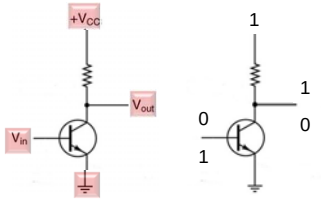
The Digital Logic Level

- lowest level – just above **transistors**
- **digital circuits**
  - collections of **gates** and wires
  - only deal with 0's and 1's
  - described by **Boolean algebra**

Transistors

- a transistor is a **switch** with
  - one input Vin
  - one output Vout
  - constant supply voltage Vcc
  - a ground



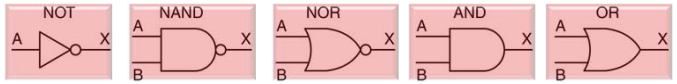
- basic **operation**
  - Vcc = 1
  - when Vin = 0, switch is closed => Vout = 1
  - when Vin = 1, switch is open => Vout = 0
- most important observation:
  a transistor **inverts** the input signal

Vin	Vout
0	1
1	0

- transistors are used to implement **gates**

Gates

- all **digital circuits** are constructed out of gates
- **gates** are:
  - the lowest level building blocks
  - described using **Truth Tables**



A	X
0	1
1	0

(a)

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

(b)

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

(c)

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

(d)

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

(e)

Boolean Algebra

- truth tables are very cumbersome
- introduce notation to capture only 1's:
  - variables: A, B, C, …
  - negated variables: A', B', C' (sometimes shown as Ā)
  - OR function: +
  - AND function: . or just adjacency
- Examples:
  - A + B = 1 if A = 1 or B = 1
  - AB = 1 if A = 1 and B = 1
  - A' = 1 if A = 0
  - AB'C = 1 if ???
- We are now ready to design some basic circuits

A Simple Digital Circuit

- Task: design a circuit to perform **majority** function
  - 3 inputs, 1 output
  - out = 1 if at least 2 inputs are 1
- Step 1: build a truth table
  - M = 1 with majority of ABC
- Step 2: write a Boolean expression
  - any truth table is a sum of products
  - each row is an AND expression (minterm)
  - combined using OR

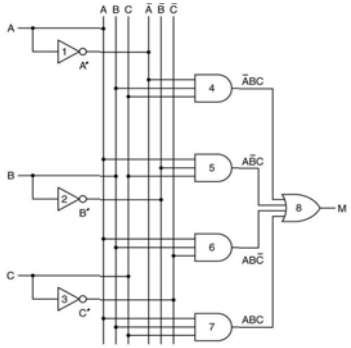
M = A'BC + AB'C + ABC' + ABC

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

A Simple Digital Circuit (cont)

M=A'BC+AB'C+ABC'+ABC

- Step 3: Lay out circuit
  - provide inverter for each negated variable
  - draw parallel lines for legibility
  - provide AND gate for each term
  - provide OR gate to combine terms
- not most efficient but it works



Circuit Optimization

- main goals
  1. reduce the total **number** of gates
  2. use only **one type** of gate (e.g. NAND or NOR)
- for goal 1:
  - apply **laws of Boolean algebra** to simplify expression
  - then lay out circuit
- for goal 2:
  - apply rules of **gate substitution**

Laws of Boolean Algebra

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

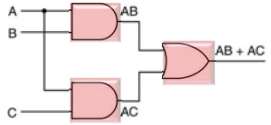
absorption law

A	B	A+B	A(A+B)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

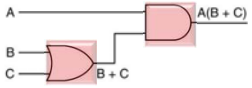
- laws 1-6 are intuitive
- distributive law: OR form is intuitive, AND form is the **dual**
- absorption law: consult truth table
- De Morgan: not intuitive but **extremely important**

Apply Laws of Boolean Algebra

- Task: implement and simplify function **M = AB + AC**
  - as given: need 2 AND gates, 1 OR gate



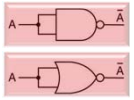
  - apply distributive law: M = AB + AC = **A(B+C)**
  - need only 1 OR gate, 1 AND gate



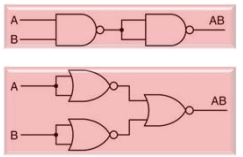
- There are systematic approaches to circuit optimization
  - subject of more advanced courses

Gate Substitution

- NOT, AND, OR can be replaced by NAND or NOR only
  - NOT A
    - using NAND
    - using NOR



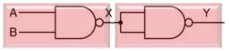
  - A AND B
    - using NAND
    - using NOR



  - analogous substitutions exist for A OR B
  - How to we know that the substitutions are equivalent?

