UF Herbert Wertheim College of Engineering
UNIVERSITY *of* FLORIDA

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE

Department of Electrical & Computer Engineering

Digital Design
Chapter 05 – Integration

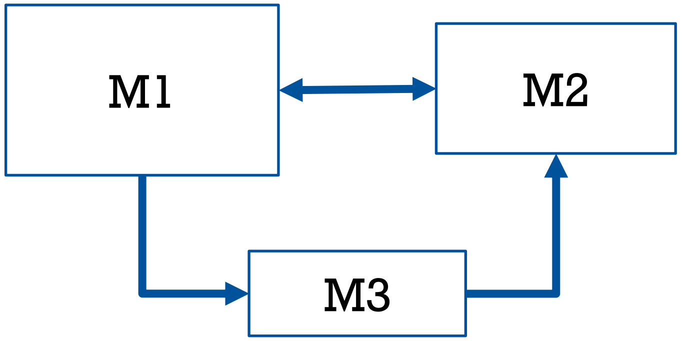
Dr. Christophe Bobda

EEL4712C Spring 2024

Agenda

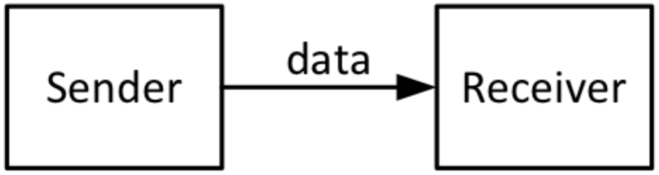
- ❑ Tackling complexity
- ❑ Interfacing
- ❑ Interconnect
- ❑ Memory

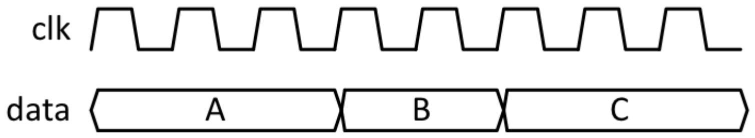
Complex Systems



- Divide and conquer to tackle system complexity
  - System partitioning.
  - Component integration.
- Component integration requires well-defined interfaces
  - Interconnection
    - Physical connections among system components: where the data flow.
  - Protocol/timing
    - Data transfer methodology: how the data flow.
    - A convention for sequencing the transfer of data.
    - To transfer a datum from a source module S to a destination module D, we need to know
      - when the datum is valid (i.e., when the source module S has produced the datum and placed it on its interface pins) and
      - when D is ready to receive the datum (i.e., when D samples the datum from its interface pins.

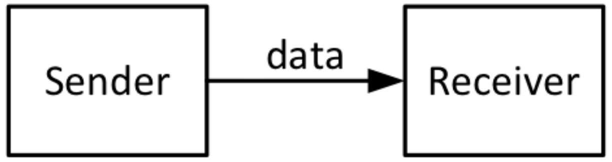
Always Valid Timing/Protocol

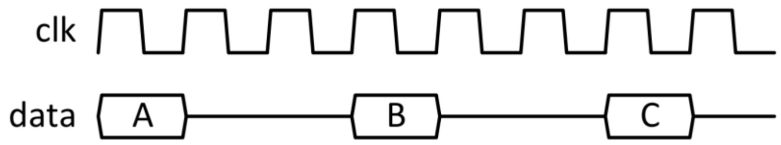




- As the name implies, data is always valid.
- Does not require any sequencing signals.
- The datum is valid every cycle and can be sampled by the receiver at any time.
- Example:
  - A temperature sensor that constantly outputs an eight-bit digital value representing the current temperature.
  - The ballPos, leftPadY, rightPadY, and score signals in the pong game.
  - A static or constant signal is a special case of an always valid signal where the signal is guaranteed not to change values between specified events (e.g., system resets).

Periodically Valid Timing

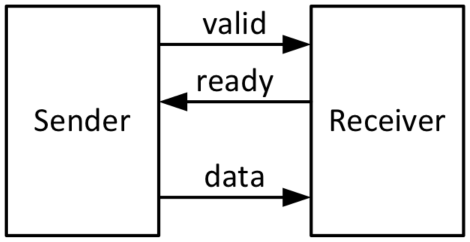


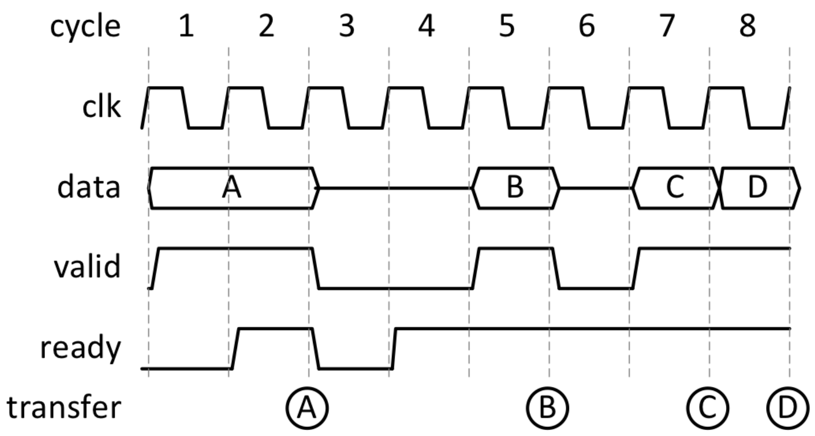


- Signal is valid once every N cycles.
  - The interval N is the period of the signal.
- Unlike an always valid signal, each value of a periodically valid signal represents a particular event, task, or token, and cannot be dropped or duplicated
- This distinction between always valid and periodically valid signals becomes apparent when we move signals between clock domains.
  - It is easy to move always valid signals between clock domains as long as we avoid synchronization failure because it is acceptable to duplicate or drop values.
  - On the other hand, flow control is required in order to move a periodically valid signal with a period of one clock cycle across clock domains.

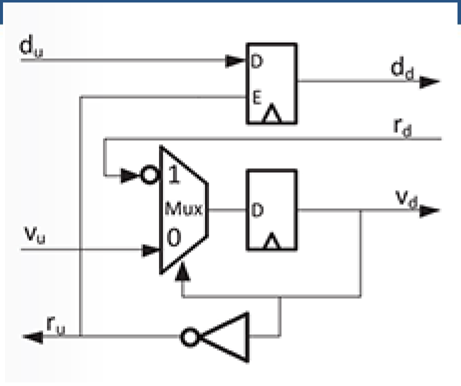
Flow Control

- Flow control uses explicit sequencing signals (valid and ready), to sequence the transfer of data over the interface.
  - The sending module signals when a valid datum is present on the interface by asserting valid.
  - The receiving module indicates that it is ready to accept a new datum by asserting ready.
  - The datum is transferred only when both valid and ready are asserted.





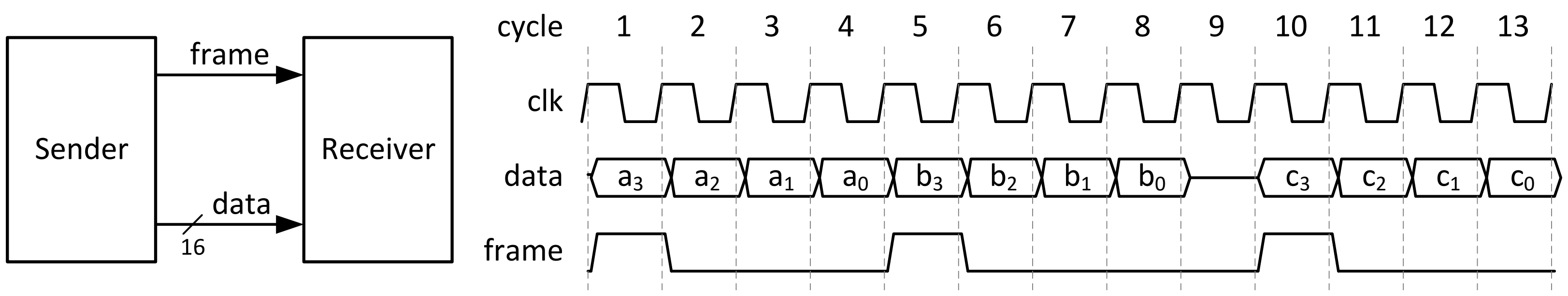
Flow-control



- Example: Flow-controlled registers
  - Design a register in which the input and output data uses flow-controlled signaling. Such a register can be used to split long, high-delay wires that are on the critical path of the design.
  - Communicating with the upstream module: du (data input), ru (ready output), and vu (valid input);
  - Communication with downstream: dd (output), rd (input), and vd (output)
  - Every cycle, when no valid data are stored (vd = 0), both the valid register and data are updated with the upstream value.
  - The buffer also signals upstream that it is ready to accept new data (ru = 1). When valid data are stored in the register (vd = 1), the data register is disabled, holding the stored value.
  - If the downstream unit is not ready, vd is held high and ru is kept low.
  - When ready, the downstream unit asserts rd, causing the vd to be deasserted on the next cycle.
  - With only one register and no combinational paths between stages, this module can accept new data only every other cycle.

