read, register number to be written to, data to be written to
    - Output busses carrying the read values
  - Write enable
    - An input that serves as the clock
  - Reading requires big multiplexors per port
  - Writing requires a decoder per port
  - Reading is combinational, writing is sequential
  - Improving read latency requires different design

Asynchronous inputs

- Asynchronous inputs to a flip-flop may arrive at any time
  - Signals coming from outside the computer
  - It may be necessary to convert it to a synchronous signal that changes with clock
    - Otherwise cannot be used correctly in computation
  - Easy way is to pass it through a D flip-flop and use the output of the flip-flop in computation
    - Called a synchronizer
    - Output will change with clock edge and the output is synchronous with the clock (after propagation delay)
    - Fails if the asynchronous input violates setup and hold times

Asynchronous inputs

- Synchronizer failure
  - If the asynchronous input changes within the setup time + hold time window around the clock edge, the flip-flop may go into a metastable state (neither 0 nor 1)
    - It may take some time for the flip-flop to come out of the metastable state; during this time, the output cannot be used in any computation
  - Sample output after sufficient delay so that metastability is resolved with high probability
- Synchronizer with two D flip-flops in a chain
  - Works if metastability period is smaller than clock period

Static random access memory

- Static random access memory (SRAM)
  - Name of a technology used for building reasonably large (up to tens of megabytes) and fast memory
    - Register files are much smaller and faster
  - Specified as height x width
    - 4M x 8 bits
  - Address, chip select, output enable, write enable, and write data are inputs
  - Read data is the output

Static random access memory

- Reading from SRAM
  - Chip select should be asserted for reading from or writing to an SRAM chip
  - Address is decoded to select a row of SRAM
  - The data in the row are read out; called a word
  - The read out word appears on the output when output enable is asserted
    - Without output enable, the output remains in high impedance state
    - Allows tying multiple chips' outputs to the same data bus with the assumption that exactly one chip's output enable will be asserted at a time

Static random access memory

- Writing to SRAM
  - Chip select should be asserted
  - Write enable should be asserted
    - It is a narrow pulse
  - Data to be written should be on the input data bus
  - Address is decoded and the data is written to the appropriate row

Static random access memory

- Building an SRAM
  - Address decoder's outputs are called wordlines
    - Each wordline "enables" a row for a read or a write
  - Each bit is implemented as two cross-coupled inverters along with additional logic to "hold" the stored charge
    - Stored charge (or lack of charge) in a bit represents 1 or 0
    - Without this additional logic, the stored charge in a HIGH bit may leak out slowly leading to loss of memory
  - The bits of a column are tied together to a single wire called bitline through a per-bit "switch"

Static random access memory

- Building an SRAM
  - The bitlines connect to the output bus through tri-state switches
    - The tri-state switches are enabled/disabled by the output enable signal
  - The write enable pulse can be ANDed with the wordline to act as the "clock" for the row to be written to

Static random access memory

- Building an SRAM
  - Very large SRAMs are not built like one big memory due to the large size of the decoder
    - Divided into multiple chips that can operate in parallel
    - Each chip is said to form a bank
    - 4M x 8 could be built as eight 4K x 1024 chips
    - Entire address is now split into row address and column address
    - Possible to also operate only a subset of chips by driving the chip select using certain address bits e.g., 4M x 8 could be built using two "ranks" of eight 2K x 1024 chips (could use address MSB as chip select)
- SRAM is used to build fast memory (register file, caches, etc.) inside the processor chip

Dynamic random access memory

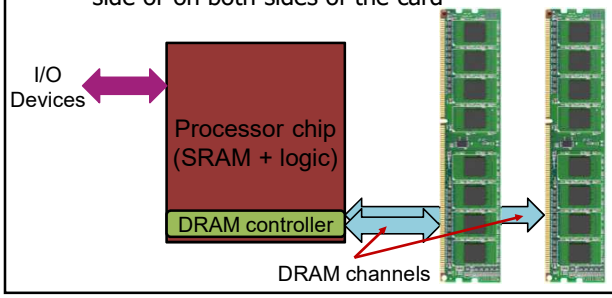
- Dynamic vs. static
  - Random access memory: can access any location by providing an address
  - Static if the charge stored in a bit does not have to be "refreshed" periodically
  - Dynamic if the charge needs to be "refreshed" periodically
    - DRAM contents can leak away if left to itself for sufficient amount of time
    - Each bit is implemented as a capacitor which slowly discharges to ground through "sneak" paths
    - The charge must be refreshed periodically
    - Refreshing is done in bulk and during refresh data cannot be accessed

Dynamic random access memory

- DRAM
  - Why each bit is just a capacitor (implemented using a single transistor)?
    - Additional logic reduces the density of storage (number of bits per unit area)
  - Density is a very important property of DRAM
  - Refreshing is done row by row
    - Involves reading out a row and writing the contents back to the row
- DRAM is used to build external memory outside the processor chip
  - Popularly known as the RAM of a computer
  - Each address refers to a DRAM byte (4 GB DRAM requires a 32-bit address)

Dynamic random access memory

- The RAM of a computer is typically sold as dual-inline memory module (DIMM) cards
  - Each card has several DRAM chips either on one side or on both sides of the card



