Notes for Psychology 465A

Dr. Lawrence M. Ward

1. What is a Computer?

1.1 A digital computer is a *symbol manipulator.*

1.11 It is electronic so it is very fast - millions of operations per second.

1.12 The computer always does the same manipulations - elementary logic - but the manipulations differ in meaning according to the interpretations of the symbols manipulated. At this time only humans function as interpreters of computer operations.

1.2 Computers (and people) use *codes* to represent symbols and their manipulations.

1.21 *Code* - a set of symbols and transformation rules that is used to represent complex patterns economically.

1.211 Example: the English alphabet has 26 letters. We can form $26^6 \approx 3 \times 10^8$ different (e.g.) 6-letter words. This is more than adequate for the roughly 450,000 words of English. It allows transformation rules (e.g., xyzwx is not a word) that limit which patterns are words and thus aid interpretation.

1.22 A code must have an appropriate symbol set in order to be useful.

1.221 Would a 2-letter alphabet be okay? Well, we would need a word length of about 19 symbols to get our lexicon ($2^{19} = 524{,}288$) and would need nearly all of the available patterns. So we would have to distinguish (e.g.) “xyxyxyxyxyxyxyxyxy” from “xyxyxyxyxyxxxyxyxy.” This is clearly unsatisfactory. On the other hand, the other extreme, a different symbol for every word, gives too many symbols to learn, print, etc. (but cf. Chinese and Japanese). English alphabet does seem to be a good compromise (human span of apprehension and memory chunk size is about 7 ± 2, the "magical" number).

1.23 Computer code: the problem is how to represent alphabetic characters (including numerals, punctuation marks, etc.) and numbers in the computer.

1.231 Electronic and logical constraints indicate a two-state device is optimal as a basis for the computer, which implies that a two-symbol code is appropriate. This is okay for computers since they don't have human memory and perceptual limitations, but produces an ongoing problem of interfacing with human operators (the necessary interpreters).

1.232 We use the symbols "**0**" and "**1**" as the two required symbols. These can either stand for the electronic state of a two-state device or a wire (voltage levels) or for the "logical state" of the device or wire.

1.24 Some common computer codes for alphanumeric characters:

1.241 ASCII (American Standard Code for Information Interchange): uses 8-symbol "words" (e.g., 00101110) to stand for alphabetic, numeric, graphics, printer, etc. characters. Part (alphabetic and numeric) is standard and part is unique to a brand of computer.

1.242 BCD (Binary Coded Decimal): simple code used in some peripherals, counter-timers, etc. Similar to code used for numbers (section 1.25).

1.243 EBCDIC (Extended Binary Code for Decimal InterChange): uses 8-symbol "words" similarly to ASCII; used in mainframe computers.

1.25 Computer coding of *numbers*

1.251 *Number*: an abstract mathematical concept (Plato) usually represented by strings of *numerals.*

1.252 We use an exponential, positional, based number representation system:

$$N = aB^{\alpha} + bB^{\alpha-1} + cB^{\alpha-2} + ... + mB^{0} + nB^{-1} + pB^{-2} + ... + yB^{-\alpha}$$

↑ "decimal" point

where N is the number we wish to represent, B is the *base* (the number of different symbols used), the constants a, b, c ... (written as the numerals) represent how many B^{α}s are meant, and α depends on position as in the equation above. The $B^{\alpha}s$ are implicit. B is completely arbitrary, but $a, b, c... \leq B - 1$. The exponents, α, can be as large as needed. We use $B = 10$ for counting *by convention* (because we have 10 "digits"?).

1.253 Example: decimal ($B = 10$) number 164.75:

$$164.75 = (1)10^{2} + (6)10^{1} + (4)10^{0} + (7)10^{-1} + (5)10^{-2}$$

↑ decimal point

1.254 Bases used in computing:

2 - binary	used by all computers
8 - octal	used by minicomputers (defunct)
10 - decimal	used for counting, and in math
12 - duodecimal	out of date, old computers
16 - hexadecimal	used by microcomputers and mainframes

1.255 Example: Express 164.75 in $B = 2$ notation [Use algorithm: find largest a such that aB^{α} in base 10 goes into N no more than B-1 times, subtract ($N-aB^{\alpha}$), repeat for remainder and b, α-1, etc., until remainder equal to zero.]

Step	aB$^{\alpha}$	decimal	remainder
1	$(1)2^7$	128	36.75
2	$(1)2^5$	32	4.75
3	$(1)2^2$	4	.75
4	$(1)2^{-1}$	.5	.25
5	$(1)2^{-2}$	.25	0.0

$$164.75 = (1)2^7 + (0)2^6 + (1)2^5 + (0)2^4 + (0)2^3 + (1)2^2 + (0)2^1 + (0)2^0 + (1)2^{-1} + (1)2^{-2}$$

$$= 10100100.11$$

1.256 Binary number system uses digits **0, l** (called Binary Digits or *bits).* Note that these are the same as used in the electronic device that encodes symbols in the computer and as the two symbols used in basic logic. It is thus natural for computers to encode numbers in base 2 representation and to do base 2, or binary, arithmetic.

1257 Hexadecimal number system ($B = 16$) uses digits 0-9 and A-F. Each set of four binary digits encodes one hex digit since $2^4 = 16$:

Dec	Bin	Hex	Dec	Bin	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

1.258 Typical microcomputers use binary and hex; particular programming or operating systems may allow use of decimal representation in some commands.

1.3 Binary arithmetic (n.b. we don't usually write leading zeros but sometimes). This is same as decimal arithmetic except $B = 2$ so we carry and borrow two (“10") not ten (also “10").

1.31 Addition: **A + B**

B \ A	0	1
0	0	1
1	1	0*

* carry 1 to next position to left

1.311 Example:

```
  1011
+ 0101
-----------
 10000
```

Note: we don’t have *names* for binary numbers; we simply list the digits.

1.32 Subtraction: **A - B**

B \ A	0	1
0	0	1
1	1*	0

* borrow 1 (2^{α}) from next position to left

1.321 Example:

```
  1010
- 0011
-----------
  0111
```

1.33 Multiplication: **A x B**

B \ A	0	1
0	0	0
1	0	1

1.331 Example:

```
   1010
 x 1010
------------
   0000
  1010
 0000
1010
--------------
1100100
```

Note: multiplication consists of shift left and add operations on successive values of multiplicand (if multiplier = 1) or shift left only (if multiplier = 0).

1.34 Division: **A ÷ B**

B \ A	0	1
0	not	def
1	0	1

1.341 Example:

```
     00111
   ----------
11 | 10101
      11
    --------
      100
       11
    --------
        11
        11
    --------
         0
```

Note: division consists of shift right and subtract divisor from dividend operations if divisor is less than dividend, shift right only if not.