Gate Substitution

- Task: prove that this is equivalent to (A AND B)



- Approach 1: construct truth table

A	B	X	X	Y
0	0	1	1	0
0	1	1	1	0
1	0	1	1	0
1	1	0	0	1

Y = A AND B

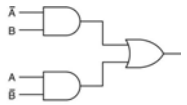
- Approach 2: use laws of Boolean algebra

$Y = (XX)' = ((AB)' (AB)')' = ((AB+AB)')' = AB+AB = AB$

from diagram | De Morgan | double NOT | idempotent

Gate Substitution

- Let's test our skills:
  - design circuit for XOR function using only NAND
  1. truth table:
  2. expression: **A XOR B = A'B + AB'**
  3. circuit:

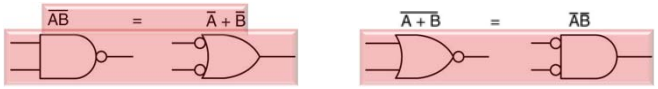


A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

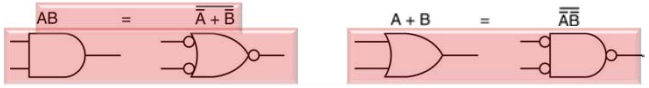
  - use NAND only:
    - replace ANDs with NAND equivalents: 2 for each
    - replace OR with NAND equivalent: 3
  - resulting circuit: 7 NAND gates!! (no way …)
    - need a better approach

Gate Substitution

- **De Morgan** suggests different substitutions:



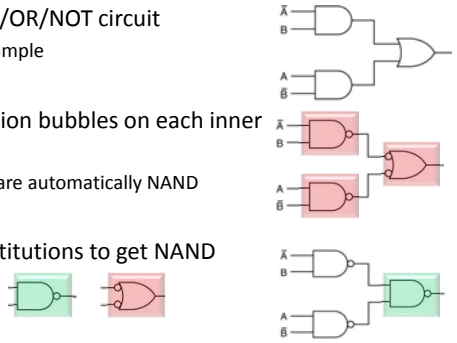
- and **negating both sides** yields yet another set:



- One last insight (trick?):
  - **two** inversion bubbles on the same line **cancel** each other
  - (A')' = A
- Now we can design with NAND (NOR) only

Gate Substitution

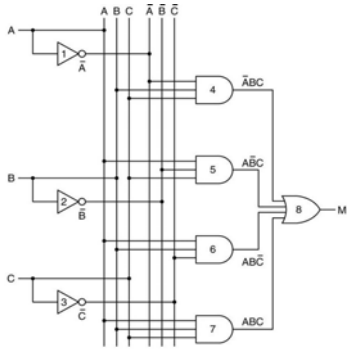
- given an AND/OR/NOT circuit
  - previous example
- place 2 inversion bubbles on each inner line
  - some gates are automatically NAND
- use gate substitutions to get NAND (NOR) only



- works for any canonical circuit:
  sum of products, product of sums

Gate Substitution

- Practice: implement majority circuit using NAND only

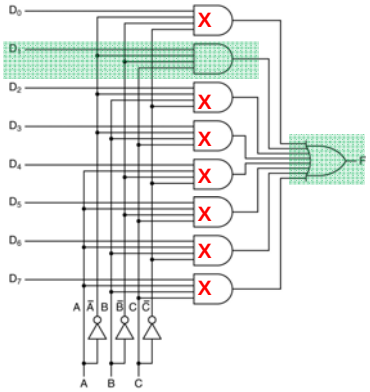


Combinational Circuits

- multiple inputs and **multiple** outputs
- output depends **only** on current inputs (no memory)
- **general purpose** circuits
  - multiplexers/demultiplexers, encoder/decoders, comparators
- **arithmetic** circuits
  - shifters, adders
- used to construct more complex circuits
  - ALUs