Dynamic random access memory

- Organization of a DRAM chip
  - Has several banks
  - Each bank stores a 2D array of bits organized in rows and columns
  - Doing a read or a write operation on a DRAM chip requires a bank number, a row address, and a column address
    - The operation is done on the designated bank at the designated row and column
    - The column address typically selects a few consecutive columns determining the output width or the column width of the chip (x4, x8, x16, etc.)
    - Example: 512Mbit DRAM chip: 4 banks, 16K rows, 1K columns, x8

Dynamic random access memory

- Organization of a DRAM bank
  - One wordline per row from the row address decoder
  - The bits in a column are connected to a bitline through a bidirectional switch (also known as a pass transistor or an access transistor)
  - The switch closes when the corresponding wordline goes HIGH; otherwise it is open

Dynamic random access memory

- Read operation to a DRAM bank
  - The row address is sent first on the address bus along with a row address strobe (RAS) signal
  - The command bus carries the command (e.g., read or write)
  - The row address decoder generates the necessary wordline signal
  - The wordline closes the switch and activates or opens the target row
    - The bitlines are precharged at the midpoint of HIGH and LOW
    - When the switch closes, the small difference in bitline voltage is sensed and magnified by an array of sense amplifiers

Dynamic random access memory

- Read operation to a DRAM bank
  - After the row is activated or opened, the entire row is stored in a per-bank area called the row buffer
  - Next the column address is sent on the address bus along with the column address strobe (CAS)
  - The column address reads out one column worth (same as output width) of data and sends that on the DRAM channel
  - It is possible to read out N consecutive columns from the row buffer without sending a new column address every time (except the first time), where N is the burst length of the DRAM module

Dynamic random access memory

- Read operation to a DRAM bank
  - It is much faster to do CAS operations from the row buffer compared to precharging a bank and activating the row over and over
    - Better to do all reads going to the same row of a bank first before changing row in that bank
- Writes are similar, but requires data to be written
- A double data rate (DDR) DRAM can transfer the data on both clock edges
  - Two data tranfers in a cycle; hence called double data rate

Dynamic random access memory

- Organization of a DIMM card
  - A bidirectional channel connects to a DIMM card for carrying data between the card and the memory controller (same as DRAM controller)
    - Channel width is usually governed by standard interfaces
    - JEDEC standards are the only accepted standard
      - Joint Electron Device Engineering Council
    - Channel width is usually 64 or 128 bits
  - In addition to the data channel, there is an address bus and a command bus
    - Unidirectional from memory controller to DIMM

Dynamic random access memory

- Organization of a DIMM card
  - The chips in a DIMM card are divided into several ranks
  - A read or a write command is sent to all the chips in a rank
    - The chip selects of the chips in that rank are made HIGH
  - The memory controller sends the same bank number, row number, and column number to all chips in the target rank
    - The corresponding bank in each chip provide an output
    - For example, with 64-bit channels and x4 chips, 16 chips constitute a rank

Dynamic random access memory

- Utility of burst length (BL)
  - Typically a large chunk of data is fetched together by the processor from DRAM
    - Tries to exploit spatial locality
    - For example, 64 bytes in a large number of processors
  - If BL is 8, an x4 DDR chip will generate four bits of output every half cycle after the first four-bit output
    - Takes four cycles to complete the burst
    - A rank of 16 chips will generate 64-byte output in four cycles after the first 64 bits
- The first 64-bit output on a 64-bit channel may have a variable latency

Dynamic random access memory

- Variable latency of first element of a burst
  - Read to already open/activated row: fastest
    - Only need to send CAS with column address
    - Known as row hit (10 to 20 ns)
  - Read to a bank that has no open/active row
    - Need to activate row and then send CAS
    - Roughly 2x latency compared to fastest
    - Known as row miss
  - Read to a bank that has a different open row
    - Need to close the row (precharge the bank)
    - Need to activate new row and then send CAS
    - Roughly 3x latency compared to fastest
    - Known as row conflict

Dynamic random access memory

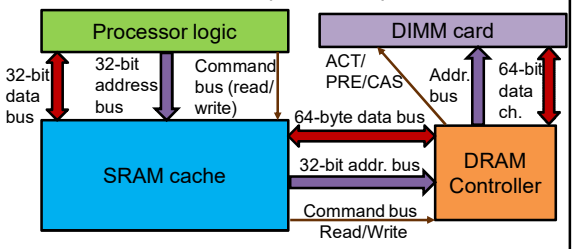
- After reading out the data from rank
  - Transfer over the channel to the memory controller; adds more cycles to overall latency
  - Transfer from memory controller to processor cache (an on-chip fast SRAM storage); even more cycles
    - Stored for quick reuse in future
- Address decoding in memory controller
  - The processor sends read and write commands to the memory controller along with an address
    - Writes need to carry data also
  - The memory controller breaks this address down into channel, rank, bank, row, column

Dynamic random access memory

- Path between processor and DRAM
  - Let us assume that the processor logic can request 32 bits of data at a time from the on-chip SRAM cache
    - Referred to as the processor datapath width
    - For example, for adding two 32-bit numbers, the processor will make two requests one after another to the cache
  - Assume that the address width is 32 bits
    - Need not be same as the datapath width and can be more or less
    - Determines the maximum amount of DRAM that can be installed
      - With 32-bit address, the maximum DRAM capacity is 4 GB