1.4 Radix complement representation of negative numbers

1.41 The *radix complement* of a number is defined by:

$$C_r(x) = R - x$$

where R is larger than largest number x can represent, and $R = r^{n+1}$ where $n + 1$ is the number of digits available to represent numbers.

1.42 If we represent negative numbers in this way then it can be shown that

$$y - x = y + (R-x) \bmod R = y + C_r(x) \bmod R$$

where *mod R* means addition using residue (or modular) arithmetic. [In residue arithmetic, a number is represented by the *remainder* or *residue* after dividing the number by the modulus (*R*). For example, let $R = 2$. Then 0 (mod 2) = 0, 1 (mod 2) = 1, 2 (mod 2) = 0, 3 (mod 2) = 1, etc. I.e. all even integers become 0, all odd integers become 1. What is 255 (mod 2)? What is 255 (mod 5)? What is 38 (mod 100)?] In other words, subtraction can be replaced by addition. If getting the radix complement of a number is easier than subtraction, a computational savings will result.

1.43 Since computers use binary representation we will consider only $r = 2$, giving what is called the *2s complement* representation for negative numbers:

$$C_2(x) = 2^{n+1} - x$$

1.44 Consider $n + 1 = 4$ (i.e., there are 4 bits available to represent numbers). Let $x = 11$ (binary notation). Then $C_2(x) = 2^4 - 11 = 10000 - 0011 = 1101$.

1.45 Is this simpler than subtraction? Only if we use a trick: the 2's complement of a number can be found by replacing all 0s by ls and all ls by 0s and then adding 1 (mod 2^{n+1}), making sure we now use all $n + 1$ available bits to represent our numbers. So for the example in 1.44 above we have

```
  0011
  ↓↓↓↓
  1100
+ 0001
---------
  1101
```

Note: if there had been a carry out to the left beyond position 4 it would simply have been dropped since addition is mod 2^4.

1.46 Now consider a subtraction problem using 2's complement representation: $y - x = 101 - 11 = 0101 + C_2(11) \bmod 2^4 = 0101 + 1101 = 0010$. In decimal form this is $5 - 3 = 2$. Note:

```
    0101
  + 1101
  ------------
1 | 0010
```
We drop digit to left of vertical line because mod 2^4.

1.47 With $n + 1$ bits available, positive integers from 0 to $2^{n+1} - 1$ or positive and negative integers from $2^n - 1$ to -2^n can be represented. For $n + 1 = 4$, $n = 3$, and we have positive integers from 0 to $2^{n+1} - 1 = 10000 - 1 = 1111$ (= 15) or positive and negative integers from $2^n - 1 = 1000 - 1 = 0111$ (=7) to $-2^n = 1000$ (= -8), a total of 16 different patterns of 0's and l's to represent a total of 16 different integers. When doing arithmetic we use both positive and negative integers (and thus 2's complement representation), but addressing uses only positive integers.

1.48 For $n + 1 = 4$, the table below is complete. As an exercise, see if you can create a comparable table for other values of $n + 1$.

Decimal	Binary 2's compl	Decimal	Binary 2's compl
7	0111	-8	1000
6	0110	-7	1001
5	0101	-6	1010
4	0100	-5	1011
3	0011	-4	1100
2	0010	-3	1101
1	0001	-2	1110
0	0000	-1	1111

1.49 For $n + 1 = 8$, integers range from 127 to -128; for $n + 1 = 16$, integers range from 32767 to -32768; $n + 1 = 32$ or $n + 1 = 64$ for most microcomputers, but older computers and integers in most programming languages use $n + 1 = 16$.

1.5 Boolean algebra: the computer performs logical operations on the symbols 0,1.

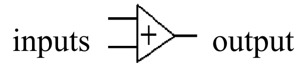
1.51 **OR:** inclusive (not the usual exclusive of language).

1.511 Truth table for 2-input OR gate:

Input l	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

Note: output is 1 *if any* input is 1. This can be generalized to any number of inputs by taking two of the inputs into a 2-input OR gate, then inputting the output of that gate and the next input into another 2-input OR gate, etc. as many times as necessary (this is called *cascading* the gates).

1.512 Symbol for 2-input OR gate logical device (realized electronically):



Labeled "+" because it is the *addition* operator in Boolean algebra.

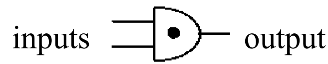
1.52 **AND:**

1.521 Truth table for 2-input AND gate:

Input l	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Note: output is 1 *if and only if all* inputs are 1. Generalizable to n inputs as for OR.

1.522 Symbol for 2-input AND gate:



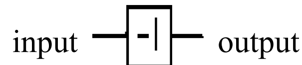
Labeled • because it is the multiplication operator in Boolean algebra.

1.53 **INVERSE** (i.e. negation):

1.531 Truth table for "inverter" gate:

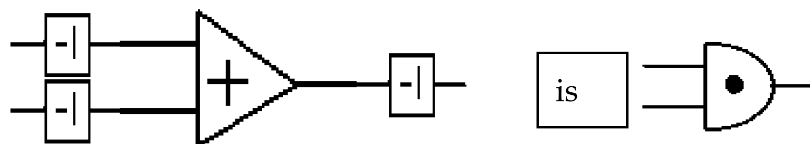
Input l	Output
0	1
1	0

1.532 Symbol:



This is equivalent to multiplying by -1 in Boolean algebra.

1.54 We really don't need AND because

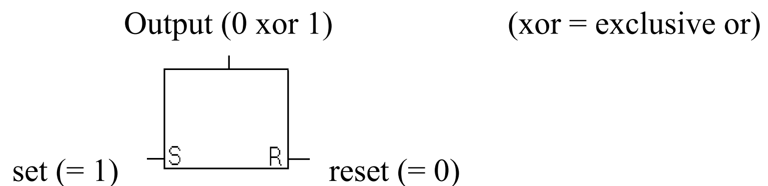


Proof:

In1	In2	Inv Inl	Inv In2	OR output	Inv OR out
0	0	1	1	1	0
0	1	1	0	1	0
1	0	0	1	1	0
1	1	0	0	0	1

1.55 Memory or storage unit: remains in state 0 or state 1 until changed *(set* or *reset).* Smallest unit stores 1 bit (1 binary digit) of information.

1.551 Symbol for memory unit, called the *flip-flop*:



If flip-flop is in state 0, activating "set" input (i.e. making set line go to 1 state) flips it to 1 state. If in 1 state, activating reset input flops it to 0.