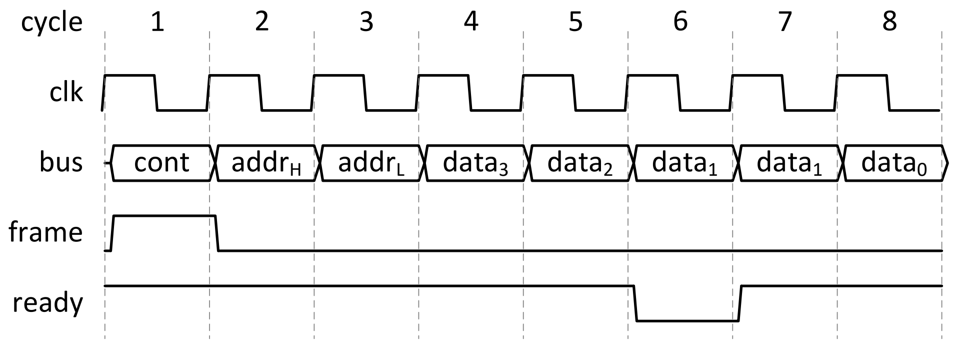
Serialization

- To transfer a large datum with a low duty factor, it may be advantageous to serialize the datum, transferring it over many cycles, one part per cycle over a narrower interface.
- Example: an interface to transfer a 64-bit block of data once every four cycles over a 16-bit interface.
  - Sending one-quarter of the block each cycle. On the first cycle a3 (a(63 downto 48)) is transferred, in the second cycle a2 is transferred, and so on

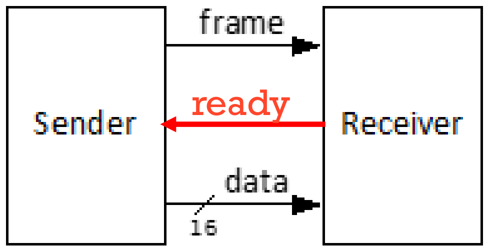

Sender
frame
data
16
Receiver
cycle 1 2 3 4 5 6 7 8 9 10 11 12 13
clk
data
a_3 a_2 a_1 a_0 b_3 b_2 b_1 b_0 c_3 c_2 c_1 c_0
frame

Serialization with word granularity FC

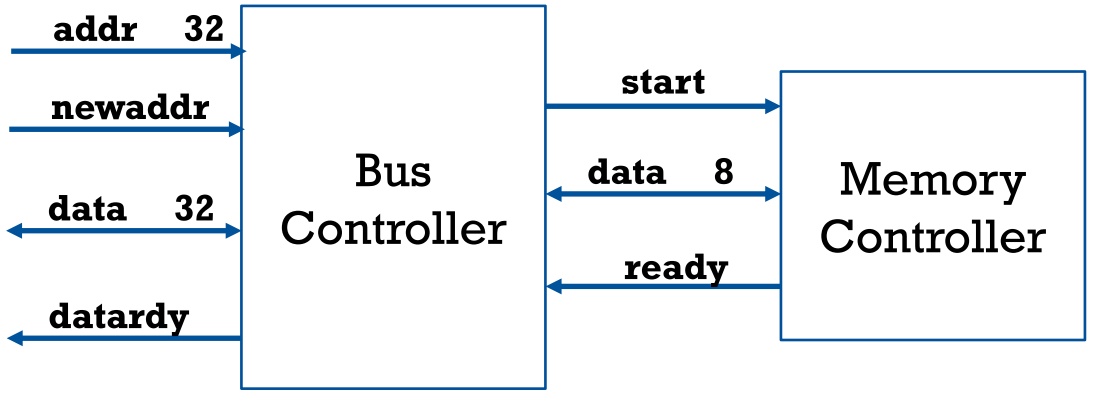
- Memory and I/O interfaces often serialize the command, address, and data fields to transmit them over a shared, narrow bus.
  - In the figure a memory transaction is serialized over a byte-wide interface over seven cycles
    - control sent on the first cycle,
    - address sent over two cycles,
    - and data sent over four cycles.



  - This example uses cycle-valid/frame-ready flow control.
    - The frame signal indicates that an entire frame of data is ready and
    - The ready signal indicates receiver readiness on a cycle-by-cycle basis.
    - The receiver signals not ready during cycle 6, causing data1 to be retransmitted in cycle 7.

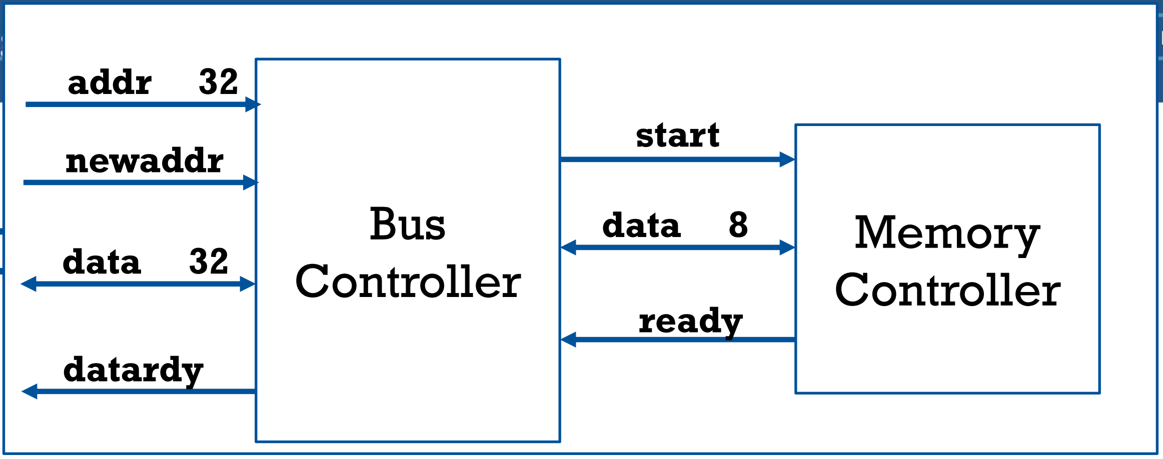


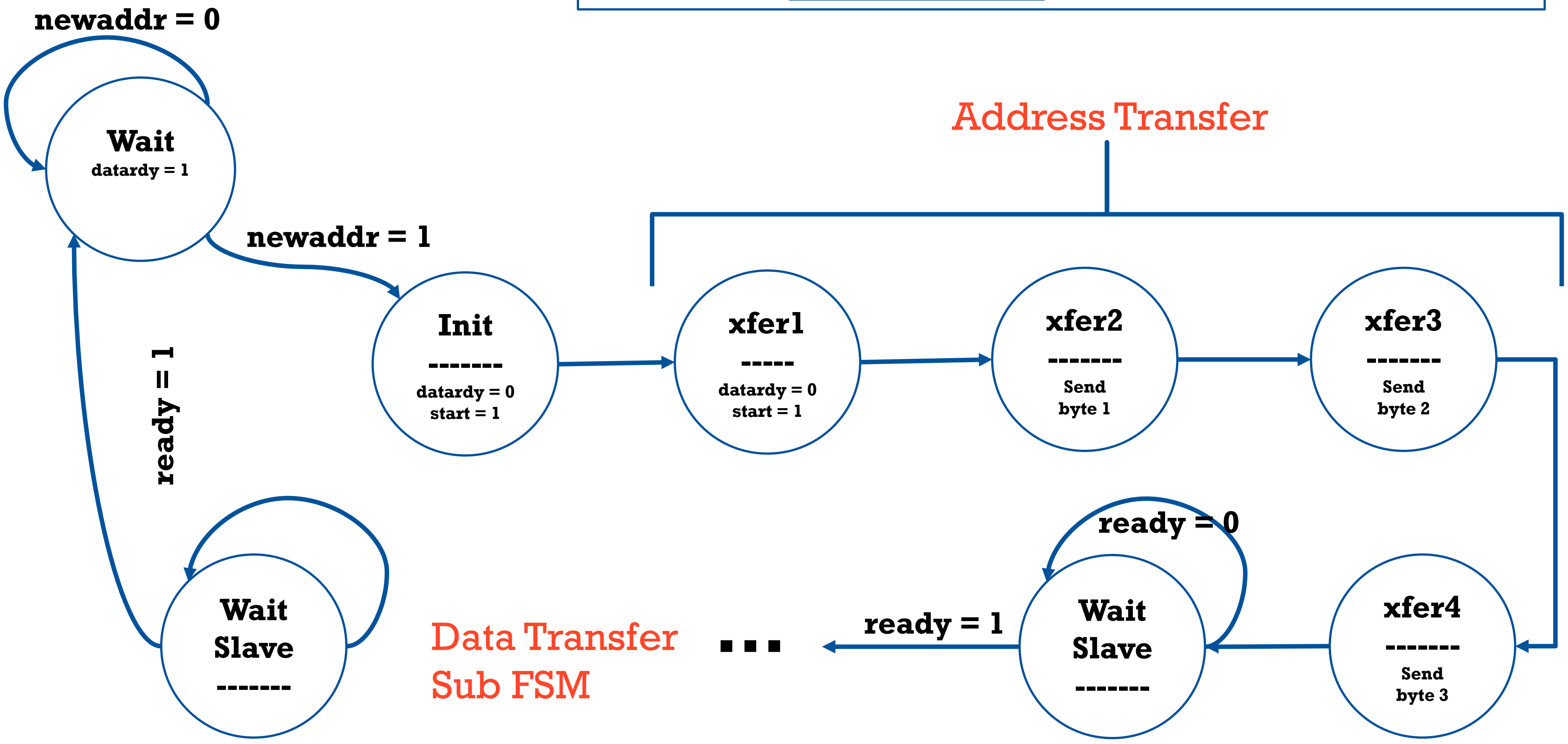
Example: Memory-Bus-Interfacing



```
Entity BusControl is
 port(clock, newaddr, ready: in std_logic;
      start, datardy : out std_logic;
       addr: in std_logic_vector(31 downto 0);
       data: inout std_logic_vector(31 downto 0);
      data8 inout std_logic_vector(7 downto 0));
End entity BusControl;
```

