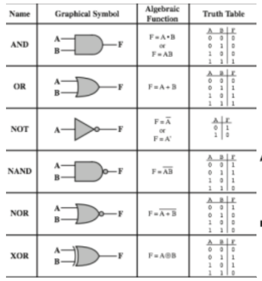
Chapter 3: Digital Logic



Name	Graphical Symbol	Algebraic Function	Truth Table
AND		F = A • B or F = AB	A B F: 0 0 0; 0 1 0; 1 0 0; 1 1 1
OR		F = A + B	A B F: 0 0 0; 0 1 1; 1 0 1; 1 1 1
NOT		$F = \overline{A}$ or F = A'	A F: 0 1; 1 0
NAND		$F = \overline{AB}$	A B F: 0 0 1; 0 1 1; 1 0 1; 1 1 0
NOR		$F = \overline{A + B}$	A B F: 0 0 1; 0 1 0; 1 0 0; 1 1 0
XOR		$F = A \oplus B$	A B F: 0 0 0; 0 1 1; 1 0 1; 1 1 0

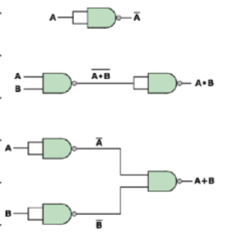


Figure 11.2 Some Uses of NAND Gates

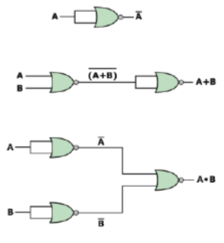


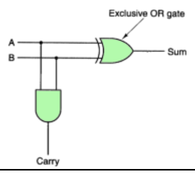
Figure 11.3 Some Uses of NOR Gates

Characteristic table

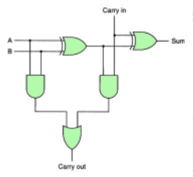
S	R	Q_{next}	Action
0	0	Q	hold state
0	1	0	reset
1	0	1	set
1	1	X	not allowed

```
      LD    P0,R4     0000:  0600ff04    0000003c
      LD    P1,R1     0008:  0600ff01    00000040
      MOV   R1,R2     0010:  05010002
      LD    P2,R3     0014:  0600ff03    00000044
L:    ADD   R4,R1,R4  001C:  00040104
      ADD   R1,R2,R1  0020:  00010201
      SUB   R3,R1,R5  0024:  01030105
      BNZ   L         0028:  0802ff00    0000001c
      ST    R4,P      0030:  0704ff00    00000048
      HLT             0038:  09000000
P0:   .WORD 0         003C:  00000000
P1:   .WORD 1         0040:  00000001
P2:   .WORD A         0044:  0000000a
P:    .WORD           0048:  00000000
```

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



A	B	Carry in	Sum	Carry 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



Chapter 4: Number Representations

Two's Complement: **Range: -2^{n-1} through $2^{n-1} - 1$, # of representations of 0:** One**, Overflow Rule:** If two numbers with the same sign are added, then overflow occurs iff the result has the opposite sign.

Multiplication example: 10011 (-13) x 01011 (11) | 1111110011 + 1111100110 + 1110011000 | 1101110001 (-143)

Division example: 01100100 (100) / 00011001 (25) → Both signs positive, quotient is positive, keep subtract the divisor (take two's complement of 25, until the partial remainder is zero/negative, add one to quotient for each subtraction)

Floating point conversion example: Bit pattern for 7.375: convert 7.375 to binary form, getting 111.011 → normalize the value: 111.011 = 1.11011 x 2^2 → significand = 1101 1000 0000 0000 0000 0000, sign-bit = 0, exponent = 2+1023 (excess-1023) = 1025 = 100 0000 0001 → bit pattern = 0100 0000 0001 1101 1000 0000 0000 0000 0000 = 401d8000

Multiplication of Floating point numbers: $exp_1 = e_1 - b$, $exp_2 = e_2 - b$, $exp_3 = exp_1 + exp_2 = e_1 + e_2 - b - b = e_3 - b$; Hence $e_3 = e_1 + e_2 - b$ → Normalize result: $(+/-)\ m_1 \times 2^{exp1} \times (+/-) m_2 \times 2^{exp2} = (+/-)\ m_1 \times m_2 \times 2^{exp1+exp2}$

Division of floating point numbers: $(+/-)\ m_1 / m_2 \times 2^{exp1+exp2}$

n-bit adders: **Ripple adder** (carry out from the previous bit is fed to carry in of the next bit, need to wait for carry propagate through the adders from left to right, so is slow), **Carry look-ahead adder** (calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of larger value bits of the adder), **Carry-Save adder** (outputs of two numbers of the same dimensions as the inputs, one which is a sequence of partial sum bits and another which is a sequence of carry bits)

Chapter 5: Instruction Execution Cycle

PC: Stores the address of the instruction to be executed; IR: Contains the current instruction; MAR: holds the address for the memory, it is directly connected to the external address bus to the memory; MBR: Contains the data returned from the external memory, or the data to be written to the external memory; IOAR, IOBR (I/O version of MAR and MDR)

Interrupt: Original instruction execution cycle is an infinite loop, there is no way to escape from program execution, other program may need CPU attention, when I/O arrives, it may need immediate attention of the CPU, otherwise there will be data loss. I/O devices will generate an interrupt signal to the CPU, telling it that it wants attention. When processing an interrupt, the system need to remember where the original program was executing, then serve the interrupt, and return to the exact location to continue execution.

Chapter 6: Memory

DRAM: Made with cells that store data as charge on capacitors, presence or absence of charge in a capacitor is interpreted as 1 or 0, requires periodic charge refreshing to maintain data storage; SRAM: uses the same logic elements used in the processor, binary value are stored using traditional flip-flop logic gate configurations, has power will hold the data; ROM, PROM, EPROM, EEPROM, Flash Memory

Location: Internal memory = main memory, external memory = peripheral storage; Capacity: usually byte addressable

Access Method: Sequential access, Direct access, Random Access, Associative

Principle of locality: Memory reference tends to be localized, data in the vicinity of a referenced word are likely to be referenced in the near future, code reference is local, data reference is less local, but it is still so, when data is referenced, it will tend to be referenced again in the near future, future access can be from higher level (cache)

Chapter 7: Cache Memory

Cache is introduced to bridge the speed gap between CPU and Primary memory (SRAM is used instead of DRAM)

Cache Read Operation: The processor generates a read address of a word to be read (if the word is contained in the cache, it is delivered to the processor, the data and address buffers are disabled, and communication is only between processor and cache, with no system bus traffic)

Cache Size: **Small enough** so that overall average cost per bit is close to that of main memory alone, **Not too large** so that the overall average access time is close to that of the cache alone

Scenario:
Number of blocks in main memory = 2S
Block size = line size = 2W words or bytes
Address length = (s+w) bits
Number of addressable units = 2S+W words or bytes

Direct Mapping:
Number of lines in cache = m = 2r
Size of tag = (s-r) bits
Associate Mapping:
Size of tag = s bits

Mapping: **Size of tag** = s bits; Set Associative Mapping: **Address length** = (s+w) bits, **# of addressable units** = 2^{s+w}, **Block size** = 2^w words, **# of blocks in main memory** = 2^s, **# of lines** = k, **# of sets** = 2^d, **# of lines in cache** = k x 2^d, Size of cache = k x 2^{d+w}, Size of tag = (s-d) bits

Replacement algorithm: **Least Recently used (LRU)** → replace that block in the set that has been in the cache longest with no reference to it, **FIFO** → replace that block in the set that has been in the cache longest (round robin fashion), **LFU** → Replace the one with fewest references

Write policy: **Write through** (all write operations are made to main memory as well as to the cache, simple and maintain consistency, but generates substantial memory traffic and bottleneck), **Write back** (only update in cache, use of dirty bit to indicate the replacement and write back to main memory

Average Access time= Cache hit time+(1-hit rate) *cache miss penalty

Chapter 8: External Storage

Hard disk performance parameters:

Avg rotation latency = 0.5 x time to complete 1 rotation

Time to read 10 consecutive tracks

= avg seek time + time to move to adjacent track x 10 + avg rot latency x 10 + time for 1 rotation x 10

Disk Access Time = Seek Time + Rotational Delay

RAID: **Redundant Array of Independent Disks (Consist of 7 levels)**, set of physical disk drives viewed by the operating system as a single logical drive, data are distributed across the physical drives of an array in a scheme known as striping, redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a dis

Level	Advantages	Disadvantages	Applications
0	I/O performance is greatly improved by spreading the I/O load across many channels and drives No parity calculation overhead is involved Very simple design Easy to implement	The failure of just one drive will result in all data in an array being lost	Video production and Editing Image editing Pre-press applications Any application requiring high bandwidth
1	100% redundancy of data means no rebuild is necessary in case of a disk failure, just a copy to the replacement disk Under certain circumstances, RAID 1 can sustain multiple simultaneous drive failures Simplest RAID storage subsystem design	Highest disk overhead of all RAID types (100%) - inefficient	Accounting Payroll Financial Any application requiring very high availability
2	Extremely high data transfer rates possible The higher the data transfer rate required, the better the ratio of data disks to ECC disks Relatively simple controller design compared to RAID levels 3,4 & 5	Very high ratio of ECC disks to data disks with smaller word sizes - inefficient Entry level cost very high - requires very high transfer rate requirement to justify	No commercial implementations exist / not commercially viable
3	Very high read data transfer rate Very high write data transfer rate Disk failure has an insignificant impact on throughput Low ratio of ECC (parity) disks to data disks means high efficiency	Transaction rate equal to that of a single disk drive at best (if spindles are synchronized) Controller design is fairly complex	Video production and live streaming Image editing Video editing Prepress applications Any application requiring high throughput
4	Very high Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency	Quite complex controller design Worst write transaction rate and Write aggregate transfer rate Difficult and inefficient data rebuild in the event of disk failure	No commercial implementations exist / not commercially viable
5	Highest Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency Good aggregate transfer rate	Most complex controller design Difficult to rebuild in the event of a disk failure (as compared to RAID level 1)	File and application servers Database servers Web, e-mail, and news servers Intranet servers Most versatile RAID level
6	Provides for an extremely high data fault tolerance and can sustain multiple simultaneous drive failures	More complex controller design Controller overhead to compute parity addresses is extremely high	Perfect solution for mission critical applications

SSD: **High performance I/O per second, Durability (less susceptible to physical vibration), Longer lifespan, Lower power consumption, Quieter and Cooler, Lower access time**

Chapter 9: I/O:

External Devices has three categories: Human readable, Machine Readable, Communication.
Major functions for an I/O module: **Control and timing** (Coordinates the flow of traffic between internal and external devices), **Processor communication** (involves command decoding, data, status reporting, address recognition), **Device communication** (involves commands, status information, and data), **data buffering** (performs the needed operation to balance device and memory speeds), **error detection** (detects and reports transmission errors)
Device with large speed variation: Thus, asynchronous communication is deployed rather than synchronous communication
I/O Techniques: **Programmed I/O** (Data are exchanged between processor and I/O module, processor directly controls the I/O operation, when processor issues a command it **must** wait until the I/O operation is completed, waste processor time if processor is faster than I/O operation), **Interrupt-driven I/O** (processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work.), **Direct Memory Access (DMA)** (The I/O module and main memory exchange data directly without CPU involvement, steals bus cycles) → has interrupt
I/O Commands: **Control, Test, Read, Write**
I/O Mapping: **Memory mapped I/O** (Devices and memory share an address space, I/O just like memory read/write, No special commands for I/O), **Isolated I/O** (separate address space, need I/O or memory select line, special commands for I/O)
Interrupt Processing: **INTA (Interrupt Acknowledgement), RETI (Return from Interrupt Instruction)**
Device Identification: **Multiple interrupt lines** (between the processor and the I/O module, straightforward, during processor design different priorities can be assigned to different interrupt lines),
Software poll (when processor detects an interrupt it branches to an interrupt-service routine whose jobs is to poll each I/O module to determine which module caused the interrupt, time consuming),
Daisy Chain (hardware poll, vectored): All I/O module shared a common interrupt request lines,
Bus Arbitration (an I/O module must first gain control of the bus before it can raise the interrupt request line)

Chapter 10: Instruction Sets

Instruction Types: **Data Processing, Data Storage, Control, Data Movement**
Arithmetic Operation: Treat the operands as numbers, consider sign of operand, shift operations are done
Logic Operation: Treat the operands as bit patterns
Transfer of Control: It is essential to be able to execute each instruction more than once (Branch, Skip, Procedure)
Procedure Call Instructions: economy and modularity, stack implementation (LIFO)
# of operand: OP A,B,C (A <- B OP C); OP A,B (A <- A OP B); OP A (AC <- AC OP A); OP (T <- (T-1) OP T)
Registers: **General Purpose Registers** (Registers can be used freely), **Special purpose registers** (Registers have dedicated purpose, for example, to point to strings, as operands for floating point instructions, e.g. PC, SP, PSW)
Addressing Mode: **Immediate (**Operand = A), **Direct** (EA = A), **Indirect** (EA = (A)), **Register** (EA = R), **Register indirect** (EA = (R)),, **Displacement** (EA = A+(R)), **Stack** (EA = T of stack)

```
.data                   # data segment
a:    .word 1           # create storage containing 1
      .word 3           # create storage containing 3
x:    .word 4           # create storage containing 4
.text                   # program segment
main:
      ld #a, r8         # r8 = address of a (#a)
      ld 0(r8),r9       # r9 = a[0], i.e. 1
      ld 4(r8),r10      # r10= a[1], i.e. 3
      bgt r9,r10,f1     # if r9>r10, goto f1
      st r10,x          # x=r10
      br f2             # goto f2
f1:   st r9,x           # x=r9
f2:   ret               # return to OS (same as return in C++)
```

Instruction Fetch

```
FI:  MAR  ←  PC
     PC   ←  PC+4
     IR   ←  mem[MAR]
```

Instruction Execute

```
CO1:          MAR  ←  PC         # find operand addr
              PC   ←  PC+4
              MBR  ←  mem[MAR]   # addr of A in MBR
              MAR  ←  MBR
FO1:          MBR  ←  mem[MAR]   # value of A in MBR
       ALU.input1  ←  MBR
```

Chapter 11: Assembly Language Programming

1. Consider a Serial Interface (e.g. Modem), containing a *Control & Status Register* and two *Buffer Registers*, Input and Output Buffer Register, residing in memory location SCSR, SBRI and SBRO,

   The SCSR has the following format:

```
Bit 0      =1 if Device Error
Bit 1      =1 if Device Ready
Bit 2      =0 if next operation is Write, 1, Read
Bit 3-5    =000 if speed = 4800 bps
           =001 if speed = 9600 bps
           =010 if speed = 19200 bps
           =011 if speed = 57600 bps
           =100 if speed = [illegible] bps
Bit 6      =0 if odd parity, 1 if even parity
```

```
        SUB  R1, R1, R1     ; Initialize index to 0(first char)
Ready:  LD   SCSR, R2       ; Wait for device ready
        AND  R2, #0x2, R3   ;   - check bit 1
        BZ   Ready          ;   - device ready if bit 1 = 1
Write:  LD   LINE(R1), R6   ; Write LINE(R1) to buffer
        ST   R6, SBRO       ;
        MOV  #0x60, R7      ; Initiate write by setting SCSR
        ST   R7, SCSR       ;
Wait:   LD   SCSR, R2       ; Wait for device ready again
        AND  R2, #0x1, R3   ;   - check bit 0 for error
        BNZ  Write          ;   - resend if error
        AND  R2, #0x2, R3   ;   - check bit 1
        BZ   Wait           ;   - device ready if bit 1 = 1
        ADD  R1, #0x4, R1   ; Move to next word (4 byte)
        SUB  R1, #0x28, R3  ; end if index ≠0 (10 words)
        BNZ  Write          ;
        HLT                 ;
```

Chapter 12: Operating System Support

Services provided by the OS: **Program Creation** (in the form of utility programs that are not actually part of the OS, but are accessible through the OS, **Program execution** (read program from hard disk to main memory, prepare resources), **I/O access** (provide a uniform I/O interface to the programs, while the details is left to the OS, handle the sharing of devices to prevent users form holding the device indefinitely), **File system management, system access, error detection and response, accounting**
Protection Scheme: **Hierarchy of privileges** (ring-based security, hierarch of privileges resembles a set of concentric rings, with the kernel mode in the centre) → need to change from user mode to kernel mode to run protected resources
Multitasking and time-sharing: **CPU idle when I/O is performed, CPU is shared among processes. Each process is allocated a time slice, when process uses up its time slice, it will stop execution, and let the next process to be executed.**
Process scheduling: the system maintains a queue of processes
Virtual memory: **logical addressing space**, not physical addressing space, mapping done by memory management unit (MMU), transparent to user, Paging is used in most virtual memory system.
Virtual memory is a memory management capability of an OS that users hardware and software to allow a computer to compensate for physical memory shortages by temporarily transferring data from random access memory to disk storage. Virtual address space is increased using active memory in RAM and inactive memory in portion of hard dis drives to form contiguous addresses that hold both the application and its data.
Paging: 32-bit address = 4GB addressing space
Page Table Entries: Valid bit, V (if the page is in memory or not) → If yes, where is it (physical frame number), Protection information, P (indicating the protection mode, usually more than 1 bit, such as r/w and supervisor/user accesses), Dirty Bit, D (if the page has been changed, to prevent unnecessarily write back when the page has not been changed), **The page table is cached in the Translation Lookaside Buffer (TLB)**
Demand Paging: When program starts, nothing is in memory, only bring in pages on demand, **Advantages:** Fast response, because we do not need to bring in the entire program to start, **Disadvantage:** a lot of page faults, until a stable working set of the program has been brought into the memory, use pre-paging to bring in a number of pages at the beginning to minimize this.

Chapter 13: Processor Organization & Reduced Instruction Sets Computer

Registers: User visible registers (GPR, Index registers, base registers), Hidden Registers (PC, IR, MAR, MBR, TEMP, PSW)
Ways to generate control signals: **Hardwired control** (control signals are generated by the logic gates, faster but the design is more complicated), **Microprogrammed control** (control signals are stored in memory – micro codes, entire truth table is stored in memory and the inputs are the address of memory, even though the control signal is inside the CPU, access to memory is still slower, simple design and implementation, easy to modify or debug)
3 stages of instruction execution cycle: **Fetch Instruction (FI), Decode Instruction (DI), Instruction Execution: Calculate Operand Address (CO), Fetch Operands from registers (FO), Execute Instruction (EI), Write Operand (WO)**
Instruction Pipelines: To increase the throughput of instruction execution, for modern processors, usually no CO stage except for LOAD and STORE instructions. The resources required by each stage should not overlap with each other, otherwise duplicate resources are needed (e.g. dedicated incrementer for PC instead of using ALU, separate read ports for data and instruction, multiple internal buses instead of single bus)
Pipeline hazards: **Resource hazard** (resource conflict), **Data hazard** (instruction waiting for the result of previous instructions (data dependency), **Control hazard** (don't know where to continue until the branch instruction finishes)
Solutions: **For resource hazard** (Dedicated incrementer for PC, separate data and instruction cache, FI and FO will use different hardware), **For data hazard** (rearrange the instructions, hardware solution using data forwarding), **For control hazard** (Branch prediction, Dynamic Branch prediction)
Processor performance: **Execution Time = Instruction Count x CPI (Clock per instruction) x Clock cycle time**

Instruction TYPE	Instruction count	Clock cycle count
Integer Arithmetic	45000	1
Data transfer	32000	2
Floating point	15000	2
Control transfer	8000	2

	Cache	Virtual
Placement	only on a particular set	full associative
Identification	Tag Matching by associative memory	Table lookup throu Page Table
Replacement	FIFO/LRU	Approximate FIFO/LRU
Write	either	always write back

Determine the effective CPI, MIPS (Millions of instructions per second)rate, and execution time for this program.

$$\mathrm{CPI} = \frac{45000\times 1 + 32000\times 2 + 15000\times 2 + 8000\times 2}{100000} = \frac{155000}{100000} = 1.55$$

$$400Mhz = 400,000,000Hz$$

$$\textbf{Execution time}(T) = \textbf{CPI} \times \textbf{Instruction count} \times \textbf{clock time} = \frac{\textbf{CPI} \times \textbf{Instruction Count}}{\textbf{frequency}} = \frac{1.55\times 100000}{400\times 1000000} = \frac{1.55}{4000} = 0.0003875\,\mathrm{sec} = 0.3875\,\mathrm{ms}$$

To increase processor performance: Exploit more effective instruction pipelining (can lead to smaller value of CPI), use multiple instruction execution units, large register file (reduce memory access, hence increases speed), use simplified instruction set (reduces or eliminates the need of microprograms, which is slower than hardwired implementation)
Performance issue: **Simple instruction set and hardwired logic** → high clock rate/low clock cycle time, **Extensive pipelining** → low CPI, **Simple instruction** → high instruction counts (only 20%-30%, can have overall performance increases)