1.56 These *logical* operators are all realized by *electronic* devices/circuits: diode gates (OR, AND, INV) and bistable multi-vibrator (flip-flop).

1.6 Registers

1.61 *Register*: a set of flip-flops functionally joined together.

1.62 A *digital computer* consists of organized groups of registers that process (and, or, inverse), route and store information (coded as binary digits - bits).

1.63 Since each flip-flop can store one bit of information, a register can store 1 bit times the number of flip-flops it contains. Typical microcomputer registers contain either 8, 16, 32 or 64 flip-flops, depending on the computer and the purpose of the register. Registers can often be manipulated in groups so that larger symbols can be easily manipulated.

1.64 *Byte*: a group of bits; now standardized at 8 bits.

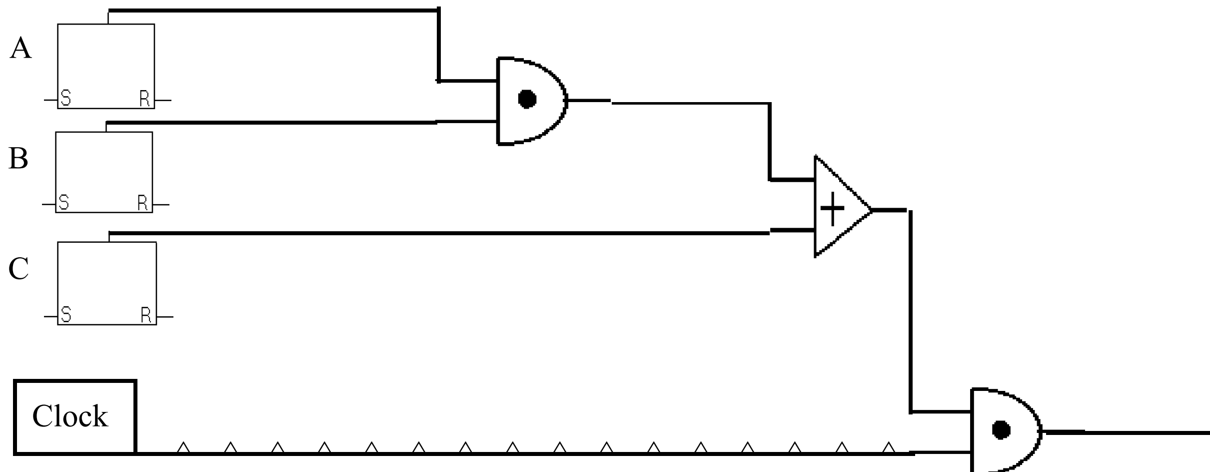
1.65 The memory size of a computer is given in bytes, or rather kilobytes (1 Kb = 2^{10} bytes = 1024 bytes) or in megabytes (1 Mb = 1024 Kb = 1,048,576 bytes. Typical microcomputers have memories of 8 Mb, 6 Mb, 32 Mb, 64 Mb ... 1 Gb (gigabyte = 1024 Mb) depending on the computer. External memory (e.g., disk drives) also are measured in Mb or Gb. Typical 3.5" floppy diskettes contain 1.44 Mb or 2.0 Mb, typical modern hard disk drives contain 2 Gb to 16 Gb.

1.66 The memory registers in a computer are called *RAM* memory (Random Access Memory). The are several other kinds of electronic memory: *ROM* (Read Only Memory), *PROM* (Programmable ROM), *EPROM* (Eraseable PROM), *EEPROM* (Electrically Eraseable PROM). Disk drive memories are magnetic or optical. Other types of memories are being worked on (e.g., bubble memories).

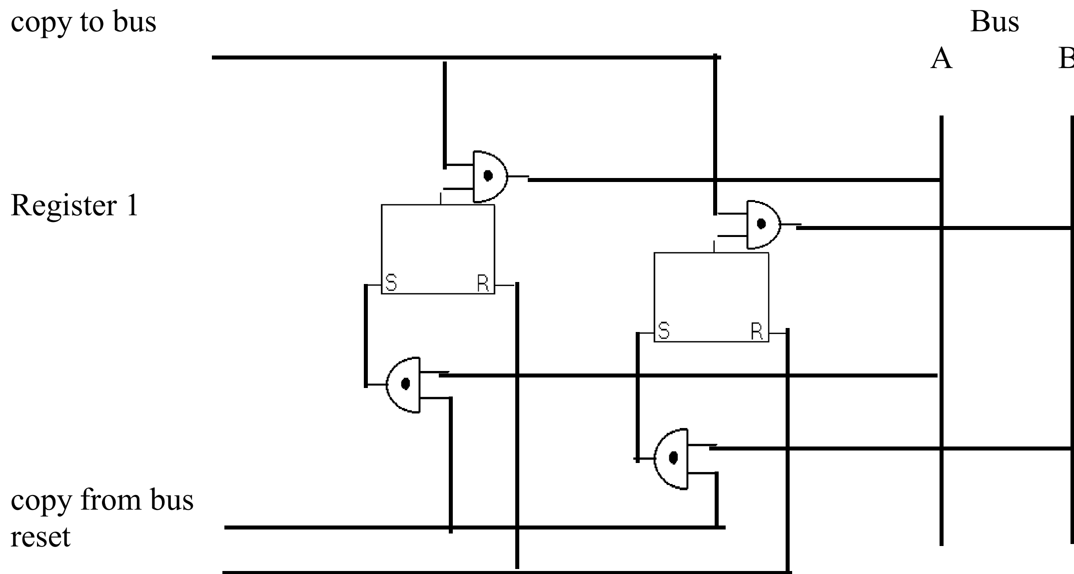
1.67 Computer word length (8-bit, 16-bit, 32-bit, 64-bit) refers to the size of the central registers (especially the Arithmetic and Logic Unit) and the data bus (a *bus* is a group of wires that connect two or more registers). Most microcomputers are either 16-bit or 32-bit machines. Memory address bus length determines the size of the memory that can be addressed (22 bits is standard, 32 bits on the more advanced microcomputers): addresses can range from 0 to 2^{22} -1 for a 22-bit address bus.

1.7 Registers perform many functions depending on how they are linked together. In the examples that follow, the circuits are simple ones that accomplish the tasks but they are not the only ones possible for those tasks. Modern computers use specialized gates and circuits that may not closely resemble the ones shown but do perform the same functions, usually more economically.