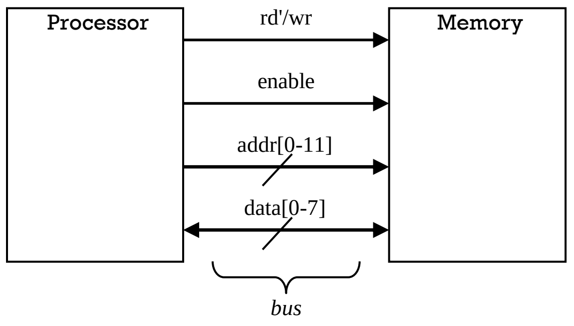
Example: Memo





Generalized Communication Protocols

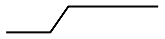
- Wires:
  - Uni-directional or bi-directional
  - One line may represent multiple wires
- Bus
  - Set of wires with a single function
    - Address bus, data bus
  - Or, entire collection of wires
    - Address, data and control
    - Associated protocol: rules for communication

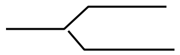


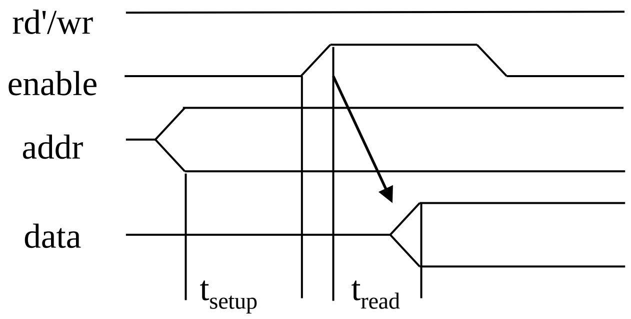
bus structure

Timing Diagrams

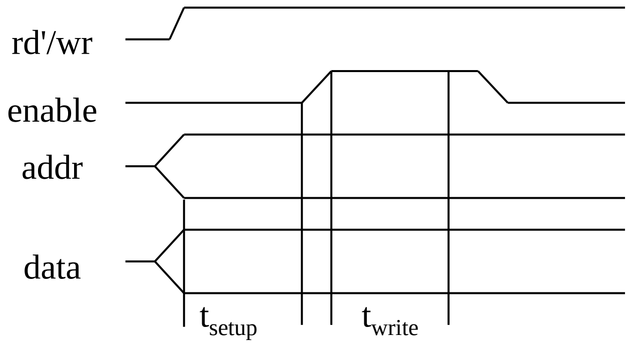
- Most common method for describing a communication protocol
- Time proceeds to the right on x-axis
- Control signal: low or high
  - May be active low (e.g., go', /go, or go_L)
  - Use terms *assert* (active) and *deassert*
  - Asserting go' means go=0
- Data signal: not valid or valid
- Protocol may have subprotocols
  - Called bus cycle, e.g., read and write
  - Each may be several clock cycles
- Read example
  - *rd'/wr* set low,address placed on *addr* for at least t_{setup} time before *enable* asserted, enable triggers memory to place data on *data* wires by time t_{read}







read protocol

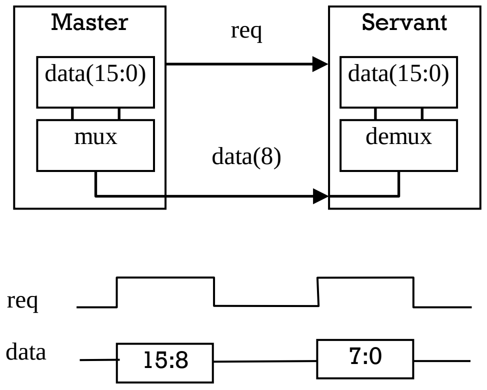


write protocol

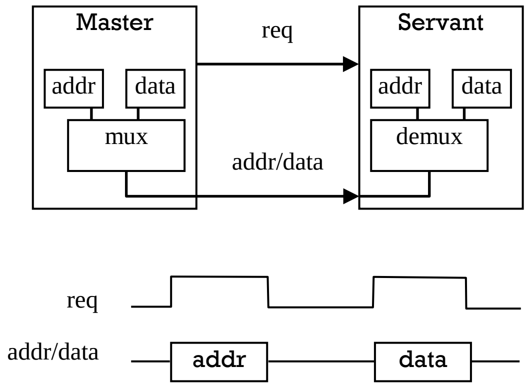
Basic protocol concepts

- Actor: master initiates, servant (slave) respond
- Direction: sender, receiver
- Addresses: special kind of data
  - Specifies a location in memory, a peripheral, or a register within a peripheral
- Time multiplexing
  - Share a single set of wires for multiple pieces of data
  - Saves wires at expense of time

Time-multiplexed data transfer

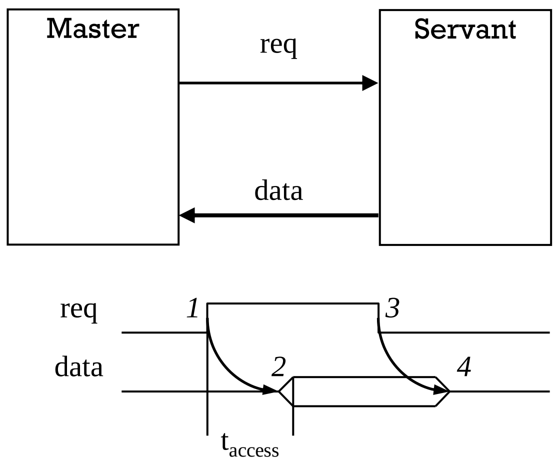


data serializing



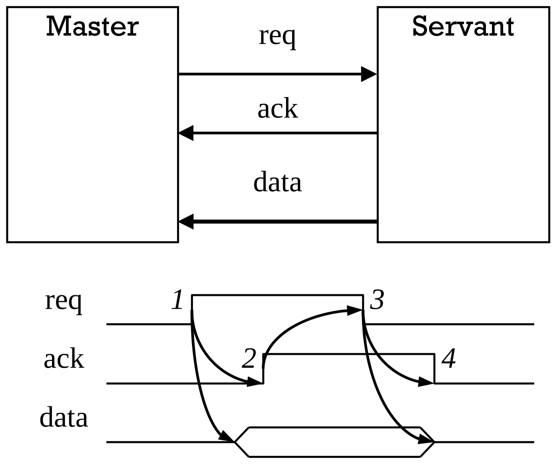
address/data muxing

Basic protocol concepts: control methods



1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}**
3. Master receives data and deasserts *req*
4. Servant ready for next request

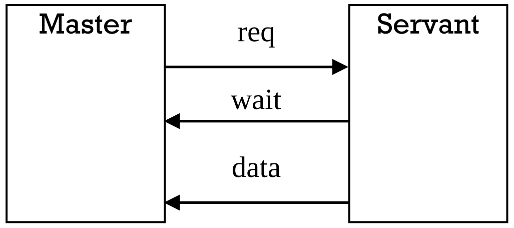
Strobe protocol

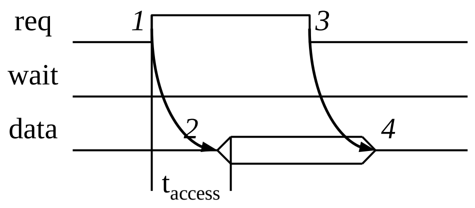


1. Master asserts *req* to receive data
2. Servant puts data on bus **and asserts *ack***
3. Master receives data and deasserts *req*
4. Servant ready for next request