Dynamic random access memory

- Path between processor and DRAM
  - Let us assume that the SRAM cache fetches 64 bytes of data at a time from DRAM
    - Beneficial because of spatial locality



Dynamic random access memory

- Path between processor and DRAM
  - DRAM controller extracts row, rank, bank, column, channel numbers from the address it receives from the SRAM cache
  - If a new row needs to be activated, the command is ACT (or RAS) and the address bus carries the rank, bank, row numbers
  - If a bank needs to be precharged, the command is PRE and the address bus carries the rank and bank numbers
  - If a column read/write needs to be done, the command is CAS and the address bus carries the rank, bank, and column numbers

Dynamic random access memory

- Address decoding in DRAM controller
  - Each address corresponds to a DRAM byte
  - DRAM controller receives an address from the SRAM cache and fetches the 64-byte chunk from DRAM containing that address
    - Achieved by programming the BL to eight
    - Within a bank, the column number is automatically incremented to burst out eight 64-bit chunks
  - Usually the channel number is decoded from the bits right after the least significant six bits of the address covering 64 bytes of data
    - Ensures parallelism across channels for consecutive 64-byte requests
    - Ensures that a request is never split across channels

Dynamic random access memory

- Address decoding in DRAM controller
  - Example decoding (assuming 64-bit channels):

Row	Rank	Bank	Column-high	Channel	Column-low	000

  - Interface to DIMM is eight-byte wide (64-bit ch.)
    - Lower three bits of the address must be zero to make it aligned to eight bytes (a 128-bit ch. has 4 bits zero)
  - Why is row number on the most significant side?
    - The penalty of switching row number is tRP+tRCD compared to a row hit
    - Row number should change least frequently

Dynamic random access memory

- Address decoding in DRAM controller
  - Example decoding (assuming 64-bit channels):

Row	Rank	Bank	Column-high	Channel	Column-low	000

  - Why is row number on the most significant side?
    - Suppose row number is placed to the right of bank number; consider running the following skeleton program on a DRAM chip with four banks and 1024 columns (assume x4 chip)

```
for (i=0; i<100; i++) {
    Access 1024 consecutive bytes of an array
}
```

      - Every outer loop iteration suffers from activating two rows when (Rank, Bank, Row, column-high, channel, column-low) decoding is used
      - Only the first outer loop iteration suffers from activating two rows when (Row, Rank, Bank, column-high, channel, column-low) decoding is used

Dynamic random access memory

- Address decoding in DRAM controller
  - Example decoding (assuming 64-bit channels):

Row	Rank	Bank	Column-high	Channel	Column-low	000

  - Why rank number appears to the left of bank number?
    - There is usually a one-cycle penalty in rank-to-rank switching while there is no penalty in bank-to-bank switching
    - So, rank should change less frequently than bank

Dynamic random access memory

- Address decoding in DRAM controller
  - Example decoding (assuming 64-bit channels):

Row	Rank	Bank	Column-high	Channel	Column-low	000

  - Example: 4 GB DRAM, two 64-bit channels, 1Gb x4 DRAM chips, 32K rows, four banks, BL=8
    - Since BL=8, the block size is 64 bits x 8 or 64 bytes and block offset is 6 bits
    - 32-bit address
    - 2 GB per channel, which means 16 chips per channel
    - One rank per channel, 2K columns (each 4 bits)
    - Column-low=3 bits, Channel=1 bit, Column-high=8 bits, Bank=2 bits, Rank=0 bit, Row=15 bits
    - Column-low is incremented internally in the DRAM chips to cover eight bursts

Dynamic random access memory

- Address decoding in DRAM controller
  - Example decoding (assuming 64-bit channels):

Row	Rank	Bank	Column-high	Channel	Column-low	000

  - Example (continued):
    - Suppose Column-low=3 bits, Channel=1 bit, Column-high=8 bits, Bank=2 bits, Rank=0 bit, Row=15 bits, BL=8, x4 chips; there is only rank per channel
    - CPU needs 32 bits starting at address 0xabcdef94
    - SRAM cache sends address 0xabcdef94 to DRAM controller
      - DRAM controller needs to respond with 64-byte data starting from address 0xabcdef80 to 0xabcdefbf
    - DRAM controller decodes channel no.=0, column no.=1784, bank no.=3, row no.=21990

Dynamic random access memory

- Address decoding in DRAM controller
  - Example decoding (assuming 64-bit channels):

Row	Rank	Bank	Column-high	Channel	Column-low	000

  - Example (continued):
    - DRAM controller decodes channel no.=0, column no.=1784, bank no.=3, row no.=21990
    - Suppose a different row is currently open in bank#3 of the only rank (say, rank#0) in channel#0
    - DRAM controller issues a PRE command to bank#3 of rank#0 in channel#0
    - After tRP cycles, DRAM controller issues an ACT command to row#21990 of bank#3 of rank#0 in channel#0
    - After tRCD cycles, DRAM controller issues a CAS command to column#1784 of bank#3 of rank#0 in channel#0

Dynamic random access memory

- Address decoding in DRAM controller

Row	Rank	Bank	Column-high	Channel	Column-low	000

  - Example (continued):
    - After tCAS cycles, each of the 16 chips in rank#0, bank#3 of channel#0 outputs 4 bits from column#1784
      - Burst#0: Bytes from address 0xabcdef80 to 0xabcdef87
      - Eight bits coming out of the first two chips correspond to the byte at address 0xabcdef80
      - Eight bits coming out of the next two chips correspond to the byte at address 0xabcdef81
      - Eight bits coming out of the last two chips correspond to the byte at address 0xabcdef87
    - Column no. is incremented automatically to 1785
      - Column-low increases from 0 to 1 leaving column-high unchanged