1.71 *Gating* refers to combining logic gates to perform a function. In this example, A, B, and C are three of the flip-flops of a register. The clock emits pulses (wire is in state 1 during a pulse, 0 otherwise) at a fixed rate for which the duration of a pulse is much less than the interval between pulses. In the circuit below, the final AND gate will emit pulses at the same rate as the clock whenever (and only then) *either* both flip-flops A and B are in state 1 *or* flip-flop is in state 1 *or* both. A counter attached after the lower AND gate would count the number of times these conditions obtained. Alternatively, we could see the lower AND gate as ensuring that the operations of the other gates would only be activated when a clock pulse occurred (this is how the entire computer operates; it contains many passive logic circuits that are activated by clock pulses into AND gates).

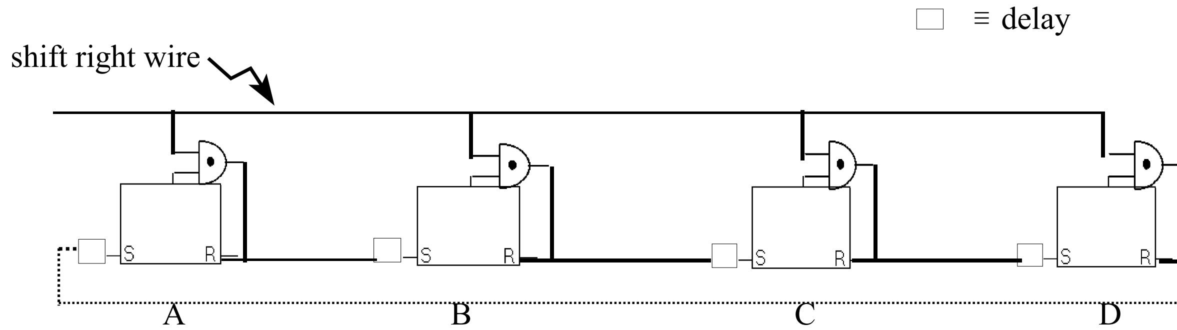


1.72 *Copying* (transferring): In the circuit below, the *bus* (a group of wires that transmit pulses) is always in the 0 state unless a pulse is traveling on it. The flipflops copy to the bus whenever a pulse travels down the “copy to bus” wire, activating the topmost AND gates. If the flipflop contains a 1 a pulse is sent to and along the corresponding bus wire; if a flipflop contains a 0 nothing happens because the AND gate has a 0 on its output unless its inputs are both 1s. The“copy from bus” is actually a microprogram of two steps: (1) pulse "reset" wire to put 0s in both flipflops, and (2) pulse “copy from bus” wire to activate lower AND gates. If bus has 1 then flipflop is set to 1; if bus has 0 nothing happens to connected flipflop.

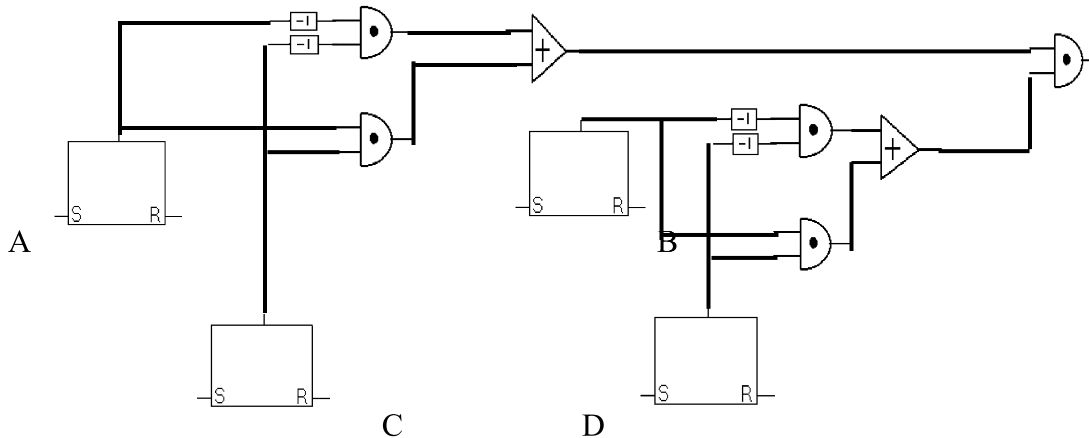


In order to copy from one register to another the CPU sends a “copy to bus” pulse to the origin register, copying its contents onto the bus, and then a “copy from bus” to the destination register, copying the contents of the origin register from the bus into the flipflops of the destination register.

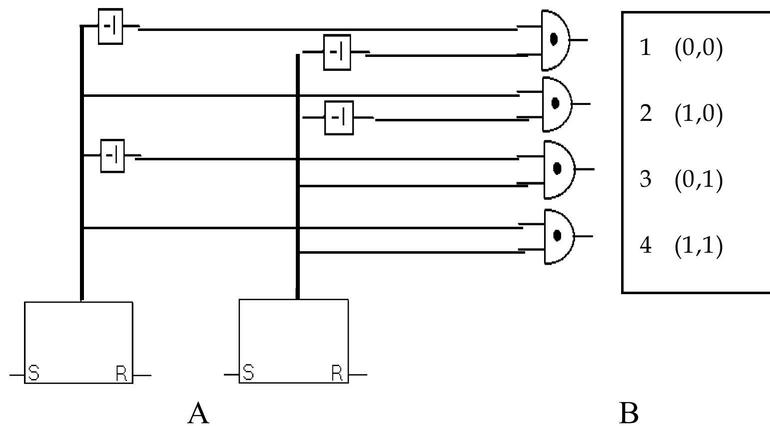
1.73 *Shifting* uses a *shift register* that operates as below. When "shift right" wire is pulsed, if flipflop A= 1 then a reset pulse is sent to flipflop A and a set pulse is sent to flipflop B (note the little box indicates a brief delay so that the set pulse arrives at B after B’s own reset pulse resets it to “0"), etc. for the others. If any flipflop is 0, then no pulses are sent from that flipflop. Each "shift" pulse only shifts contents one position, so repeated pulses are required to shift repeatedly (e.g., in division). Shift left is reverse (used in multiplication). The shift register has both circuits and separate shift wires for shift left and shift right operations. Notice the importance of timing: each flipflop in state 1 is reset to 0 just before it is (possibly) set to 1 by the flipflop to its left. Also, the contents of the last flipflop (D here) are lost unless, as below, it is tied to the first flipflop (A here).



1.74 *Comparing:* are the contents of Register 1 identical to those of Register 2? A, B are flipflops of Register 1 and C, D are those for Register 2. *Here and in what follows the wires that are pulsed to implement the function have been dropped so as to simplify the circuitry; however, they must always be assumed to be ANDed to the output of each flip-flop so that the actual input to the various gates shown would actually be the output of an additional AND-gate.* [As an exercise try adding these AND-gates and the appropriate wires.] Note that in the circuit below, if the output of the rightmost AND-gate is 1 then the two registers have identical contents, if 0 they are different in *at least* one flip-flop.