Handshake protocol

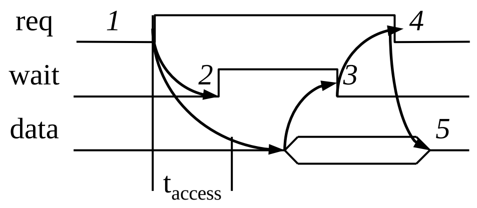
A strobe/handshake compromise





1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}** (wait line is unused)
3. Master receives data and deasserts *req*
4. Servant ready for next request

Fast-response case

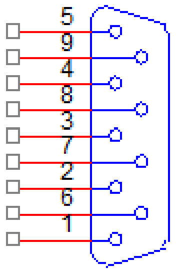


1. Master asserts *req* to receive data
2. Servant can't put data within **t_{access}**, **asserts *wait*** ack
3. Servant puts data on bus and **deasserts *wait***
4. Master receives data and deasserts *req*
5. Servant ready for next request

Slow-response case

UART

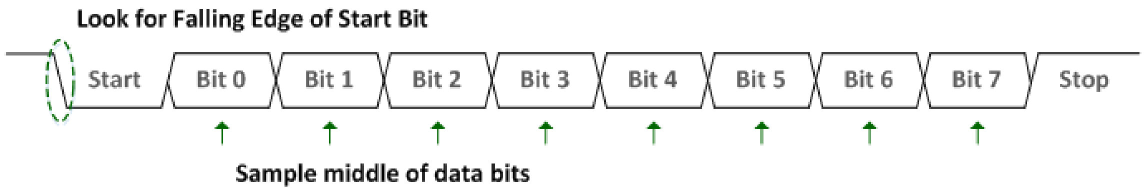
- Universal Asynchronous Transmitter Receiver.
  - To convert parallel data (8 bit) to serial data.
  - UART transmits bytes of data sequentially one bit at a time from source and receive the byte of data at the destination by decoding sequential data with control bits.
  - asynchronous communication.

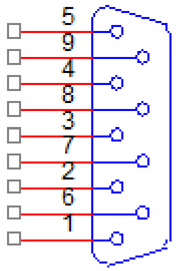


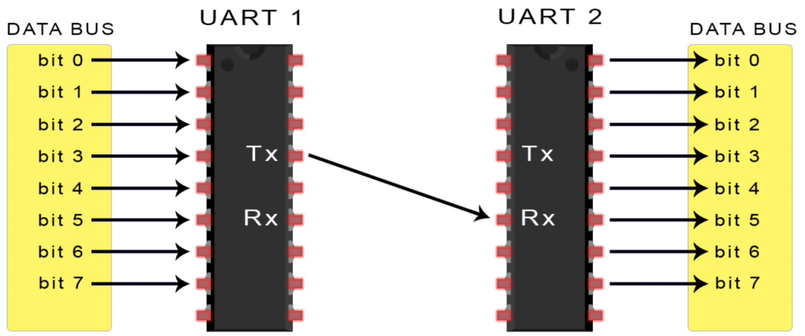
Pin no	Signal
1	Data carrier detect(DCD)
2	DReceived data(RD)
3	Transmitted data(TD)
4	Data terminal ready(DTR)
5	Signal ground(GND)
6	Data set ready(DSR)
7	Request to send(RS)
7	Clear to send(CS)
8	Ring indicator(RI)

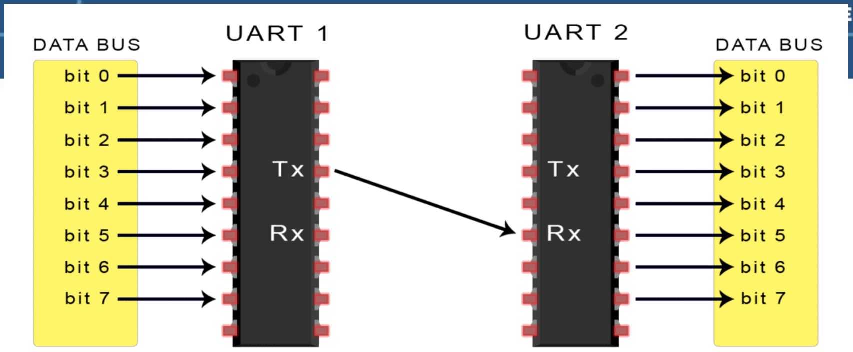
UART

- Universal Asynchronous Transmitter Receiver.
  - To convert parallel data (8 bit) to serial data.
  - UART transmits bytes of data sequentially one bit at a time from source and receive the byte of data at the destination by decoding sequential data with control bits.
  - asynchronous communication.


Look for Falling Edge of Start Bit
Start
Bit 0
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
Stop
Sample middle of data bits

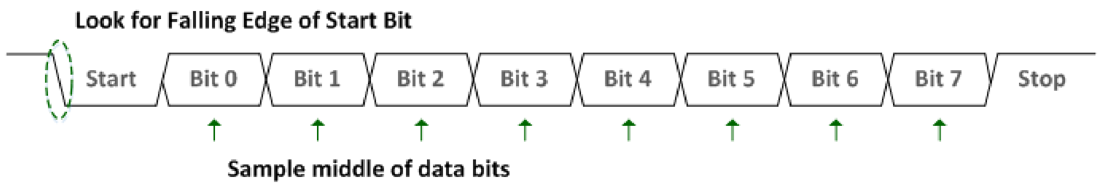




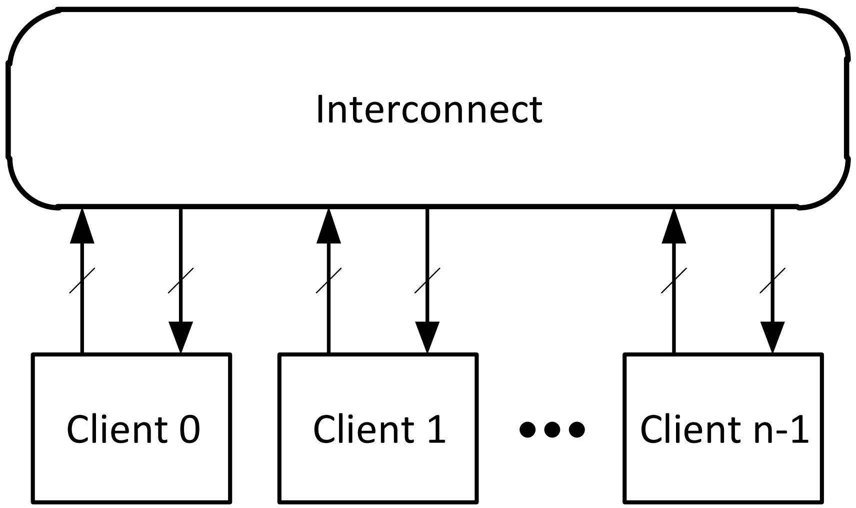


UART

- Transmission protocol
  - Start Bit
    - To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle.
  - Data Frame
    - The data frame contains the actual 8-bit data being transferred.
  - Parity
    - Additional bit to detect that an error has occurred.
  - Stop bit
    - To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for at least two bit durations.

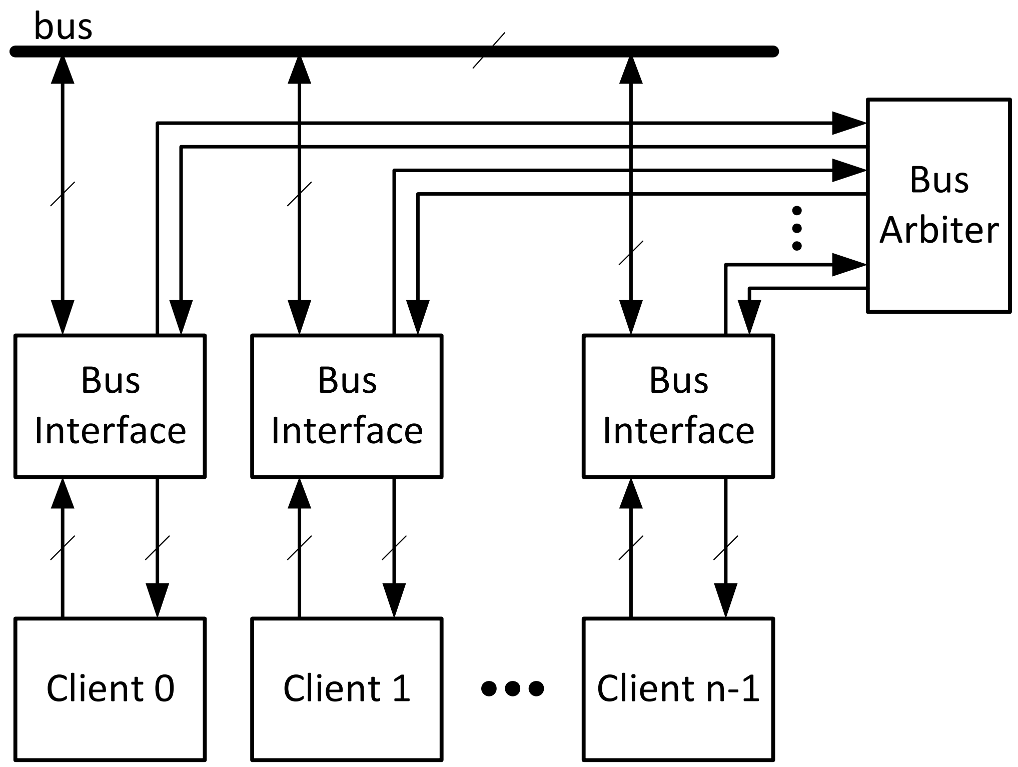

Look for Falling Edge of Start Bit
Start
Bit 0
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
Stop
Sample middle of data bits