Dynamic random access memory

- Address decoding in DRAM controller

Row	Rank	Bank	Column-high	Channel	Column-low	000

  - Example (continued):
    - After half cycle, each of the 16 chips in rank#0, bank#3 of channel#0 outputs 4 bits from column#1785 and column no. is incremented automatically to 1786
      - Burst#1: Bytes from address 0xabcdef88 to 0xabcdef8f
      - Column-low increases from 1 to 2 leaving column-high unchanged
    - The previous step is repeated six more times
      - Burst#2 to Burst#7
      - Column-low increases by 1 in each step
    - Total time = tRP+tRCD+tCAS+BL/2 cycles (row conflict case)

Dynamic random access memory

- Error detection and correction in DRAM
  - Densely packed DRAM bits can suffer from flips (0 becoming 1 or 1 becoming 0) for certain reasons
    - For correct computation, it is important to detect such errors and possibly abort computation if the error cannot be corrected; need error correction code (ECC)
  - Simple parity code can detect one-bit errors, but cannot correct any error
    - Parity = Bitwise xor of data (no. of 1's odd or even?)
    - One parity bit is stored per DRAM byte: each DRAM address refers to nine bits of data+ECC
    - Complex codes are needed to correct errors
    - SECDED (single error correct double error detect) codes are very popular

Finite state machines (FSMs)

- Every digital design is a marriage of combinational logic and sequential logic
  - Combinational logic does the computation
  - Sequential logic implements the state/storage elements
    - Any realizable design would have a finite number of state elements
    - This leads to a finite number of states e.g., with n state elements (each storing one bit), 2^n different states are possible
  - A digital design with finite number of states is called a finite state machine (FSM)
    - Every realizable digital design is an FSM

Finite state machines (FSMs)

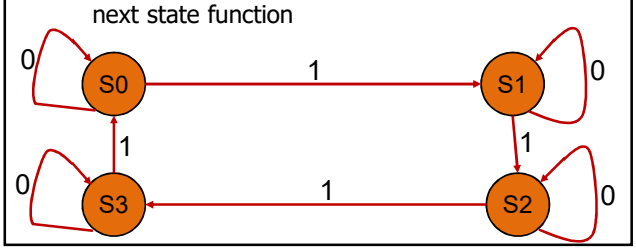
- Each state can have an output value attached to it
  - In a four-state FSM, these values could be 00, 01, 10, 11
- At any point in time, the FSM is in one of its possible states
- The state can change based on the input and the combinational logic
  - For example, in a four-state FSM, the current state 10 can change to 11 on an input 0; the current state 10 can change to 00 on an input 1

Finite state machines (FSMs)

- Next state function
  - A function that defines the state transitions when inputs are applied to the FSM
  - In a four-state FSM, the function defines the new state when the current state receives a 0 or a 1 input; F: S x I → S
    - Eight possible entries in the function: F(S0, 0), F(S0, 1), F(S1, 0), F(S1, 1), F(S2, 0), F(S2, 1), F(S3, 0), F(S3, 1) where S0, S1, S2, S3 are the four states
  - We will be interested only in synchronous FSMs where the state changes take place on clock edges (after propagation delay)
    - Clock is an implicit input to all FSMs of our interest

Finite state machines (FSMs)

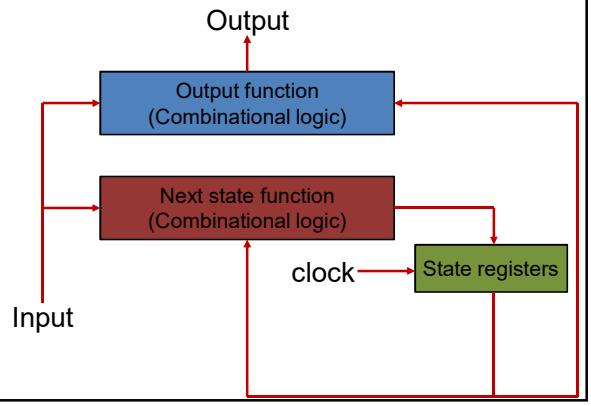
- Next state function: an example
  - F(S0, 0) = S0, F(S0, 1) = S1, F(S1, 0) = S1, F(S1, 1) = S2, F(S2, 0) = S2, F(S2, 1) = S3, F(S3, 0) = S3, F(S3, 1) = S0
  - Usually a state diagram is used to represent the next state function



Finite state machines (FSMs)