1.75 *Decoding:* selecting one of *n* output lines depending on the number *n* encoded into the flipflop states of a register. Below is a circuit for decoding the outputs of a 2-flipflop register (A, B). It works as follows. If $A = 0$ *and* $B = 0$ then output line 1 will be in state 1 while lines 2-4 will be in state 0. Similarly for the others, only one output line will be in state 1 for any particular combination of states of A, B. Decoding is used in the computer to select a line to pulse in order to operate on the contents of a particular register; see address decoder in 1.8.



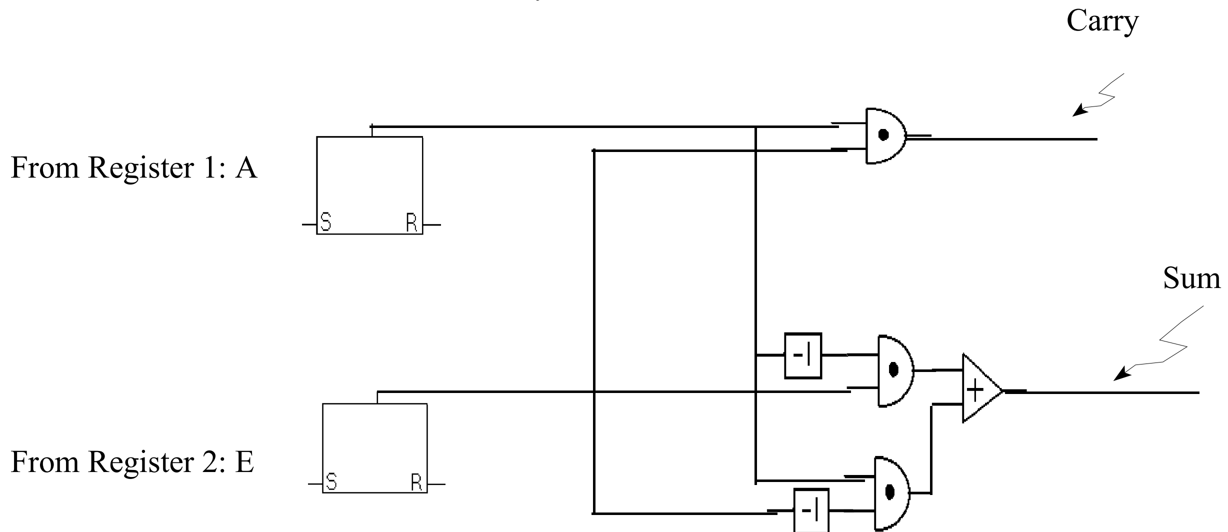
1.76 *Adding*

1.761 In binary addition if two binary digits to be added are the "same" (i.e., both 0 or both 1) the result is 0; if they are different the result is 1 (see section 1.31).

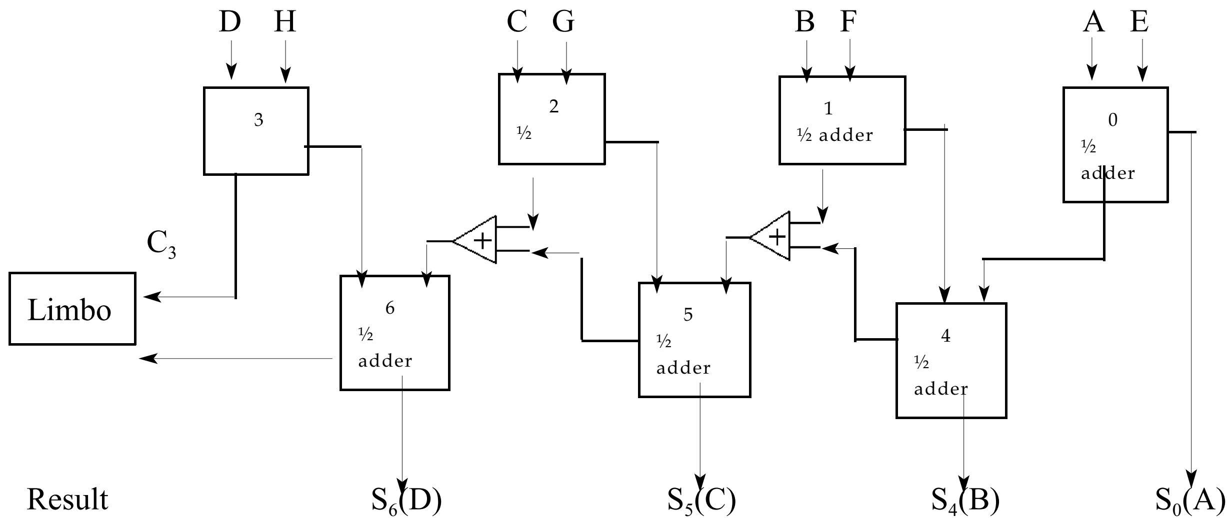
1.762 In the special case of 1 + 1 = O* there is a carry to the next position to the left.

1.763 A device to perform addition of the contents of two flip-flops but without the carry bit included is called a *half-adder*. A full adder combines half-adders to deal with the carry bits and add the contents of two entire registers.

1.764 Below is an example of a half-adder. Notice that it has two outputs, a *sum* output that reflects the rule in 1.761, and a *carry* output that reflects the rule in 1.762. The carry output will be 1 if and onlv if *both* flipflops A and E are 1, in which case the sum output will be 0. If both A and E are 0, the carry will be 0 and so will the sum. If either A = 1 and E = *0 or* A = 0 and E = 1, the carry bit will be 0 but the sum bit will be 1.