Multiplexers

- interface
  - 2^n inputs
  - 1 output
  - n control lines
- basic idea:
  - select one of the inputs D_i to be the output F using ABC
    - 000 selects D_0
    - 001 selects D_1 .....
    - 111 selects D_7
- how does it work?
  - A=0, B=0, C=1

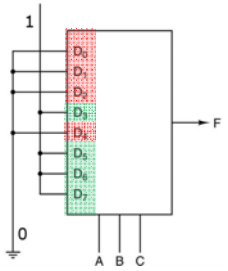


- demultiplexer: connect 1 input to one of 2n outputs

Using a Multiplexer

- implement **majority** function using **multiplexor chip**

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

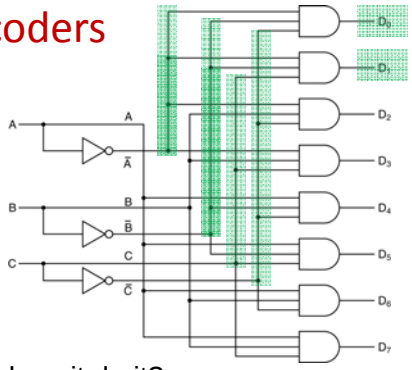


- each line in the table corresponds to one input
  - connect 0 or 1 to inputs based on table
  - ABC chooses the correct output
- Examples: ABC=011, ABC=100

Decoders

- interface
  - n inputs
  - 2^n outputs
- basic idea:
  - turn a number into a position

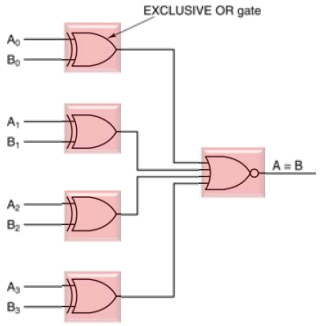
A	B	C	
0	0	0	**D0**
0	0	1	**D1**
0	1	0	**D2**
0	1	1	**D3**
1	0	0	**D4**
1	0	1	**D5**
1	1	0	**D6**
1	1	1	**D7**



- how does it do it?
  - ABC=000, D_0 needs 111: connect A'B'C'
  - ABC=001, D_1 needs 111: connect A'B'C
  - ABC=111, D_7 needs 111: connect ABC

Comparators

- interface
  - input: 2 n-bit numbers A, B
  - 1 output
- basic idea:
  - output=1 if A=B
- how does it do it?
  - use XOR gates
    - 1 if A_i or B_i is 1, i.e. **NOT** equal
    - 0 if equal
  - A=B iff **all** lines are 0
  - OR is 0 if all 0's, NOR is 1



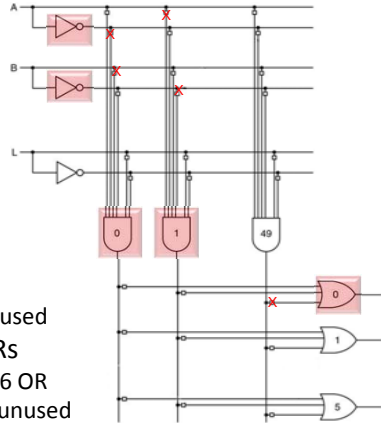
- recall how to implement XOR gates (3 NAND or 3 NOR)