Interconnect



- Many clients need to communicate
- Ad-hoc point-to-point wiring or shared interconnect
- Like a telephone exchange

Bus


bus
Bus Arbiter
Bus Interface
Bus Interface
Bus Interface
Client 0
Client 1
Client n-1

VHDL for a simple bus interface

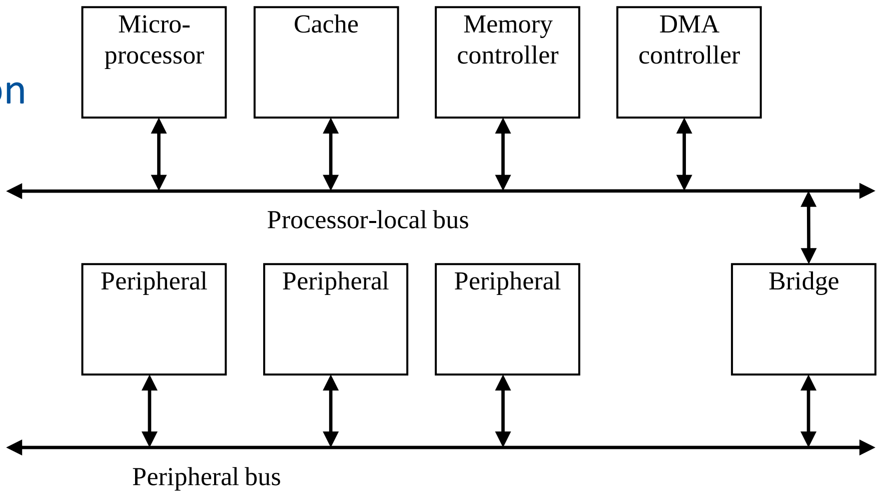
```
-- Combinational Bus Interface
-- t (transmit) and r (receive) in signal names are from the
-- perspective of the bus
library ieee;
use ieee.std_logic_1164.all;
entity BusInt is
  generic( aw: integer := 2;    -- address width
           dw: integer := 4 ); -- data width
  port( cr_valid, arb_grant, bt_valid: in std_logic;
        cr_ready, ct_valid, arb_req, br_valid: out std_logic;
        cr_addr, bt_addr, my_addr: in std_logic_vector(aw-1 downto 0);
        br_addr: out std_logic_vector(aw-1 downto 0);
        cr_data, bt_data: in std_logic_vector(dw-1 downto 0);
        br_data, ct_data: out std_logic_vector(dw-1 downto 0) );
end BusInt;

architecture impl of BusInt is
begin
  -- arbitration
  arb_req <= cr_valid;
  cr_ready <= arb_grant;

  -- bus drive
  br_valid <= arb_grant;
  br_addr <= cr_addr when arb_grant else (others => '0');
  br_data <= cr_data when arb_grant else (others => '0');
```

```
  -- bus drive
  br_valid <= arb_grant;
  br_addr <= cr_addr when arb_grant else (others => '0');
  br_data <= cr_data when arb_grant else (others => '0');
```

```
  -- bus receive
  ct_valid <= '1' when (bt_valid = '1') and (bt_addr =
my_addr) else '0';
  ct_data <= bt_data ;
end impl;
```

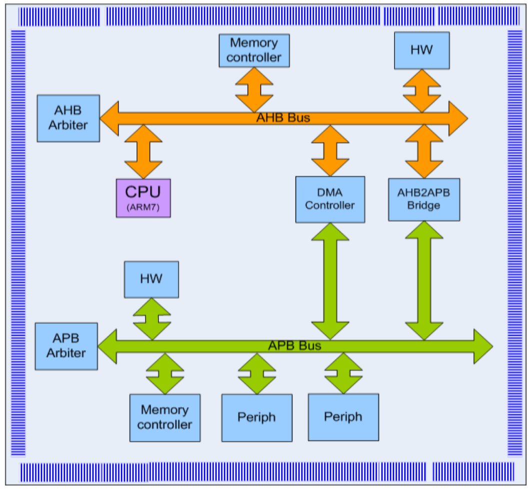
Multilevel bus architectures

- Don't want one bus for all communication
  - Peripherals would need high-speed, processor-specific bus interface
    - excess gates, power consumption, and cost; less portable
  - Too many peripherals slows down bus
- Processor-local bus
  - High speed, wide, most frequent communication
  - Connects microprocessor, cache, memory controllers, etc.
- Peripheral bus
  - Lower speed, narrower, less frequent communication
  - Typically, industry standard bus (ISA, PCI) for portability
- Bridge
  - Single-purpose processor converts communication between busses