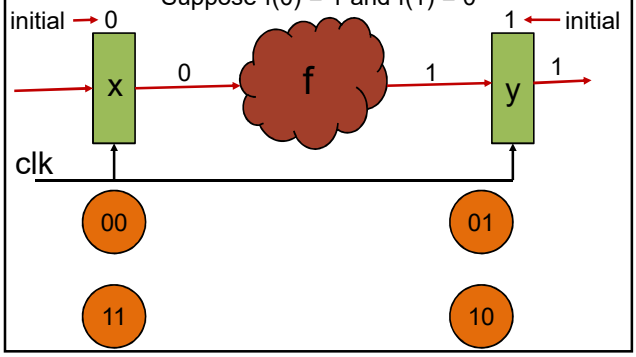
- Output function
  - The FSM can output a value based on its current state and input; G: S x I → N
  - In a four-state FSM, there are eight possible output function entries
  - Example: G(S0, 0) = red, G(S0, 1) = green, G(S1, 0) = blue, G(S1, 1) = black, G(S2, 0) = red, G(S2, 1) = green, G(S3, 0) = orange, G(S3, 1) = brown
  - If the output function depends only on current state, the FSM is called a Moore machine
  - If the output function depends only on current state and input, the FSM is called a Mealy machine

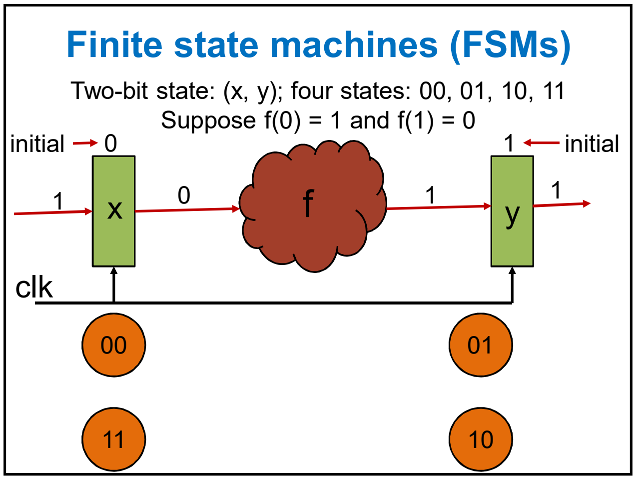
Finite state machines (FSMs)

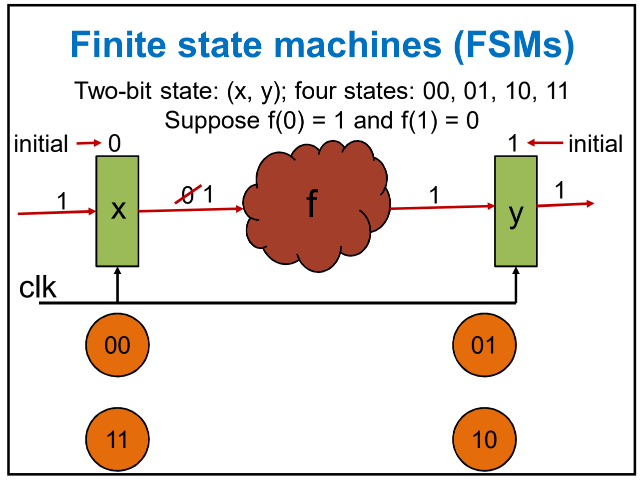


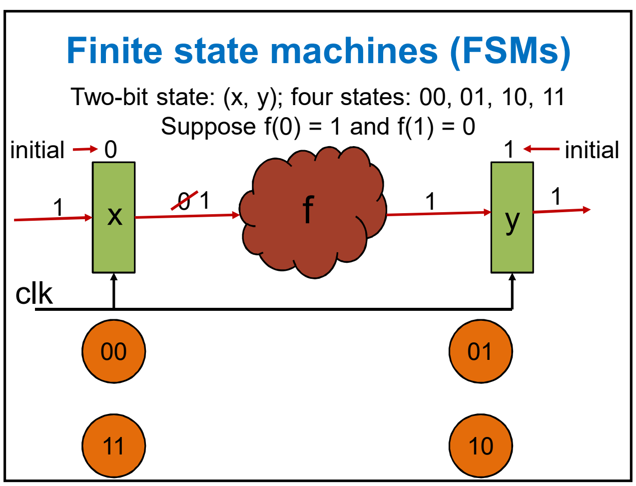
Finite state machines (FSMs)

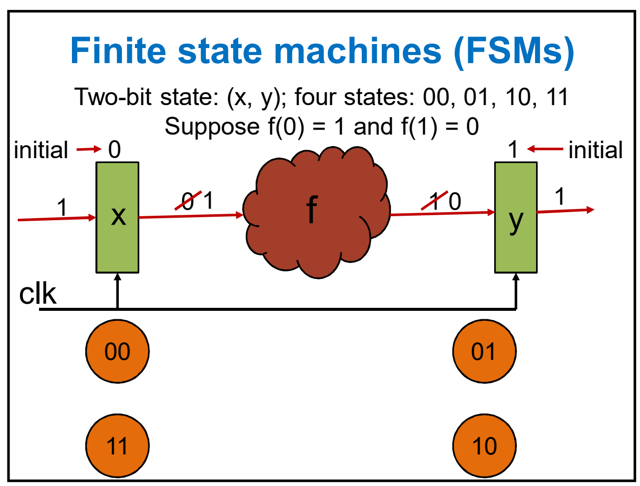
Two-bit state: (x, y); four states: 00, 01, 10, 11
Suppose f(0) = 1 and f(1) = 0

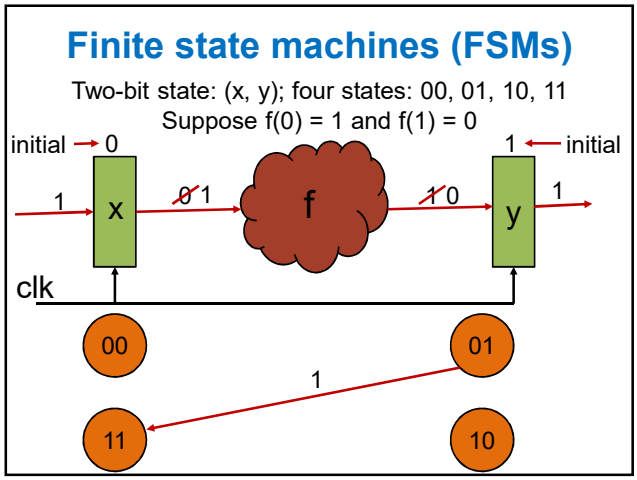


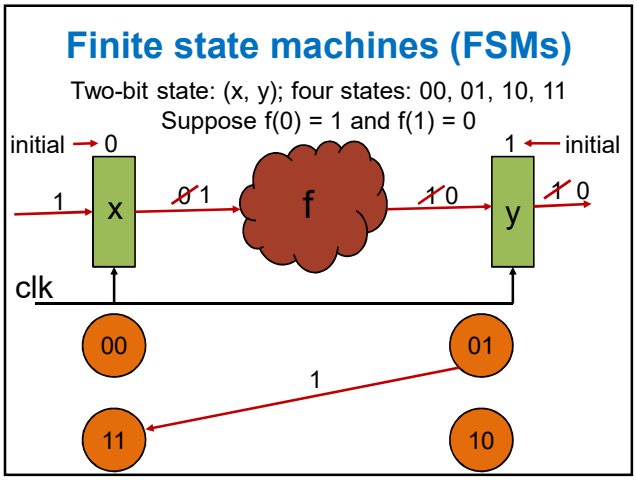

Finite state machines (FSMs)
Two-bit state: (x, y); four states: 00, 01, 10, 11
Suppose f(0) = 1 and f(1) = 0
initial
0
1
initial
1
x
0
f
1
y
1
clk
00
01
11
10

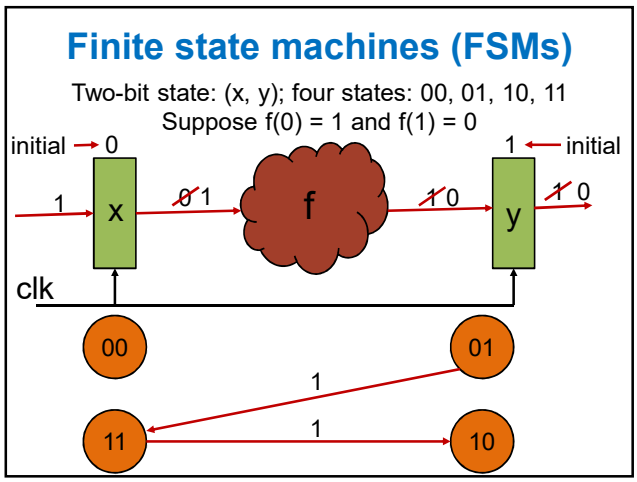

Finite state machines (FSMs)
Two-bit state: (x, y); four states: 00, 01, 10, 11
Suppose f(0) = 1 and f(1) = 0
initial
0
1
initial
1
x
0 1
f
1
y
1
clk
00
01
11
10

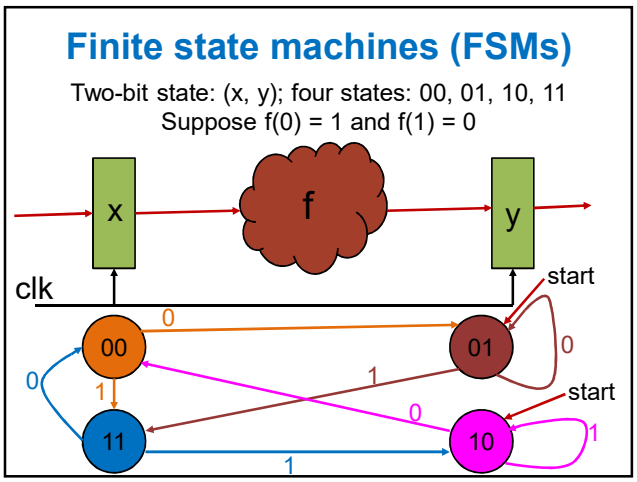

Finite state machines (FSMs)
Two-bit state: (x, y); four states: 00, 01, 10, 11
Suppose f(0) = 1 and f(1) = 0
initial
0
1
initial
1
x
0 1
f
1
y
1
clk
00
01
11
10


Finite state machines (FSMs)
Two-bit state: (x, y); four states: 00, 01, 10, 11
Suppose f(0) = 1 and f(1) = 0
initial
0
1
initial
1
x
0 1
f
1 0
y
1
clk
00
01
11
10

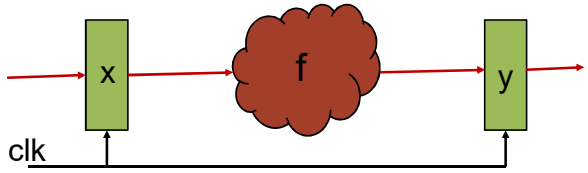

Finite state machines (FSMs)
Two-bit state: (x, y); four states: 00, 01, 10, 11
Suppose f(0) = 1 and f(1) = 0
initial
0
1
initial
1
x
0 1
f
1 0
y
1
clk
00
01
1
11
10