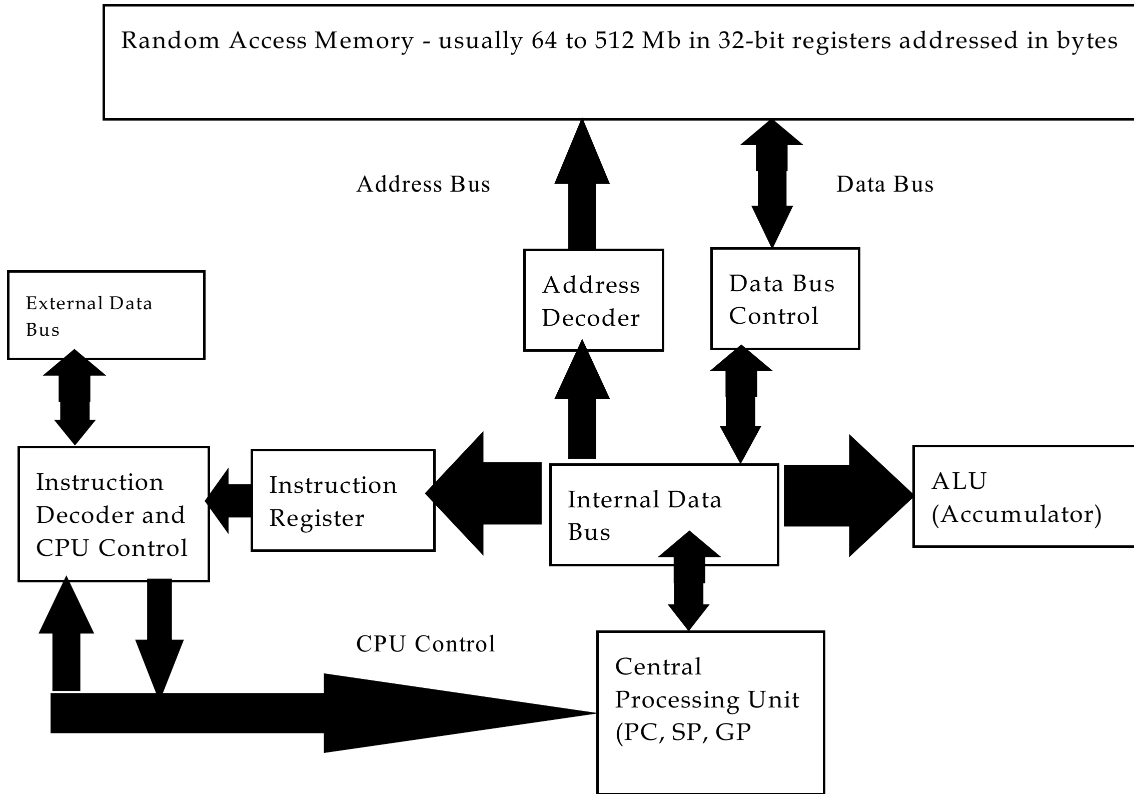
1.765 Below is an example of a full adder to add the contents of two 4-bit registers (AB,C,D and E,F,G,H - note the register symbols have been omitted). "1/2 Adder" stands for the entire circuit in 1.764 with S_i (or rightmost output if omitted) indicating the sum output of half-adder *i*, and C_i (leftmost) indicating the carry output for that half-adder. One of the registers is usually in the *Arithmetic and Logic Unit* (ALU) (that register is called the *accumulator)* and so the result is usually fed back into that register (i.e., the contents of an outside register are added to those of the ALU register). Also, the carry outputs of the final adders have nowhere to go and so they are lost (residue arithmetic - see section 1.42).



1.77 Note: The logic circuits given above are sufficient for most computer operations since all arithmetic can be done by addition and shifting when 2's complement representation of negative numbers is used.

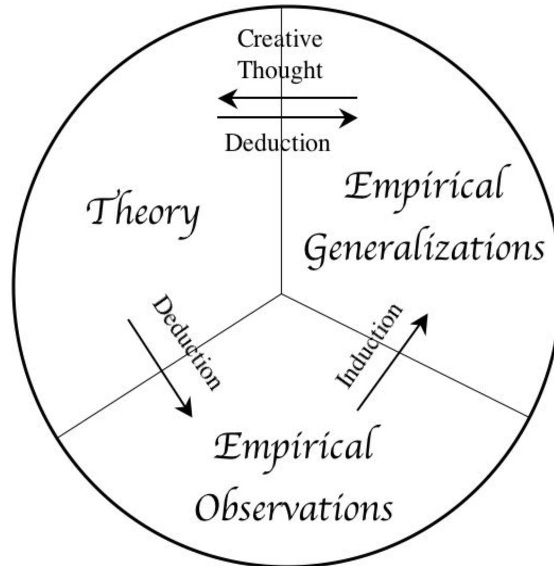
1.8 Below is a drawing of a generic microcomputer at a fairly general level (i.e., only a few individual registers are shown). The various parts can be connected in other ways in actual computers, depending on the architecture. "Microprograms" are implemented by sending out pulses on lines from the Instruction Decoder and CPU Control in a temporal order dictated by other logic circuits and a clock in the Instruction Decoder. Peripherals are connected to a bus that resembles the memory address and data buses (it can be the same one). The program counter (PC register in the CPU) contains the address of the "next" instruction which is fetched from RAM when the current one has finished executing. The Instruction Decoder decodes instructions into microprograms (sequences of pulses on wires). The ALU does arithmetic (only adding and shifting - see section 1.5) and logic (ANDing, ORing, INVERTING), while the CPU has several general purpose registers and several special purpose registers (e.g., the PC). Data and program instructions (the contents of RAM registers) are sent to and fetched from RAM by copying an address onto the address bus, which decodes it into a copy pulse to or from

the data bus for a particular register; if to the data bus it is copied onto the internal data bus and then into the appropriate destination register (e.g., instruction register or accumulator or another register in CPU or RAM or an external register). The program is stored in RAM as a series of instructions in registers the address of the first one of which is loaded into the program counter to start the program.



Computer Simulation of Psychological Theories

1. What is a **theory**?
   1.1 One of the products in the Mandala of Science (Michael Ovenden).



   1.2 Set of general principles from which empirical observations (experiments) and empirical generalizations (laws) can be deduced.
   1.3 Can be informal (verbal, pictorial) or formal (mathematical or other formal system).
   1.4 Deduction in 1.1 and 1.2 above constitutes "explanation."

2. **Model**: a representation of a theory in some medium.
   2.1 Usually more concrete than a theory.
   2.2 Applies theory to a specific situation; more restricted.
   2.3 Model allows consequences of the theory to be explored but doesn't explain the phenomena.
   2.4 Computer model is **dynamic** (operates in time).

3. **Simulation**: operation of a model (e.g., running a computer program, model boat rocking with waves in a wave tank).
   3.1 Games are often simulations (e.g., Monopoly, Diplomacy).
   3.2 Abelson's definition: exercise of flexible imitation of processes and outcomes for purposes of clarifying and explaining underlying mechanism.
   3.3 Two basic forms:
      3.31 Model is a representation of a theory (as above).

3.32 Computer program **is** the theory.

3.4 **Computer simulation**: operation of a computer program, from input through output, that is a process model of some empirical phenomenon and is based on a theory concerning that phenomenon.