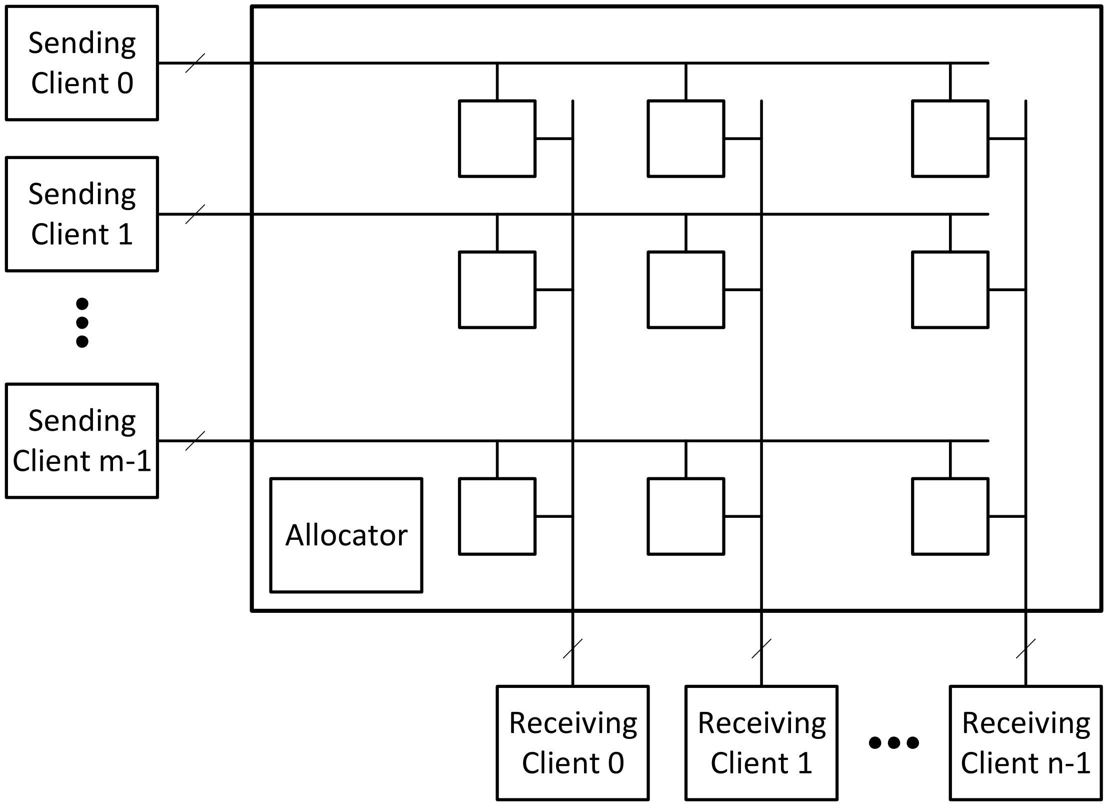


ARM’s AMBA

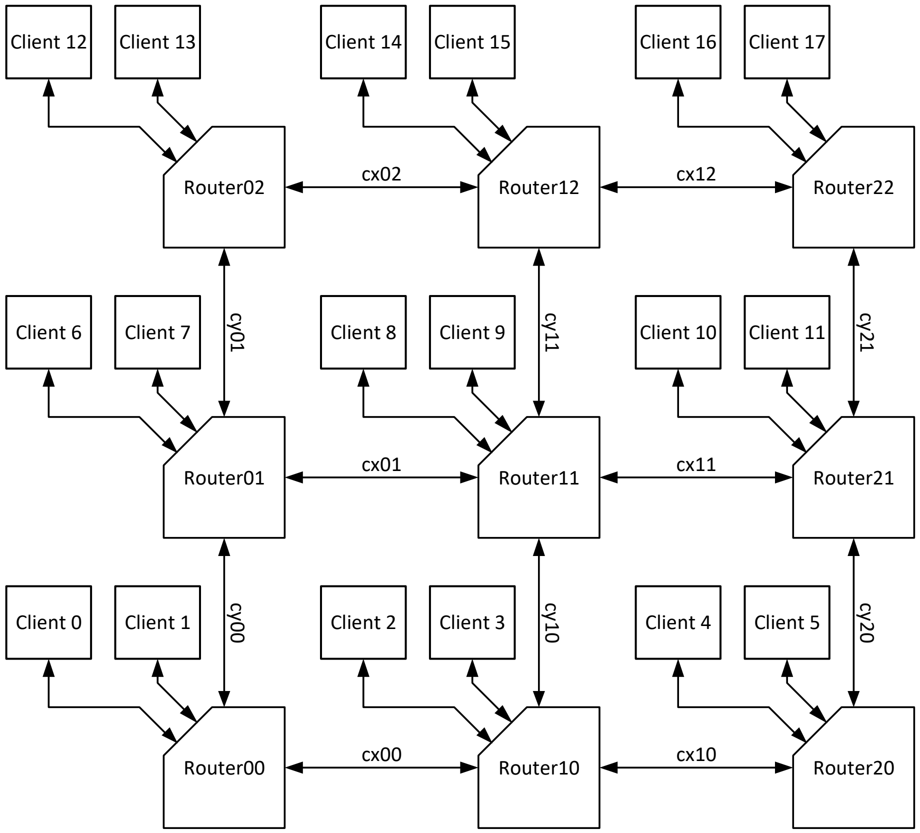
- Advanced Microcontroller Bus Architecture (Introduced by ARM in 96)
  - Open standard, on-chip interconnect specification for the connection and management of functional blocks in a System-on-Chip (SoC)
  - 32-bit addressing
  - Early SoC Architectures
    - high-performance system interconnect
      - Advanced System Bus (ASB): Version 1
      - Advance High-Speed Bus (AHB): Version 2
    - Low-speed peripheral bus:
      - Advance Peripheral Bus (APB): Version 1 & 2
    - Cross Communication via a bridge
    - 2003: 3rd generation, including AXI for connection of memory mapped components, with Advanced Trace Bus (ATB)
    - 2010: 4th generation, AMBA4, incl AXI4 2011: 5th generation, AMBA5
    - 2013: 5th generation AMBA5 with CHI (Coherent Hub Interface)



Crossbar Switch

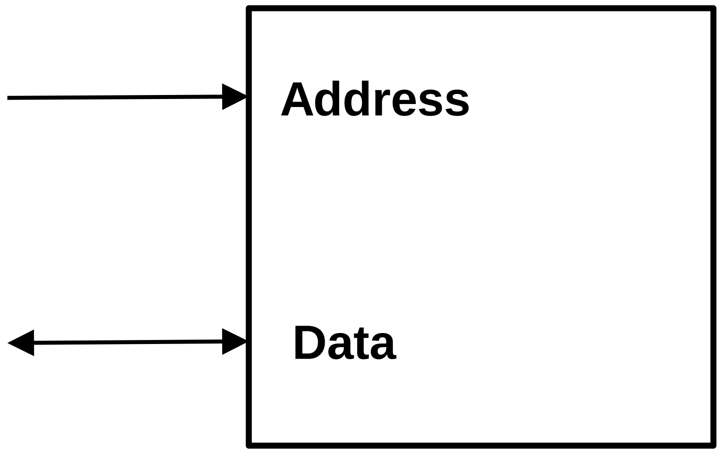


Interconnection Networks



What factors determine which interconnect solution you pick?