PLAs

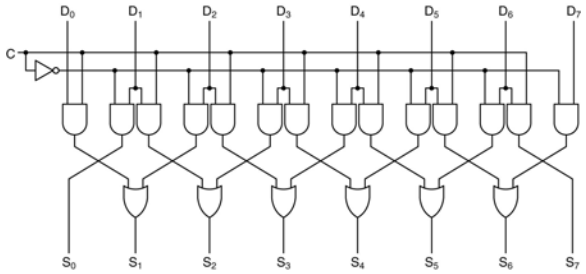
- Programmable Logic Arrays
  - no specific function is predefined
  - superset of all possible canonical functions
    - sum of products
  - chosen function is programmed once – "burnt in"
    - analogous to a CD-R
  - how does it work?
    - every internal variable has both options, X and X'
    - every line has a fuse
    - one of the two fuses is burnt out to chose X or X'

PLAs

- Example
  - 12 inputs
  - 6 outputs
- implement XOR
  - F = AB'+A'B
  - 2 AND, 1 OR, A', B'
  - burn fuses
    - AB'
    - A'B
    - OR
  - all other gates are unused
- can implement 6 XORs
  - 12 in, 6 out, 12 AND, 6 OR
  - 50-12=38 AND gates unused

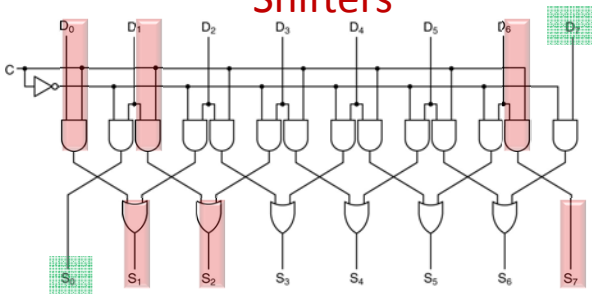


Shifters



- interface
  - n inputs
  - n outputs
  - 1 control line
- basic idea:
  - if C=0, shift all inputs left by one bit
  - if C=1, shift right

Shifters

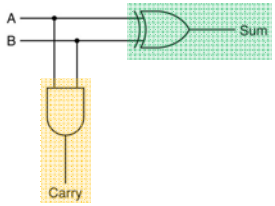


- how does it work?
  - each D_i is connected to 2 AND gates
  - C determines which gate is opened
    - E.g., C=1 then every D_i goes to S_{i+1}
  - Note: shift is not circular, outer-most bit is lost
    - E.g., C=1 then S_0=0 and D_7 is lost

Adders

- A Half Adder
  - adding 2 bits produces a sum bit and a carry bit

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

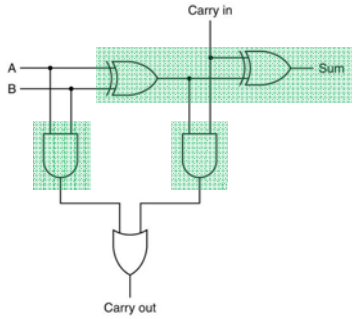


  - carry propagates to the next higher-order bit
  - to add multi-bit numbers need circuit with input carry

Adders

- A Full Adder

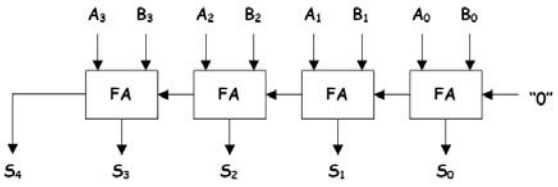
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



  - Sum=1 if odd number of A, B, C_{in} are 1 (2 XOR gates)
  - C_{out}=1 if A, B, C_{in} add up to more than 1, i.e.:
    - when AB=11, regardless of C_{in} (first AND gate), or
    - one of A or B is 1 (given by first XOR) and C_{in}=1 (second AND)

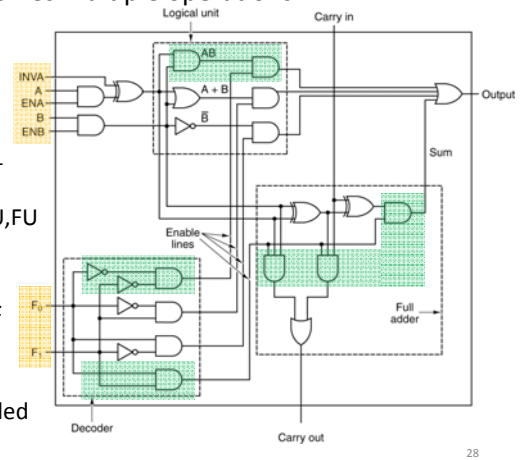
Adders

- A multi-bit adder simply replicates n Full Adders
  - a Ripple Carry Adder



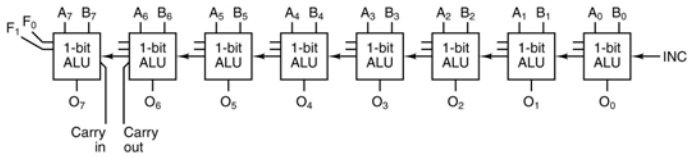
Arithmetic Logic Units

- An ALU combines multiple operations
- 1-bit ALU
- inputs:
  - A, B, F, Cin
- assume
  - ENA=ENB=1
  - INVA=0
  - A,B go to LU,FU
- Logical Unit:
  - AB, A+B, B'
- selected by F
  - e.g. F=00
- with F=11:
  - adder enabled



Arithmetic Logic Units

- n-bit ALU is constructed from n 1-bit ALUs
- each 1-bit ALU is a "bit slice"



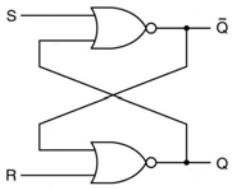
- depending on F, performs:
  - AB, A+B, B', A+B on 8-bit numbers A and B
  - with INC=1 can increment, e.g. A+1, B+1, A+B+1

Memory

- **combinational** circuits – discussed thus far
  - output depends **only on current** inputs
- **memory** circuits
  - output depends on **current and past** inputs
- memory is essential for any computer
  - holds data and instructions while manipulated by combinational circuits
- memory types:
  - latches, flip-flops, registers, RAM

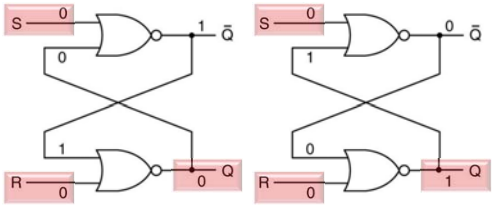
Latches

- SR latch: simplest 1-bit memory circuit
- 2 inputs (S,R), 2 outputs (Q, Q')
- outputs depend on inputs **and** outputs
  - Q and Q' are complements



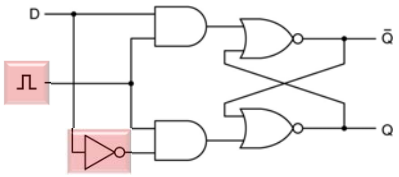
Latches

- basic idea:
  - 2 stable states
    - with S=R=0, Q can be 0 or 1
  - S=1 sets the latch (Q=1)
    - no change if already set
  - R=1 resets (Q=0)
    - no change if already reset



Latches

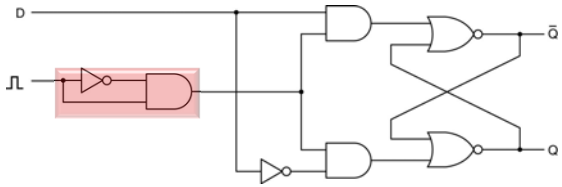
- what happens when S=R=1?
  - Q=Q'=0
  - but when S=R=0 again, new state is **nondeterministic**
  - solution: D latch



- Clock signal
  - permits set/reset only during a specific interval (strobe)

Flip-Flops

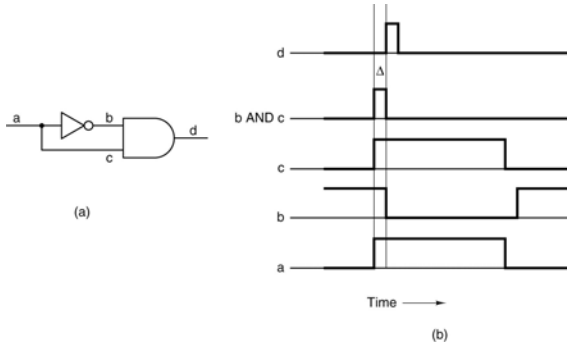
- what happens when D keeps changing while clock=1?
- better: capture D at a precise instant of time
  - need a circuit to generate a short pulse
  - flip-flop captures D at that moment:
    flip-flop: **edge**-triggered vs. latch: **level**-triggered



  - how does the pulse-generating circuit work?

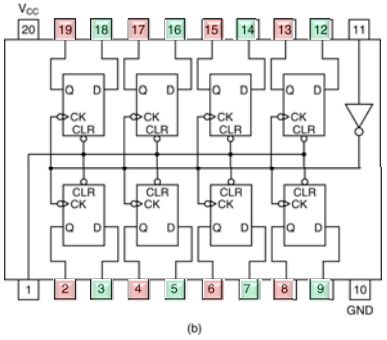
Flip-Flops

- pulse-generating circuit



Registers

- combine flip-flops on a chip:
- 8-bit register
- outputs:
  - Q_i=2,4,6,8,13,15,17,19
  - an 8-bit number currently stored
- inputs:
  - D_i=3,5,7,9,12,14,16,18
  - clock (CK)=pin 11
  - clear (CLR)=pin 1

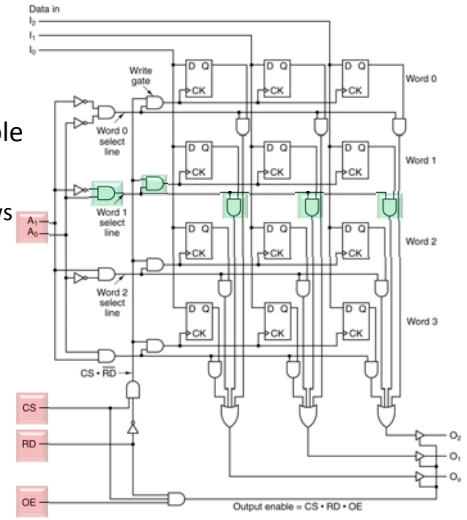


RAM Memory

- RAM: random access memory
  - a block of n-bit **words** (rows of n bits)
  - need **address** to select word
  - can **read or write**
- interface
  - n input bits I_i (one n-bit word to be written)
  - n output bits O_i (one n-bit word to be read)
  - k address lines (to select among 2^k words)
  - RD/WR control
  - additional chip controls

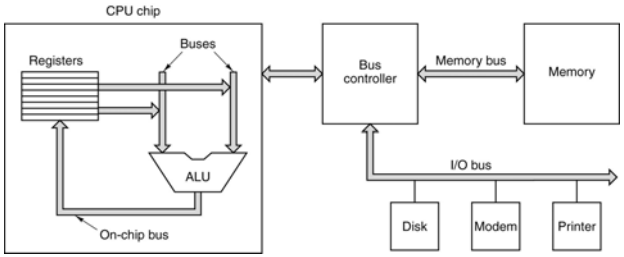
4x3 RAM

- CS: chip select
- OE: output enable
- A: 2-bit address:
  decoded to 4 rows
  Ex: A=01
- RD=1: read
  Q_i -> O_i
- RD=0: write
  I_i -> D_i



Computer Buses

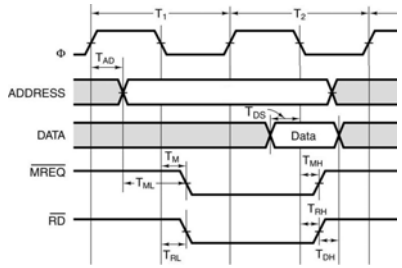
- a bus is a **pathway** (parallel wires) between devices
  - can be **internal** to CPU: ALU to registers
  - external to CPU: connect to memory or I/O devices
- **master**: initiate transfer
- **slave**: respond to request



Bus Clocking

- **synchronous** bus
  - driven by clock
  - square wave with frequency 5 MHz – 100 MHz
  - 1 MHz = 1 million cycles/sec
  - Example: 100 MHz → 10^8 cycles/10^9 ns → 1 cycle = 10 ns
- assume a **memory** bus and a **read** operation
  - CPU places memory address on address lines
  - CPU commands memory to read
  - memory places content on data lines
  - CPU copies data and informs memory

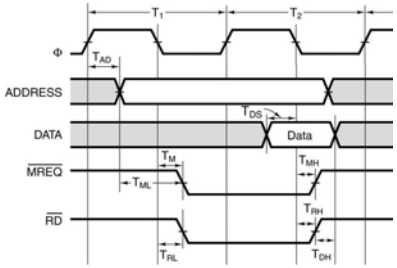
Bus Clocking



operations need to obey mutual **timing constrains**

- T_{AD}: max time for CPU to output address
- T_{ML}: min time for CPU to wait before MREQ (mem spec)
- T_M/T_{RL} : max time for CPU to request operation
- T_{DS}: min time for memory to output data
- T_{MH}/T_{RH} : max time for CPU to signal that it copied data

Bus Clocking



Q: how much time does memory have to produce data?

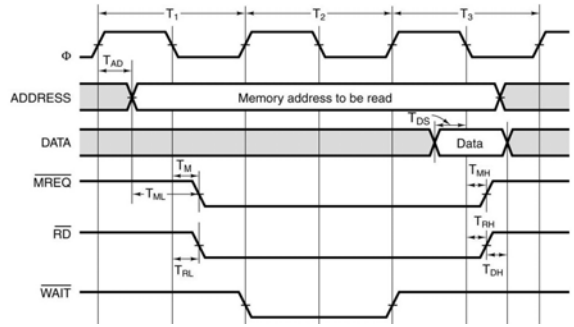
A1: $15–T_{AD}–T_{DS} = 15–4–2 = 9$ ns

A2: $10–T_M–T_{DS} = 10–3–2 = 5$ ns

- A1 is NOT subsumed by A2 because T_{ML} could be negative
- i.e., MREQ and RD could be asserted before address is stable
- However, T_{ML} must be positive for **write** operation (address must be stable before operation is invoked)

Bus Clocking

- now assume memory needs 13 ns to produce data
- memory inserts **wait cycle**
- now it has up to 15 ns to produce data (using constraint A2)

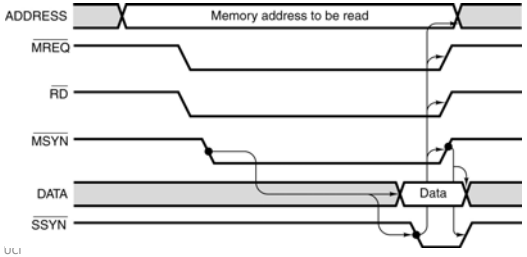


Bus Clocking

- drawbacks of synchronous buses
  - all operations must be in **multiples** of clock cycles
  - see previous example: memory has 15ns; if it needs 16ns, entire read cycle expands by 10ns (additional clock period)
  - cannot take advantage of future faster memory
- **asynchronous** bus
  - no master clock
  - bus cycle can be any length and can vary for any device
  - CPU and device use **handshake** protocol to request operation and signal availability of data

Bus Clocking

- CPU places memory address on address lines
- CPU commands memory to read (MREQ, RD)
- CPU asserts MSYN (master synchronization signal)
- memory places content on data lines asap
- memory asserts SSYN (slave synchronization signal)
- CPU copies data and negates MSYN (and all other lines)
- memory negates SSYN (completing the cycle)



Block Transfers

- reading/writing one word at a time is inefficient (needs several clock cycles)
- when an entire block is needed (e.g. when caching) the bus may support block transfer operations