4. Benefits of computer simulation.
    - 4.1 Clarification of a theory: precise explicit statements are required by the computer.
    - 4.2 Critique of a theory: can the theory be useful?
    - 4.3 Completeness of a theory: both a test and an opportunity to introduce complexity.
    - 4.4 Generate new hypotheses and implications (deductions); .can study the theory.
    - 4.5 Model's processes parallel "real" processes.
    - 4.6 Cheap subjects; no ethical problems.
    - 4.7 Easy to represent and deal with randomness.
    - 4.8 Data archives
    - 4.9 Theory development tool/formal language.
    - 4.10 Can make predictions/forecasts.
    - 4.11 Can be used to train people in use or practical applications of theory.
    - 4.12 It's fun.

5. Drawbacks of computer simulation.
    - 5.1 Difficult to ascertain validity of results.
    - 5.2 Seductive - never finished.
    - 5.3 Simplification and precision may distort reality too much for usefulness.
    - 5.4 **Bonini's paradox**: the more complex (and thus realistic) the simulation, the more difficult it is to understand; a complex simulation may be no easier to understand than the processes represented.

6. Planning and organizing the simulation.
    - 6.1 Decide the purpose.
        - 6.11 Prediction/forecast: can often be done using a statistical model without process specification.
        - 6.12 Instruction: can be simpler; model only those phenomena that are to be taught.
        - 6.13 Science: simulate existing theory or break new ground.
    - 6.2 Summarize relevant phenomena; consider level of analysis at which phenomena are to be modeled.
    - 6.3 Specify conceptual model (model to be coded in program).
        - 6.31 It should be dynamic (processes in time).
        - 6.32 It should be a closed system: all inputs and variables completely specified.
        - 6.33 It should be well-specified: all of the following should be included.
            - 6.331 Units: the elements operated on
            - 6.332 Properties: sets of values and constantsbound to units.
            - 6.333 Inputs: these initialize and influence processes.

6.334 Processes: detailed specification of processes and subprocesses - functions and procedures that relate units and change their properties.

6.335 Sequencing: temporal (or logical) sequence of operation of processes: flowchart of program.

6.336 Consequences: output of program; usually properties of units; final or intermediate; can be analysed as real data are.

6.4 Representation and organization of model in computer program

6.41 Computer is a binary, digital device. This limits representation, especially of infinite sets and processes.

6.42 Choose programming language (if possible) with regard to ease and "naturalness" of model representation it allows.

6.421 Data representation (e.g., in QBASIC integers, single and double precision numbers and strings either in single variables or arrays).

6.422 Processes available or definable (e.g., control structures such as conditional execution or iteration).

6.43 Choose data structures and processes as appropriate (difficult to do, no easy answers).

6.5 Sequence: flowchart at several levels.

6.6 Programming:

6.61 Structured programs best

6.62 Modularize

6.63 Subroutines useful because iteration of same processes common; use supplied functions as available.

6.64 Use GUI as needed for ease of operation; can set parameter values and display results dynamically

7. Random processes: a discrete simulation of sampling from a known continuous probability distribution

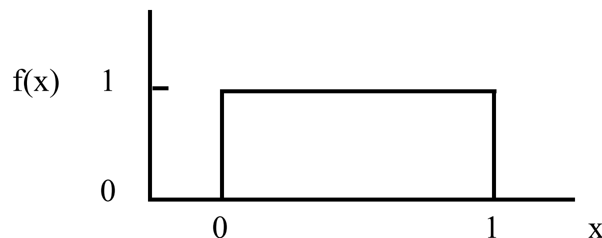
7.1 The **rectangular** or **uniform probability distribution**.

7.11

$$f(x) = \begin{cases} 1/(b-a) & \text{for } a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = (a+b) / 2 \qquad \sigma^2 = (b-a)^2 / 12$$

7.12 Unit rectangular: $a = 0$, $b = 1$, $\mu = 1/2$, $\sigma^2 = 1/12$



7.2 **Pseudo-random numbers**: a series of the form $x_i = f(x_{i-1})$. For example, $x_1 = f(x_0)$, $x_2 = f(x_1)$, $x_3 = f(x_2)$, ... Choosing x_0 is obviously important since x_0 and f together completely determine the sequence. The sequence should generate numbers in various ranges with equal probabilities (uniformity) and should repeat itself (if ever) only at very long intervals (long period).

7.3 **Fibonacci sequence** (multiplicative congruential type):

$$x_i = a\, x_{i-1} \text{ (modulo } 2^m)$$

where $\boldsymbol{a}$ is a (large) constant, $\boldsymbol{m}$ is the length of the relevant register in bits (usually the accumulator) and $\boldsymbol{x_{i-1}}$ is the previous number in the series. In effect, multiplying two large numbers in a computer causes an overflow, yielding the remainder after dividing by 2^m, which is multiplication mod 2^m. This series of numbers has long period and uniformity. However, the actual implementation of this procedure in a program can be done well or poorly, and specific sequences may be more or less uniform. Also, there are several other tricks that can be done to increase the "randomness" of the sequence (see Press et al, *Numerical Recipes in C*, for a thorough discussion of random number generators and for some very good examples of generators written in the C programming language).

7.4 The multiplicative congruential method is used by Visual Basic 6 and most other programming environments to give a sequence of random numbers when you use the **RND** function. In its baseline mode, each call of RND, as in *x = RND*, returns to the variable *x* a number between 0 and 1 that is the next number in a type of Fibonacci sequence. The first number in the sequence (x0, called the "seed") can be provided by using the RANDOMIZE TIMER statement, which uses the current time of day as the seed. (See comments on RND and RANDOMIZE statements in the on-line HELP in VB6 for more information).

7.5 Because of variability in the quality of random number generators, and in order to ensure that a particular sequence is "really random," it is often necessary to test for "randomness." There are four criteria:

7.51 Long period: Fibonacci sequence always does this if **m** is sufficiently large (16 is large enough for most purposes; some compilers offer different values of **m** depending on the type of computer).

7.52 Rectangularity (uniformity): use χ^2 test for departure from rectangular distribution.

7.521 Divide range of numbers generated into n equal intervals.

7.522 Generate a large number, **N,** of random values using the relevant generator (e.g., RND), finding for each one the interval into which it falls and tallying the frequency, **f**, with which numbers fall into each interval, **i**.

7.523 Calculate "expected" number of samples that should fall into each interval, $\mathbf{e_i}$. For the uniform distribution, $\mathbf{e_i = e_j}$ for all **i,j** and $\mathbf{e_i = N/n}$ for all intervals.

7.524 Calculate

$$\chi^2 = \sum_{i=1}^{n} (f_i - e_i)^2 / e_i$$

Look in table of χ^2 values for critical value for df = n-1. If calculated χ^2 > critical value for (say) $\alpha = 0.05$, then reject rectangularity for that sequence.