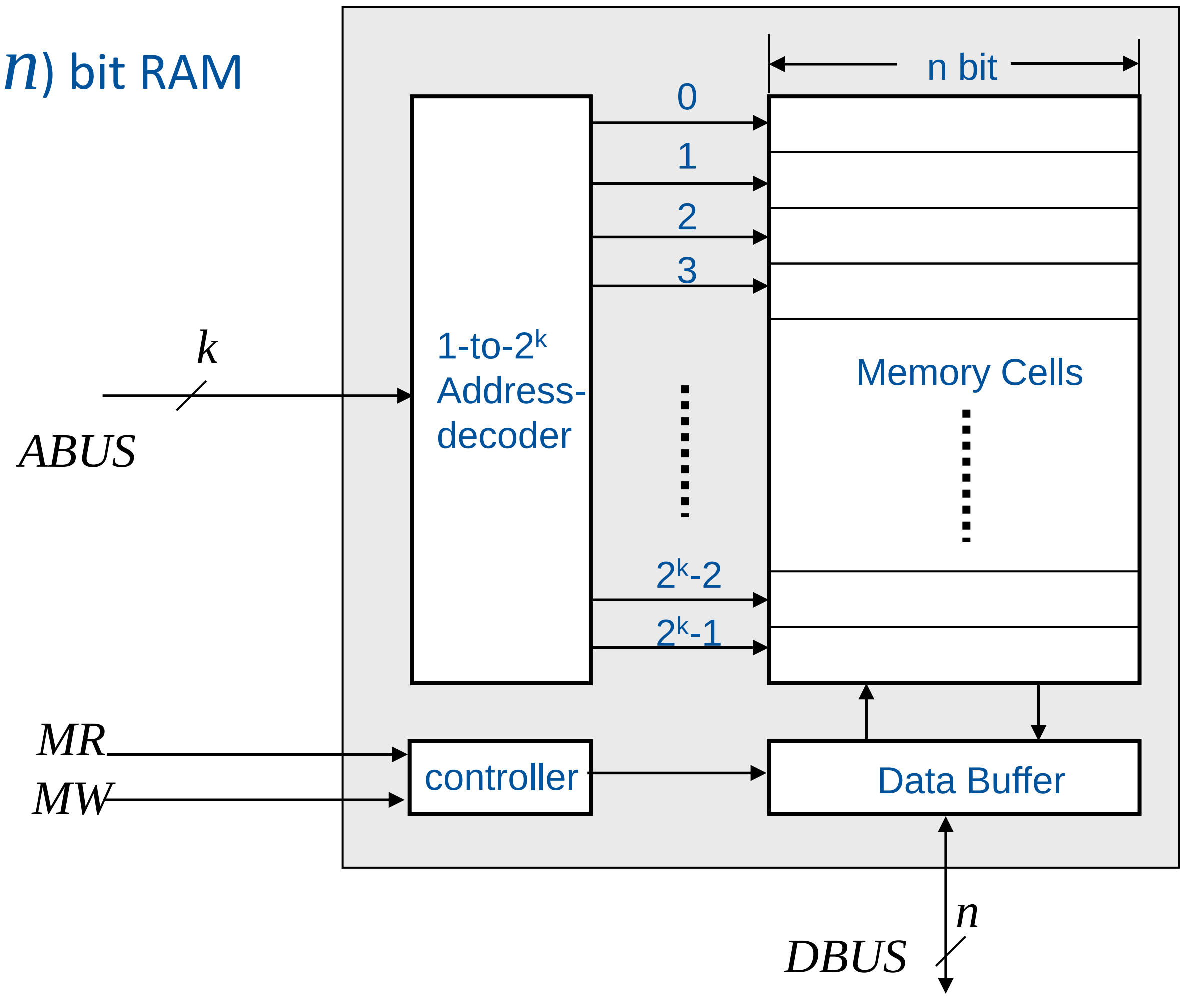
Memory



Capacity
Bandwidth
Latency
Granularity

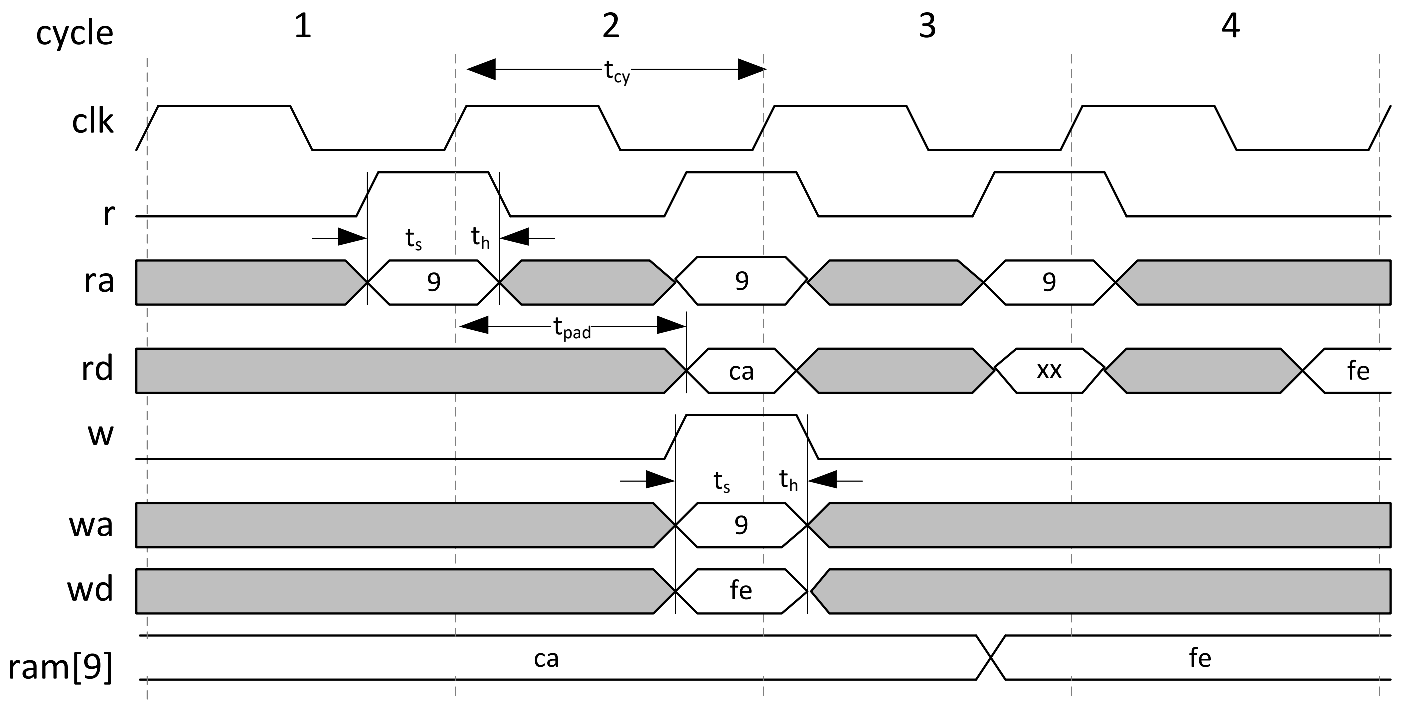
RAM

- Architecture of a $(2^k \times n)$ bit RAM

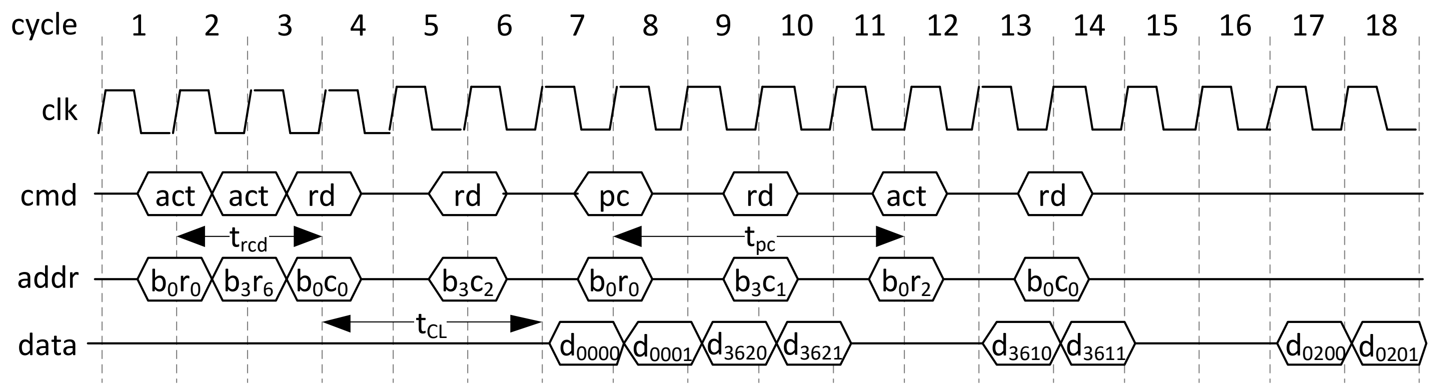


```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.Numeric_Std.all;

entity sync_ram is
  port ( clock : in std_logic;
            we : in std_logic;
       address : in std_logic_vector;
        datain : in std_logic_vector;
       dataout : out std_logic_vector );
end entity sync_ram;

architecture RTL of sync_ram is
 type ram_type is array (0 to (2**address'length)-1) of std_logic_vector(datain'range);
 signal ram : ram_type; signal read_address : std_logic_vector(address'range);

 begin RamProc: process(clock) is
    begin
      if rising_edge(clock) then
         if we = '1' then
            ram(to_integer(unsigned(address))) <= datain;
         end if;
         read_address <= address;
      end if;
 end process RamProc;

 dataout <= ram(to_integer(unsigned(read_address)));

end architecture RTL;
```

SRAM Primitive



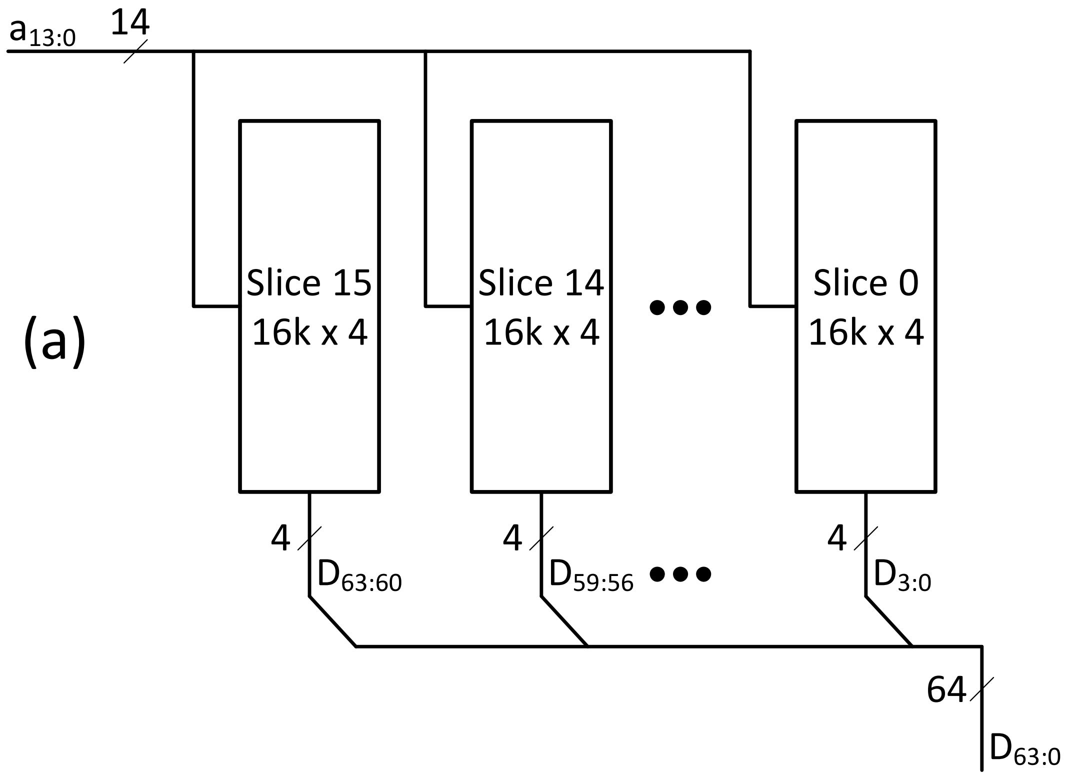
DRAM Primitive



What if you need more memory or more bandwidth than one primitive?