Finite state machines (FSMs)
Two-bit state: (x, y); four states: 00, 01, 10, 11
Suppose f(0) = 1 and f(1) = 0
initial
0
1
initial
1
x
0 1
f
1 0
y
1 0
clk
00
01
1
11
10


Finite state machines (FSMs)
Two-bit state: (x, y); four states: 00, 01, 10, 11
Suppose f(0) = 1 and f(1) = 0
initial
0
1
initial
1
x
0 1
f
1 0
y
1 0
clk
00
01
1
11
1
10


Finite state machines (FSMs)
Two-bit state: (x, y); four states: 00, 01, 10, 11
Suppose f(0) = 1 and f(1) = 0
x
f
y
clk
start
0
00
01
0
0
1
1
start
0
11
10
1
1

Finite state machines (FSMs)



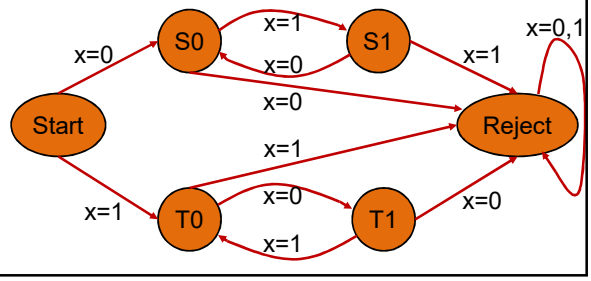
If the environment is sampling the output only at y, the output function depends only on the current state and is equal to the y bit of the state

Finite state machine design

- Step#1: derive the FSM from problem description
  - This is usually the challenging step
- Step#2: decide the state elements
- Step#3: design the combinational logic for the next state function
- Step#4: design the combinational logic for the output function

Finite state machine design

- Example
  - We would like to design a digital logic that detects if the input pattern is alternating i.e., the input is 101010… or 010101…



Finite state machine design

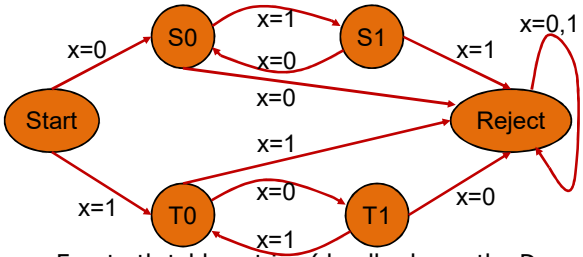
- Example: detecting alternating input patterns
  - Output of Reject state is 0 and of all other states is 1
    - Could drive an LED with the output and the LED will keep glowing as long as the input alternates
    - Empty and single-bit inputs are assumed to be alternating
  - Six states require three state bits
    - Call these a, b, c. Assume following state encoding:
    - Start = 000 or ~a & ~b & ~c, S0 = 001 or ~a & ~b & c, S1 = 010 or ~a & b & ~c, T0 = 011 or ~a & b & c, T1 = 100 or a & ~b & ~c, Reject = 101 or a & ~b & c
    - Output = ~Reject = ~a | b | ~c
      - Includes states 110 and 111 also, but that is not wrong

Finite state machine design

- Example: detecting alternating input patterns
  - Next step is to design the next state function
    - Purely combinational logic
    - Can be designed by preparing a truth table where the inputs are current states and x and output is the next state
    - The goal is to write down three combinational logic formulas that determine the D inputs to the three flip-flops storing a, b, c

Finite state machine design

- Example: detecting alternating input patterns



  - Few truth table entries (da, db, dc are the D inputs of the a, b, c flip-flops)

a,b,c,x=0,0,0,0 then da,db,dc=0,0,1 (this is S0)
a,b,c,x=0,0,0,1 then da,db,dc=0,1,1 (this is T0)

Finite state machine design

- Example: detecting alternating input patterns
  - Next state function's truth table may not have all entries fully specified
  - In this example, the following four entries (out of sixteen) will not have any next state specified

a,b,c,x=1,1,0,0
a,b,c,x=1,1,0,1
a,b,c,x=1,1,1,0
a,b,c,x=1,1,1,1

  - da,db,dc can be taken as anything (don't cares) in these cases to simplify the combinational logic

Finite state machine design

- Example: detecting alternating input patterns

a,b,c,x=0,0,0,0 da,db,dc=0,0,1 (this is S0)
a,b,c,x=0,0,0,1 da,db,dc=0,1,1 (this is T0)
a,b,c,x=0,0,1,0 da,db,dc=1,0,1 (this is Reject)
a,b,c,x=0,0,1,1 da,db,dc=0,1,0 (this is S1)
a,b,c,x=0,1,0,0 da,db,dc=0,0,1 (this is S0)
a,b,c,x=0,1,0,1 da,db,dc=1,0,1 (this is Reject)
a,b,c,x=0,1,1,0 da,db,dc=1,0,0 (this is T1)
a,b,c,x=0,1,1,1 da,db,dc=1,0,1 (this is Reject)
a,b,c,x=1,0,0,0 da,db,dc=1,0,1 (this is Reject)
a,b,c,x=1,0,0,1 da,db,dc=0,1,1 (this is T0)
a,b,c,x=1,0,1,0 da,db,dc=1,0,1 (this is Reject)
a,b,c,x=1,0,1,1 da,db,dc=1,0,1 (this is Reject)

Finite state machine design

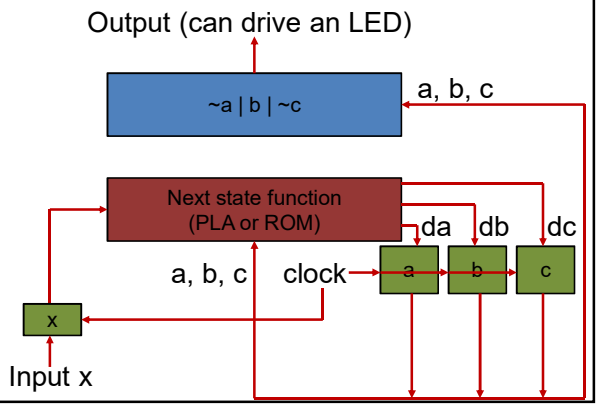
- Example: detecting alternating input patterns
  - Next state function

    da = (~x & (c | a)) | (a & (b | c)) | (b & x)

    db = (a & b) | (~a & ~b & x) | (a & ~c & x)

    dc = ~c | a | (b & x) | (~b & c & ~x)
  - Possible implementations of the next state combinational logic
    - PLA (requires deriving the formulas)
    - ROM (requires only the truth table): 12 rows x 3 bits
  - If the next state function is implemented with a ROM, we can store the output function also in ROM and remove the output function logic
    - One extra column of bits in the ROM for this example

Finite state machine design



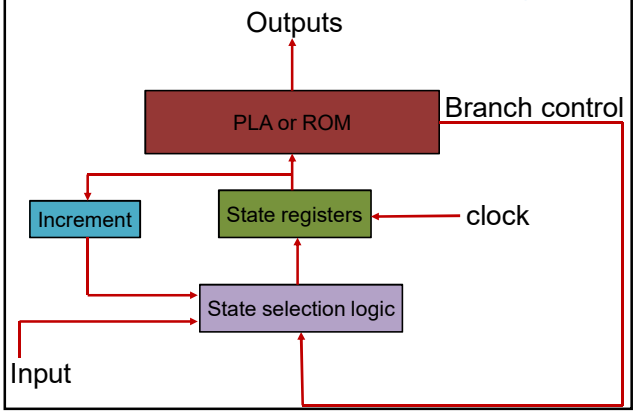
Finite state machine design

- The output function, the next state function, and the states constitute the control unit of the FSM
- The outputs of the FSM may drive the external datapaths
  - In this example, the datapath is as simple as a discrete LED
  - In more complex examples, the FSM outputs can be used as inputs to other computations

Finite state machine design

- Implementing next state function using a sequencer
  - With an increasing number of states, it becomes difficult to design the next state function using PLA or ROM
  - In many large FSMs, the states are often visited sequentially
    - Appropriate state assignment can make it even more often
  - In such FSMs, the next state function can be implemented using a simple counter with the option of "branching out" occasionally
    - Branching happens when the next state is not current state + 1 or there are multiple possible next states

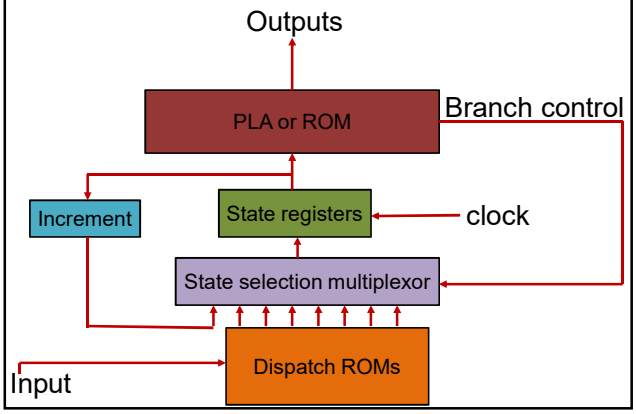
Finite state machine design



Finite state machine design

- Implementing next state function using a sequencer
  - The PLA or the ROM now implements the output function and the branch control
  - The non-consecutive next states are handled by reading out the next state from separate ROMs in such cases
    - These are called dispatch ROMs
    - One dispatch ROM for each non-sequential case or all dispatch ROMs can be fused into one dispatch ROM
    - Input decides the row of the dispatch ROMs

Finite state machine design



Finite state machine design

- Implementing next state function using a sequencer
  - For the states, where the next state is just the incremented state, the branch control is stored as zero in the ROM
    - Ensures that the incremented state will be selected by the multiplexor
  - For each of the remaining states, the branch control specifies which dispatch ROM's output should be selected
    - Each dispatch ROM is indexed by the input and provides one output
  - The ROM that stores the output function and the branch control is much smaller in width now

Finite state machine design

- Implementing next state function using a sequencer
  - Each entry of the ROM that implements the outputs and the branch control is often seen as storing the instruction for controlling the next state and the output
    - Referred to as a microcode ROM
    - Each ROM entry is referred to as a microinstruction
    - The state registers together form the microinstruction address or the microprogram counter which determines the ROM entry of the next microinstruction
  - This style of FSM control implementation is known as microprogrammed control unit design