7.53 Sequential independence: correlation of sequence with itself must be 0 at all lags other than 0.

7.531 The cycle 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9..... is rectangular but obviously predictable from a simple rule.

7.532 The correlation of a sequence with itself is called the autocorrelation “function" of the sequence because it must be calculated for each possible "lag." A lag is a displacement of the sequence from the copy it is being correlated with. For example, the lag of the two (identical) sequences below is 2:

```
3 4 5 8 1 8 6 4 9 3 4 5 1
    3 4 5 8 1 8 6 4 9 3 4 5 1
```

Note that each number in the top row is being paired with a number in the bottom row that occurred 2 places before it in the top row. The correlation of the top row with the bottom row (for only the paired numbers) is the autocorrelation at lag = 2.

7.533 One simple test of sequential independence is to calculate the autocorrelation function for all lags from 1 to some reasonable number, say 10. If none of them are significantly different from 0 using standard statistical techniques, then the sequence is sufficiently sequentially independent. The largest lag used will depend on the application, of course.

7.54 Runs & Gaps

7.541 Run: the number of elements in a subsequence in which each element is greater than the immediately preceding one (ascending run, e.g., 3456 is a run of integers of length 4) or in which each element is less than the immediately preceding one (descending run, e.g., 987654 is a run of integers of length 6). Typical test is to compare the number of ascending and descending runs of each length (up to some limit) with the number of such runs expected from the hypothesis of randomness using the same χ^2 test as for rectangularity.

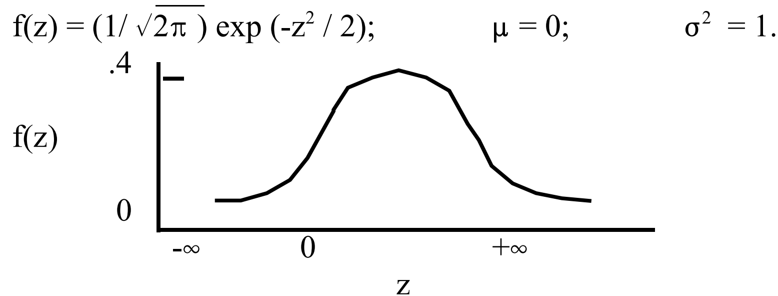
7.542 Gap: the number of elements in a subsequence intervening between a particular element and its recurrence. Typical test is to compare number of gaps of each length (up to some limit) with the

number expected in a random series of the same length using the χ^2 test.

7.6 Standard normal probability distribution

7.61 Continuous probability density function with definitions and graph:

$$f(z) = (1/\sqrt{2\pi})\exp(-z^2/2); \qquad \mu = 0; \qquad \sigma^2 = 1.$$



7.62 To generate simulated discrete samples from the standard normal probability distribution (called "normal variates"), notice that for samples, **x_i, i = 1,2,3,....n,** from the unit rectangular distribution, the sum,

$$y = x_1 + x_2 + x_3 + ... + x_n$$

more closely approximates a sample from a normal distribution the larger **n** is (by the central limit theorem of probability theory). To transform such samples to the standard normal, as in 7.61 above, we apply the z-transform. Since $\mu_y = n\,\mu_x = 0.5\,n$ and $\sigma_y^2 = n\,\sigma_x^2 = n(1/12) = n/12$, we have

$$z_y = (y - 0.5\,n) / (\sqrt{n/12}).$$

7.63 For example, for n = 12, (i.e., $y = \sum_{i=i}^{12} x_i$), we have

$$z_y = (y - 0.5\,(12)) / (\sqrt{12/12}) = (y - 6) / 1 = y - 6.$$

7.64 To test the "normality" of these simulated samples from the standard normal distribution, we would use the χ^2 test as we did for the uniform distribution.

7.641 Divide the range of **z** into a number of convenient intervals (e.g., -∞ to -3; -3 to -2.5; -2.5 to -2; ......).

7.642 Generate a large number, **N,** of samples, $\mathbf{z_y}$, using the procedure described above. For each sample, determine into which of several intervals on the z axis it falls and tally the number of samples that fall into each of the intervals, $\mathbf{f_i}$.

7.643 Calculate the expected frequency of samples in each of the intervals, $e_i = P(z_l \le z_y \le z_u)\,N$. Remember that for the normal distribution,

$$P(z_l \le z_y \le z_u) = \int_{z_l}^{z_u} (1/\sqrt{2\pi})\, \exp(-z^2/2)\, dz$$

which can be looked up for any interval in a table in any stats text.

7.644 Calculate

$$\chi^2 = \sum_i (f_i - e_i)^2 / e_i$$

and compare it with the tabled value for df number of intervals minus 1 and, say, $\alpha = 0.05$. If χ^2 > tabled value we must reject the null hypothesis that that particular group of samples was from the standard normal distribution.

8. Validation
    - 8.1 Apply criteria of a good theory:
        - 8.11 Generality
        - 8.12 Falsifiability
        - 8.13 Accuracy: deduction of empirical observations and generalizations (laws).
        - 8.14 Parsimony (Occam's razor): all other things being equal (*ceterus paribus*) choose the simplest theory.
        - 8.15 Beauty, elegance: Ceterus paribus, choose the most beautiful theory (Einstein believed his general theory of relativity to be correct not because of its agreement with observation, which hadn't happened yet, but because of its beauty and elegance).
    - 8.2 Is the computer simulation a valid representation of the theory? Its description must convince the reader.
    - 8.3 Indistinguishibility tests for the accuracy of the simulation (relevant to 8.13 above).
        - 8.31 **Turing test**: originally Alan Turing replaced the question "Can a computer think?" with the following question: Consider the old parlor game in which one man and one woman are hidden behind a curtain and interrogated with the aim of discovering from their answers which is the man and which the woman. Questions and answers are typewritten so that voices give no information. Now replace one of them with a computer. Is the interrogator no better able to distinguish the computer from the person than they are to distinguish the man from the woman?
        - 8.32 **Extended Turing test**: make an extended series of guesses for human-computer and man-woman pairs (where computer is simulating male or female). If both guess rates are better than chance (0.50) and not different from each other, then the simulation is successful.
        - 8.33 **Matching to known results**: Compare simulation output to empirical data using χ^2 or other test of goodness of fit (e.g., r^2). This is somewhat controversial, since with poor data, even a bad theory may be confirmed, but with good data even a good theory may be disconfirmed. Also, there

is the question of whether a qualitative match could be sufficient for establishing the usefulness of the theory.

8.34 The Turing test and other indistinguishibility tests (including a hypothetical series of Turing tests, each more stringent than the previous one) have become the criteria for establishing whether a computer can exhibit human qualities, including even consciousness.