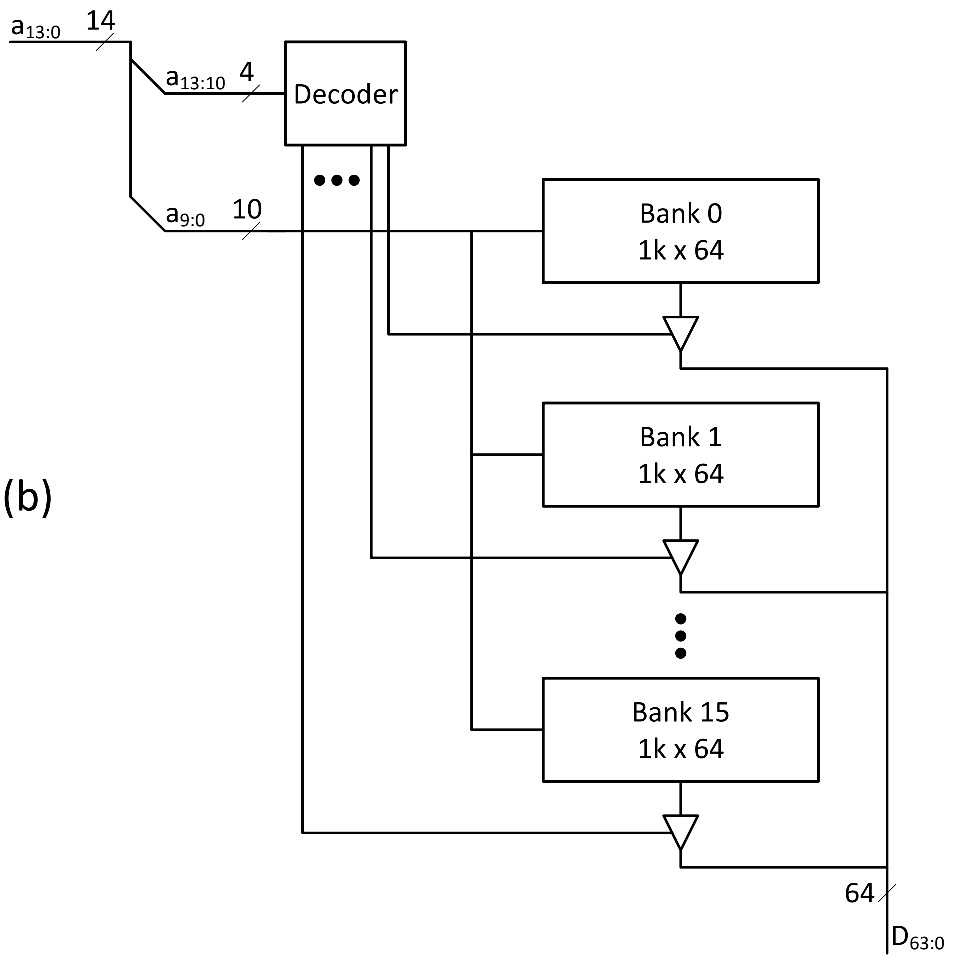
Bit-Slicing

(a)

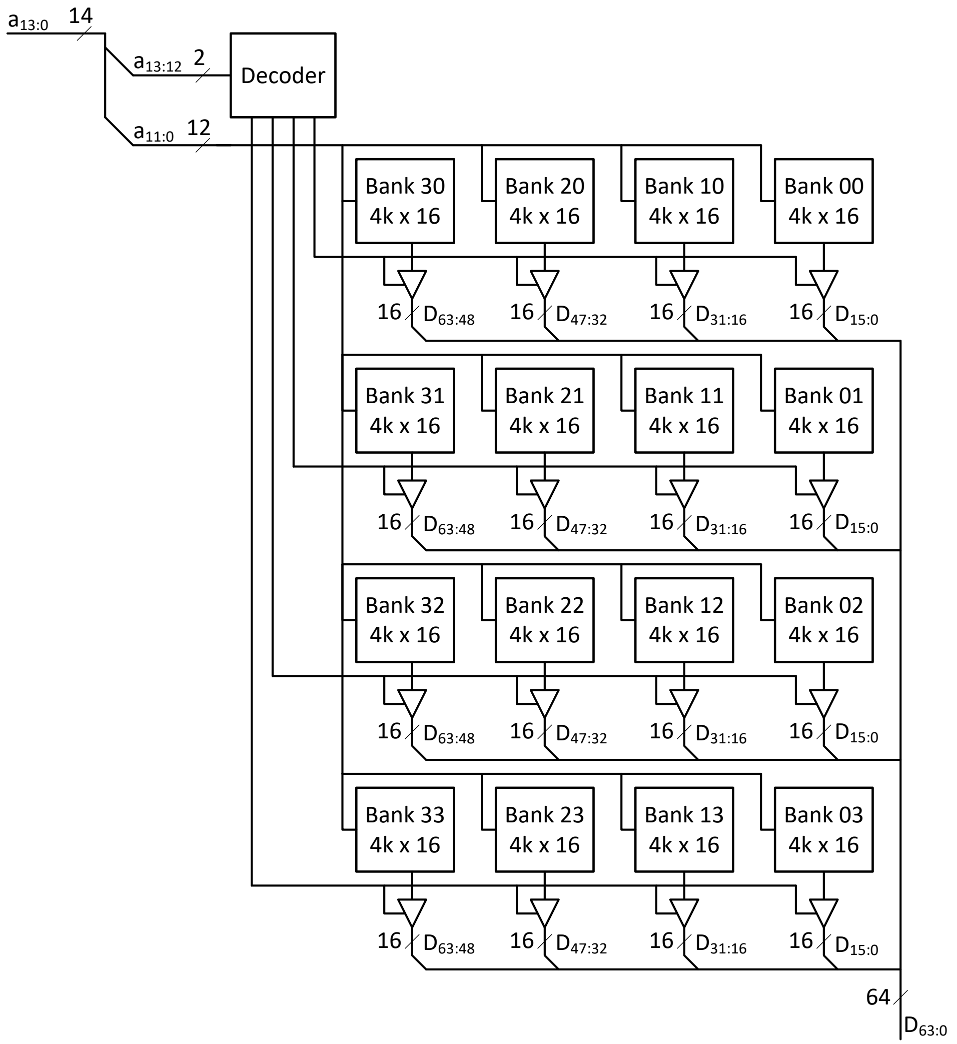


Banking

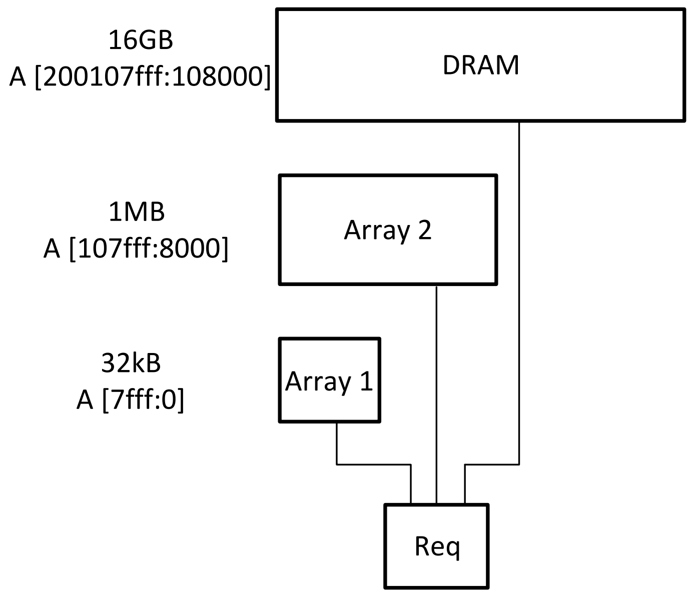
(b)


$a_{13:0}$
14
$a_{13:10}$
4
Decoder
$a_{9:0}$
10
Bank 0
1k x 64
Bank 1
1k x 64
Bank 15
1k x 64
64
$D_{63:0}$

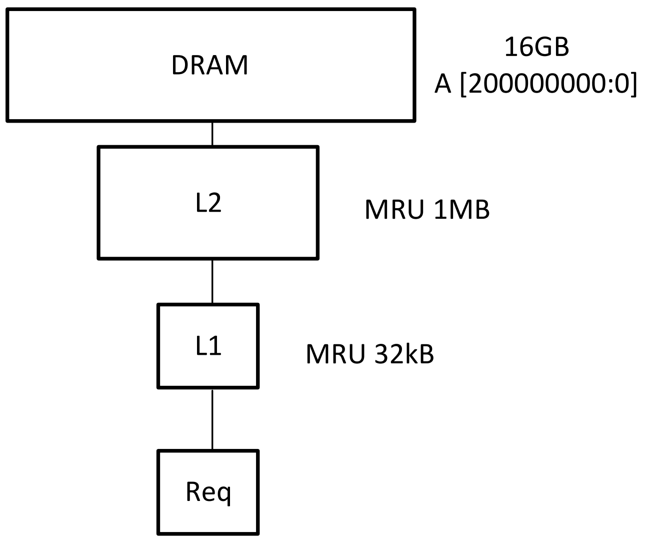
Bit slicing & banking



Hierarchy



(a)



(b)

UF Herbert Wertheim College of Engineering
UNIVERSITY *of* FLORIDA

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE