Computer Organization (CS 1410) Fall 2017

Time and Place: 2 - 3:15 PM Tuesday and Thursday in ITT 322

eLearning Site: CS 1410 Computer Organization Section 01 Fall 2017
Course website: www.cs.uni.edu/~fienup/cs1410f17/

Class Email List: Send messages to CS-1410-01-FALL@uni.edu from your UNI account

Instructor: Mark Fienup (fienup@cs.uni.edu)
Office: ITTC 313
Phone: 273-5918 (Home 266-5379)
Office Hours: M: 8-11:45, 1:10-3, T: 1:10-2, W: 1:10-2, Th: 1:10-2, F: 8-10

Pre- or Co-requisite: Intro. to Computing (CS 1510) or any programming course

Goals: After this course, you should understand: (1) how data is represented and manipulated on the computer, (2) simple combinational and memory circuits used to build computer components, (3) how these circuits are organized to build a computer, (4) how to program in assembly language, (5) how high-level language programming languages are implemented with respect to the run-time stack and built-in data structures such as arrays and records, and (5) general concepts of hardware support necessary for an operating system.

Required Text: *Essentials of Computer Organization and Architecture*, 4th edition, by Linda Null and Julia Lobur. Jones and Bartlett Learning, 2015. ISBN-13: 978-1-284-04561-1.

Assignments: Assignments will be both "pencil-and-paper" exercises and assembly-language programming.

Pedagogic Approach: This is a "*flipped*" class! Before coming to each class, you will be asked to do the assigned reading, watch mini-lecture video(s), and take eLearning quiz(zes). The pre-class activities frees up class time to focus on the more challenging content of the course. In class, we'll dive into active and group learning worksheets and activities. As necessary I'll do mini-lectures with the whole class when extra explanation is needed. While the in-class worksheets (etc.) are not formally graded, part (5%) of your grade will be based on your participation in these in-class activities.

Grading policy: There will be three tests (including the final). I'll announce tests at least one week in advance to allow you time to prepare. Tentative schedule and weighting of course components are:

Pre-class Work:	15%	
In-class Work:	5 %	
Assignments:	20 %	
In-class Test 1:	20 %	(September 28)
In-class Test 2:	20 %	(November 2)
Final:	20 %	(Wednesday, December 13 from 1:00 to 2:50 PM in ITT 322)

Grades will be assigned based on straight percentages off the top student score. If the top student's score is 92%, then the grading scale will be, i.e., 100-82 A, 81.9-72 B, 71.9-62 C, 61.9-52 D, and below 52 F. Plus and minus grades will be assigned for students near cutoff points.

Scholastic Conduct: You are responsible for being familiar with the University' Academic Ethics Policies (http://www.uni.edu/pres/policies/301.shtml). Copying from other students is expressly forbidden. Doing so on exams or assignments will be penalized every time it is discovered. The penalty can vary from zero credit for the copied items (first offense) up to a failing grade for the course. If an assignment makes you realize you don't understand the material, ask questions designed to improve your understanding, *not* ones designed to discover how another student solved the assignment. The solutions to assignments should be **individual, original** work unless otherwise specified. Remember: discussing assignments is good. Copying code or test-question answers is cheating.

Any substantive contribution to your assignment solution by another person or taken from a publication (**or the**

web) should be properly acknowledged in writing. Failure to do so is plagiarism and will necessitate disciplinary action. In addition to the activities we can all agree are cheating (plagiarism, bringing notes to a closed book exam, etc), assisting or collaborating on cheating is cheating. Cheating can result in failing the course and/or more severe disciplinary actions.

Special Notices:

- In compliance with the University of Northern Iowa policy and equal access laws, I am available to discuss appropriate academic accommodations that may be required for students with disabilities. Requests for academic accommodations are to be made during the first three weeks of the semester, except for unusual circumstances, so arrangements can be made. Students are encouraged to register with Student Disability Services, 103 Student Health Center, to verify their eligibility for appropriate accommodations.

- I encourage you to utilize the Academic Learning Center's free assistance with writing, math, science, reading, and learning strategies. UNI's Academic Learning Center is located in 008 ITTC. Visit the website at http://www.uni.edu/unialc/ or phone 319-273-2361 for more information.

Computer Organization Fall 2017

Lect #	Tuesday		Thursday	
1	8/22	Sect 1.1-1.8: Introduction, Terminology, and Computer History	8/24	Sect 2.1-2.3: Unsigned integer numbers; Converting between base 10, 2, and 16
3	8/29	Sect 2.4: Signed integers	8/31	Sect 2.5: Floating Point Representation
5	9/5	Sect 2.6-2.7: Character representation and Error Detection and Correction	9/7	Sect 3.1-3.3: Boolean logic and Gates
7	9/12	Sect 3.4-3.5: Digital Components: decoders, multiplexers, adders	9/14	Sect 3.6: 1-bit memories: SR-latch, timing diagrams, D-latch, D-flip flop, Shift register & Register file
9	9/19	Register file vs. Square-Memory RAM	9/21	Sect 4.8: MARIE architecture and assembly language
11	9/26	Review for Test 1	9/28	**Test 1**
13	10/3	4.1-4.9: MARIE CPU, Bus, Clock, I/O, Memory, and RTL/RTN	10/5	Sect 4.10-4.13: MARIE Hardwired Control Unit
15	10/10	MARIE Microprogrammed Control Unit	10/12	Sect 4.14: MIPS Architecture and Assembly Language Control structures
17	10/17	Sect 4.14: MIPS Assembly Language 1D-Array Examples	10/19	Practice MIPS; Walking pointers in an array and 2-D arrays
19	10/24	Run-time stack in HLL; CalculatePowers MIPS calling convention example	10/26	Insertion Sort MIPS calling convention example
21	10/31	Review for Test 2	11/2	**Test 2**
23	11/7	PCSpim I/O; MIPS Logical and shift instruction	11/9	Sect 8.1-8.2: Hardware support for OS: CPU timer, privileged instrs, dual-mode CPU, memory protection; Sect 7.1-7.4: I/O modules and methods (programmed-I/O, interrupt-driven I/O, DMA I/O) Sect 9.1-9.4: Memory-mapped vs. I/O instructions
25	11/14	Sect 8.1-8.2: OS queues and process mgt; Instr Pipelining; Data and control hazards	11/16	Dynamic Branch Prediction: Branch-Prediction Buffer (BPB/BHT)
27	11/28	Superscalar, WAR & WAW dependencies; Ch 5: Memory Hierarchy and cache	11/30	Virtual Memory: paging, page table, TLB, segmentation and combining
29	12/5	Virtual Memory Examples	12/7	Review for Final
Final: Wednesday, December 13 from 1:00 to 2:50 PM in ITT 322				

Below is the description of the hardware features of a desktop PC:

- Intel® 2nd Generation Core™ i3 (3.4GHz, L2 Cache Memory 3MB)
- System Memory: (RAM) 8GB DDR3 SDRAM
- Hard Drive: 1.5TB SATA (7200 rpm)
- Intel® HD Graphics 2000 (Video Memory Up to 4GB shared)
- Network Card: Built-in 10/100/1000Base-T Ethernet LAN
- Wireless Networking: Built-in 802.11b/g/n wireless LAN
- Recordable DVD Drive: 8x DVD+R DL; 8x DVD-R DL; 16x8x16 DVD+RW; 16x6x16 DVD-RW; 5x DVD-RAM; 40x24x40 CD-RW
- Digital Media Card Reader
- Available Expansion Bays: External: 1 (3.5"), 1 (5.25"); Internal: 1 (3.5")
- Available Expansion Slots: 1 PCI Express x1, 1 PCI Express x16
- USB 2.0 Ports: 2 USB 3.0 (front); 4 USB 2.0 (rear)
- Wireless Keyboard with volume control
- Wireless optical mouse
- 20" widescreen flat panel monitor
- Operating System Windows 8

1) What does the processor do?

2) What is stored in main memory (RAM)?

3) What is stored on the hard disk?

4) What is the purpose of cache memory?

5) What terms relate to interconnection of internal PC components?

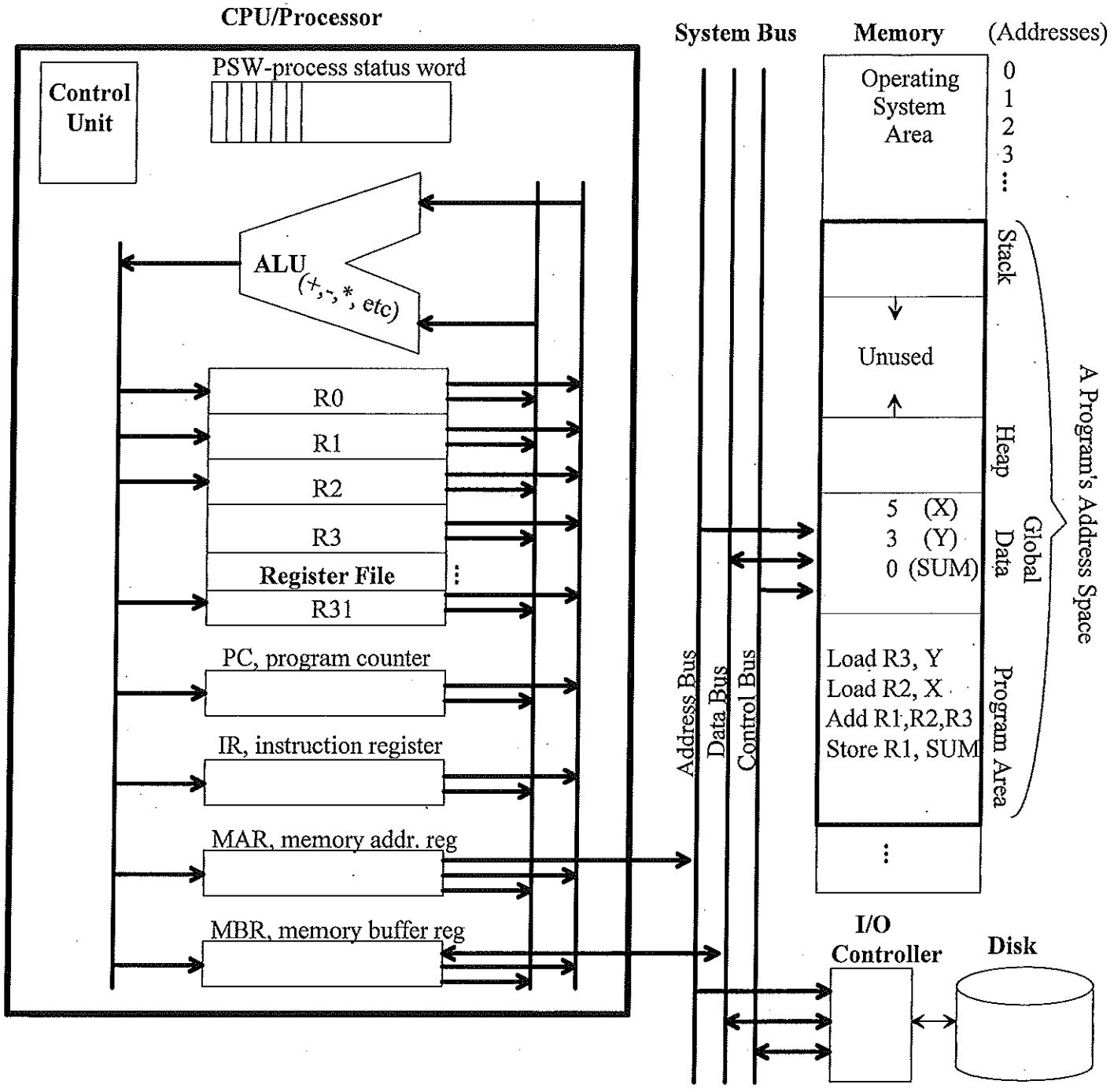
6) What terms relate to interconnection of external PC components?

7) What is a KB, MB, GB, Gigabit, MHz, GHz?

8) What is the role of the operating system?

High-level lang. (HLL) **Assembly Lang. (AL)** **Machine Lang. (ML) (three-addr. RISC format)**

```
SUM = X + Y
```



***Processing (/Instruction/Machine) Cycle* of stored-program computer - repeat all day**

1. Fetch Instruction - read instruction pointed at by the program counter (PC) from memory into Instruction Reg. (IR)
2. Decode Instruction - figure out what kind of instruction was read
3. Fetch Operands - get operand values from the memory or registers
4. Execute Instruction - do some operation with the operands to get some result
5. Write Result - put the result into a register or in a memory location

(Note: Sometime during the above steps, the PC is updated to point to the next instruction.)

9) What is the role of a compiler?

10) What advantages do high-level languages (Ada, C, C++, Java, Python, etc.) have over assembly language?

11) Why do people program in assembly language (AL)?

Type of Instruction	MIPS Assembly Language	Register Transfer Language Description
Memory Access (Load and Store)	lw $4, Mem	$4← [Mem]
	sw $4, Mem	Mem←$4
	lw $4, 16($3)	$4← [Mem at address in $3 + 16]
	sw $4, Mem	[Mem at address in $3 + 16]← $4
Move	move $4, $2	$4← $2
	li $4, 100	$4← 100
Load Address	la $5, mem	$4← load address of mem
Arithmetic Instruction (reg. operands only)	add $4, $2, $3	$4← $2 + $3
	mul $10, $12, $8	$10← $12 * $8 (32-bit product)
	sub $4, $2, $3	$4← $2 - $3
Arithmetic with Immediates (last operand must be an integer)	addi $4, $2, 100	$4← $2 + 100
	mul $4, $2, 100	$4← $2 * 100 (32-bit product)
Conditional Branch	bgt $4, $2, LABEL (bge, blt, ble, beq, bne)	Branch to LABEL if $4 > $2
Unconditional Branch	j LABEL	Always Branch to LABEL

Fibonacci Sequence:	0	1	1	2	3	5	8	13	21
Position in Sequence:	0	1	2	3	4	5	6	7	8

A high-level language program to calculate the nth fibonacci number would be:

```
temp2 = 0
temp3 = 1

for i = 2 to n do
    temp4 = temp2 + temp3
    temp2 = temp3
    temp3 = temp4
end for

result = temp4
```

Trace of Program (time →)

HLL variables								MIPS registers
temp2	0	1	1	2	3	5	8	$2
temp3	1	1	2	3	5	8	13	$3
temp4	1	2	3	5	8	13	21	$4
i	2	3	4	5	6	7	8	$6

(n in $5)

A complete assembly language MIPS program to calculate the nth fibonacci number.

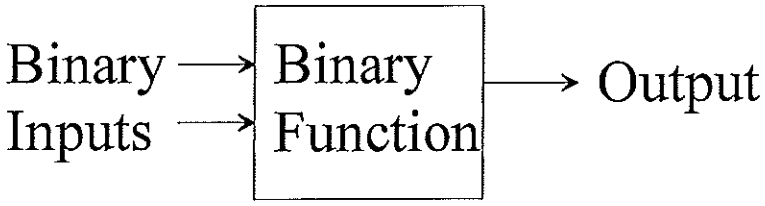
```
                .data
n:              .word   8           # variable in memory
result:         .word   0           # variable in memory


                .text
                .globl main
main:       li      $2, 0           # $2 holds temp2
            li      $3, 1           # $3 holds temp3
for_init:   li      $6, 2           # initialize i ($6) to 2
            lw      $5, n           # load "n" into $5
for_loop:   bgt     $6, $5, end_for # if $6 >= $5, then branch to end_for label
            add     $4, $2, $3      # $4 holds temp4
            move    $2, $3          # shift temp3 to temp2
            move    $3, $4          # shift temp4 to temp3
            addi    $6, $6, 1       # increment i ($6)
            j       for_loop        # unconditionally jump to for_loop label
end_for:
            sw      $4, result      # store the result to memory
            li      $v0, 10         # system code for exit
            syscall                 # call the operating system
```

Computer Technology

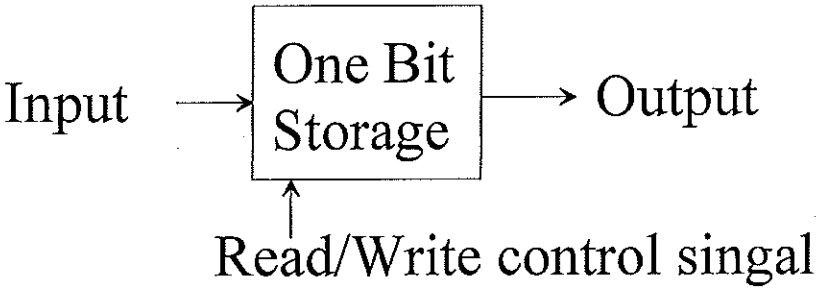
Built from two types of components plus interconnection:

Gates:



A	B	OR
F	F	F
F	T	T
T	F	T
T	T	T

Memory Cells:



Circuit - collection of gates and/or memory cells to perform some function

Computer Generations

1st Generation - vacuum tubes with wires for interconnection (one gate per vacuum tube)

Examples: ENIAC - 1943-46 Army's Ballistics Lab
Mauchly/Eckert (U. of Penn.) 1st general-purpose electronic digital computer
30 tons, 15,000 sq. ft., 18,000 vacuum tubes, 140 KW, 5000 adds/sec
Programmed with wires and switches

1945-52 - IAS John von Neumann (Princeton) used stored-program concept

2nd Generation - transistors - solid state device made from silicon, but single gate each
10,000 to 100,000 transistors soldered to circuit board
Advantages: improved speed, reliability, size, power consumption
Some system software: early OS and High-level programming languages

Computer Generations (continued)

3rd Generation - Small-scale integrated circuits (ICs)
Many gates on same wafer of silicon (chip) and connected to form circuits.
Advantages:
1) cheaper since the cost per chip the same, but fewer needed since more powerful
2) denser chips → shorter connections → faster
3) smaller computers → used in more places
4) reduced power and cooling consumption
5) interconnections on ICs more reliable that interconnections between ICs

4th Generation - Large-scale ICs - microprocessor
Enough gates per chip to implement whole CPU
Intel 4004 (1971) - 1st
Intel 8080 (1974) - lst general-purpose microprocessor

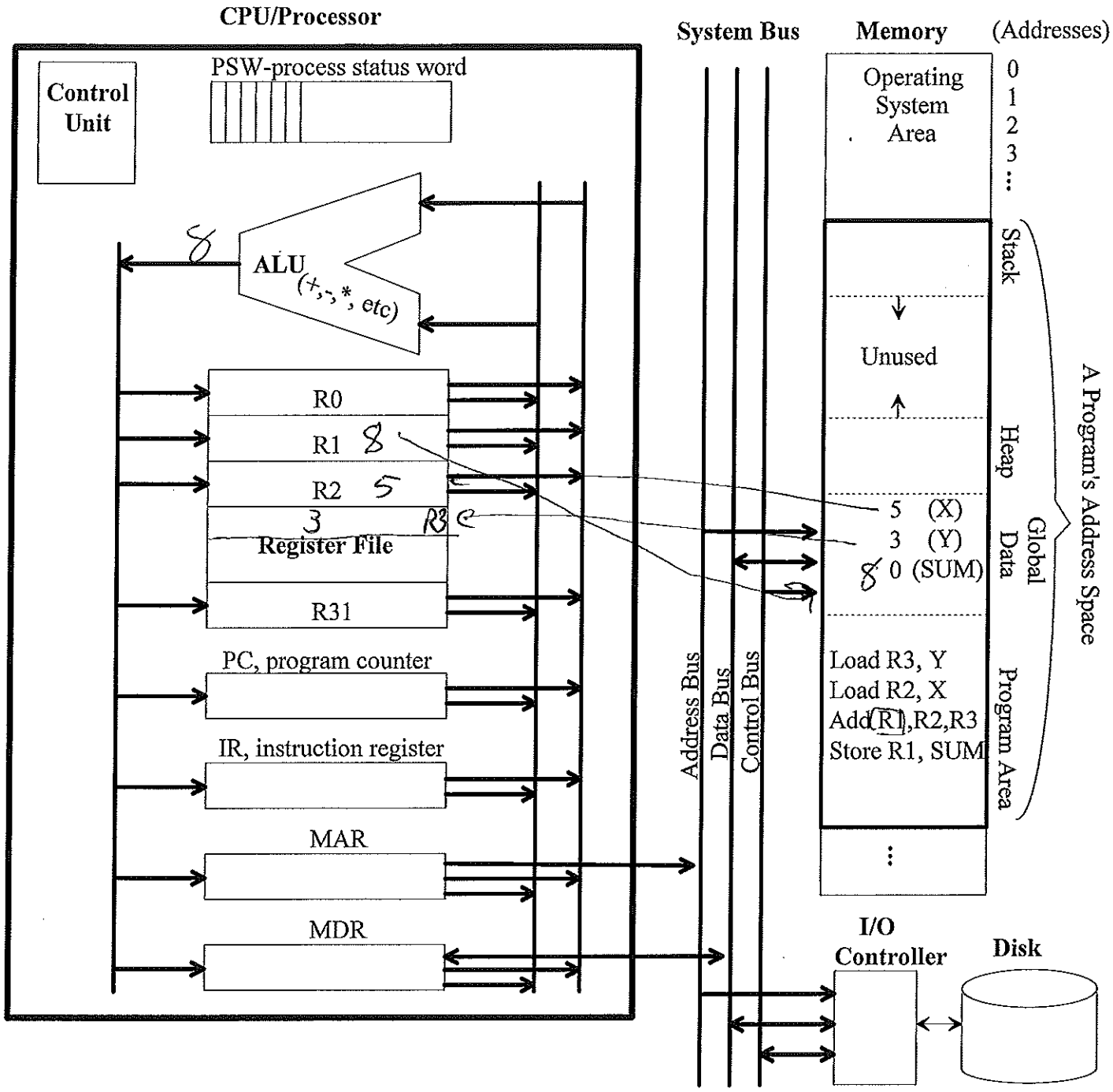
Moore's Law

Gordon Moore - Intel cofounder (1965)

Predicted that gate density would double every year into near future.

This held true until early 1970s when the rate slowed to doubling every 18 months.

von Neumann Model: Stored-Program Concept



Instruction/Machine Cycle of stored-program computer - repeat all day

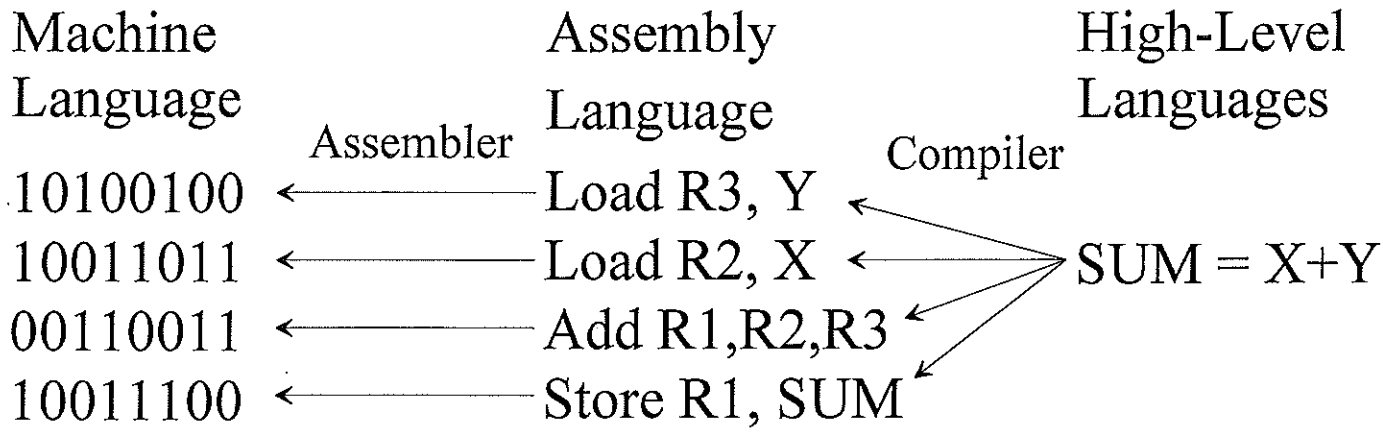
1. Fetch Instruction - read instruction pointed at by the program counter (PC) from memory into Instr. Reg. (IR)
2. Decode Instruction - figure out what kind of instruction was read
3. Fetch Operands - get operand values from the memory or registers
4. Execute Instruction - do some operation with the operands to get some result
5. Write Result - put the result into a register or in a memory location

(Note: Sometime during the above steps, the PC is updated to point to the next instruction.)

Today's stored-program computers have the following characteristics:

- –Three hardware systems:
  - •A central processing unit (CPU)
  - •A main memory system
  - •An I/O system
- –The capacity to carry out sequential instruction processing.
- –A single data path between the CPU and main memory.
  - •This single path is known as the *von Neumann bottleneck.*
  - •Register File - store a small amount of data in the CPU close to the computation circuits
  - •Level 1 (L1) and Level 2 (L2) caches on CPU store larger amounts of data and instruction on CPU

Programming Languages



Instruction Set Architecture (ISA)/Family of Computers
Several implementation of computer can run the same assembly/machine language with different cost/performance ratios.
Examples:
IBM 700/7000 Series (overhead)
Intel 486, Pentium I, II, III, IV (overhead)

Advantages of High-Level Programming Languages:
1) Portability - computer independent
2) Productivity - more and better software in less time
3) Application specific languages

Computer Networking

1962 - RAND (Research and Development a nonprofit institution) and Paul Baran begin research into packet switching.

1965 - ARPA - Advanced Research Projects Agency sponsors networking research

1969 - ARPA commissioned a Cambridge, Mass. company (Bolt, Beranek, and Newman) to build first packet switches Later, four computers connected

.

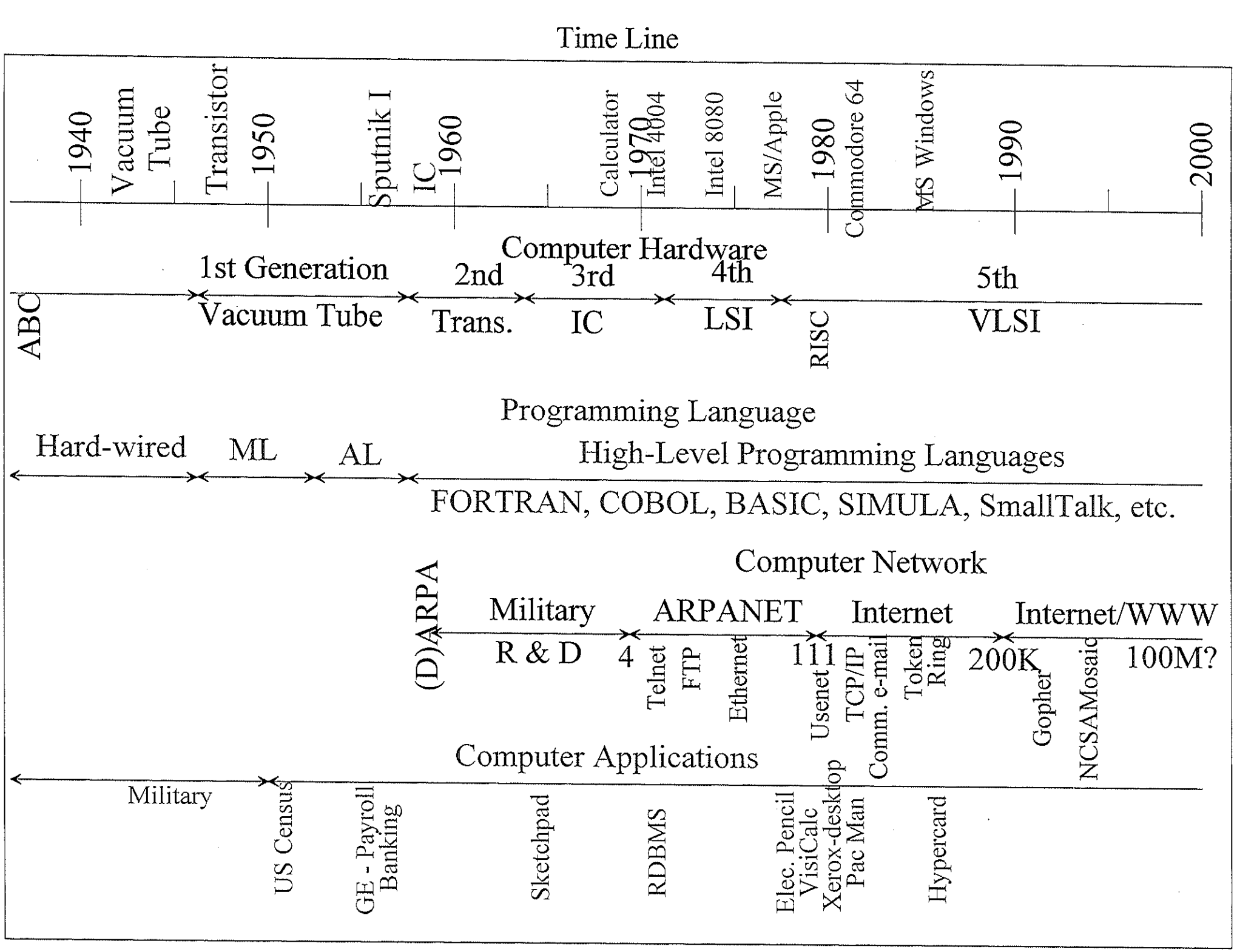
.

.

Soon networking bandwidth will be unlimited???

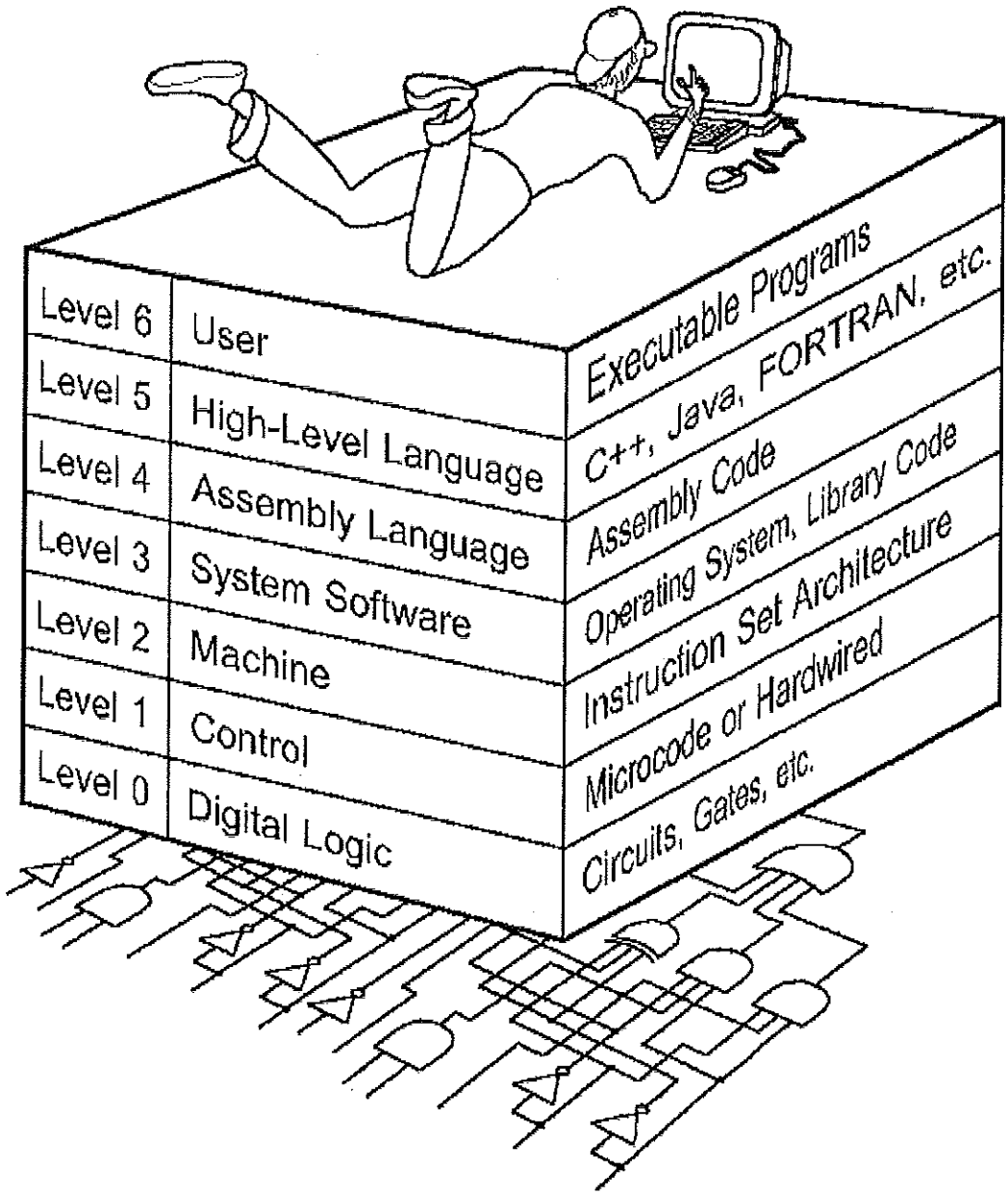
Computer Applications

Where are applications going???


Time Line
1940
Vacuum Tube
Transistor
1950
Sputnik I
IC
1960
Calculator
1970
Intel 4004
Intel 8080
MS/Apple
1980
Commodore 64
MS Windows
1990
2000
Computer Hardware
1st Generation
2nd
3rd
4th
5th
ABC
Vacuum Tube
Trans.
IC
LSI
RISC
VLSI
Programming Language
Hard-wired
ML
AL
High-Level Programming Languages
FORTRAN, COBOL, BASIC, SIMULA, SmallTalk, etc.
Computer Network
(D)ARPA
Military
ARPANET
Internet
Internet/WWW
R & D
4
111
200K
100M?
Telnet
FTP
Ethernet
Usenet
TCP/IP
Comm. e-mail
Token Ring
Gopher
NCSAMosaic
Computer Applications
Military
US Census
GE - Payroll
Banking
Sketchpad
RDBMS
Elec. Pencil
VisiCalc
Xerox-desktop
Pac Man
Hypercard

The Computer Level Hierarchy

- Computers consist of many things besides chips.
- Before a computer can do anything worthwhile, it must also use software.
- Writing complex programs requires a "divide and conquer" approach, where each program module solves a smaller problem.
- Complex computer systems employ a similar technique through a series of virtual machine layers.



The machines at each level execute their own particular instructions, calling upon machines at lower levels to perform tasks as required.

- Level 6: The User Level
  - –Program execution and user interface level.
  - –The level with which we are most familiar.

- Level 5: High-Level Language Level
  - –The level with which we interact when we write programs in languages such as C, Pascal, Lisp, and Java.

- Level 4: Assembly Language Level
  - –Acts upon assembly language produced from Level 5, as well as instructions programmed directly at this level.

- Level 3: System Software Level
  - –Controls executing processes on the system.
  - –Protects system resources.
  - –Assembly language instructions often pass through Level 3 without modification.

- Level 2: Machine Level
  - –Also known as the Instruction Set Architecture (ISA) Level.
  - –Consists of instructions that are particular to the architecture of the machine.
  - –Programs written in machine language need no compilers, interpreters, or assemblers.

- Level 1: Control Level
  - –A *control unit* decodes and executes instructions and moves data through the system.
  - –Control units can be *microprogrammed* or *hardwired*.
  - –A microprogram is a program written in a low-level language that is implemented by the hardware.
  - –Hardwired control units consist of hardware that directly executes machine instructions.

- Level 0: Digital Logic Level
  - –This level is where we find digital circuits (the chips).
  - –Digital circuits consist of gates and wires.
  - –These components implement the mathematical logic of all other levels.

1) Complete the following table.

	Decimal (Base 10)	Binary (Base 2)	Hexadecimal (Base 16)
Number of digits:	10		
Digits:	0, 1, 2, 3, 4, 5, 6, 7, 8, 9		
Counting:	0		
	1		
	2		
	3		
	4		
	5		
	6		
	7		
	8		
	9		
	10		
	11		
	12		
	13		
	14		
	15		
	16		
	17		

3. Convert 375_{10} to a binary (base 2) value.

4. Convert 375_{10} to a hexadecimal (base 16) value.

5. Convert $2BA_{16}$ to a decimal (base 10) value.

6. Perform the following arithmetic operations:

$$\begin{array}{rr} 1001010_2 & 1100010_2 \\ +1101110_2 & -\ 1001011_2 \end{array} \qquad \begin{array}{rr} & C\ B\ 3\ 1\ A_{16} \\ + & 7\ 3\ A\ 1\ 8_{16} \end{array} \qquad \begin{array}{rr} & A\ 1\ 9\ D\ 1_{16} \\ - & 4\ A\ 7\ 3_{16} \end{array}$$

Options for representing signed integers: 8-bit example for -19_{10}

a) signed magnitude:

sign bit	magnitude
1	0 0 1 0 0 1 1

positive is 0
negative is 1

b) one's complement: positive values are their binary #. For negative values, invert all the bits of binary # of the absolute value

abs(-19) = +19 = 0 0 0 1 0 0 1 1
-19 = 1 1 1 0 1 1 0 0 ← Invert bits to get one's complement

c) two's complement: positive values are their binary #. For negative values, invert all the bits of binary # of the absolute value, then add 1

abs(-19) = +19 = 0 0 0 1 0 0 1 1
1 1 1 0 1 1 0 0 ← Invert bits to get one's complement, then
+ 1 ← Add 1 to get two's complement
-19 = 1 1 1 0 1 1 0 1

7. Represent the following decimal numbers in binary using **8-bit** signed magnitude, one's complement, and two's complement:

decimal number	**signed magnitude 8-bits**	**one's complement 8-bits**	**two's complement 8-bits**
97_{10}			
-45_{10}			

8. Using 8-bits what is the range of values for each of the following representations:

a) unsigned integers:

b) signed integers using two's complement:

Lecture 2 Video

Counting

0
1
2
⋮
9
10
11
12
⋮
19
20

10^3 10^2 10^1 10^0
1000 100 10 1
0 0 0 9.

0 0 1 0

0 0 1 1

1 2

0 0 1 9

0 0 2 0

Binary

0, 1

Counting

0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0

2^4 2^3 2^2 2^1 2^0
16 8 4 2 1 ← Decimal

8 4 2 1 :
$0\,1\,1\,0\,1_2 = 13_{10}$

$8+4+1 = 13_{10}$

Convert: 32 16 8 4 2 1
$10100_2 = ?_{10}$

$32+8+1 = (41_{10})$

$86_{10} = ?_2$

256 128 64 32 16 8 4 2 1
$0\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0_2$

86
−64
22
−16
6

Hexadecimal

Digits: $0, 1, 2, \ldots, 9, A, B, C, D, E, F$

Counting:

0
1
⋮
9
A
B
C
D
E
F

16^3 16^2 16^1 16^0

$000F = 15_{10}$

0010_{16}

0011_{16}

⋮

$001F_{16}$

0020_{16}

⋮

$00FF$

0100_{16}

Why hex.?

Base 2 $\longleftrightarrow$ Base 16

16^2 16^1 16^0
16 1

$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$86_{10} = 0101\,0110_2$
8 4 2 1

$2^4(2^3\ 2^2\ 2^1\ 2^0)$ 6_{16}

$16^1 = 2^4(8\ 4\ 2\ 1)$

5 6_{16}

8 4 2 1 8 4 2 1

$0000\,1010\,1010_2$

13_{10}

0 D 5_{16}

Hex	Dec
A	10
B	11
C	12
D	13
E	14
F	15

$3\ E\ A_{16} = ?_2$

8421 8421 8421

$0011\,1110\,1010_2$

Base 10 $\longleftrightarrow$ Base 2 $\longleftrightarrow$ Base 16

Binary

$$\begin{array}{r} 0\,1\,1\,0\,1\,0\,0\,1_2 \\ +\ 0\,1\,0\,0\,1\,0\,1 \\ \hline 1\,0\,0\,0\,1\,1\,1\,0_2 \end{array}$$

$$\begin{array}{r} 1\,1\,0\,\overset{0\to 2}{\cancel{1}}\,\cancel{0}\,0\,1_2 \\ -\ 0\,1\,0\,0\,1\,0\,1 \\ \hline 1\,1\,0\,0\,1\,0\,0 \end{array}$$

Decimal:

$$\begin{array}{r} 2\,3\,4_{10} \\ +\ 1\,8\,6_{10} \\ \hline 4\,2\,0_{10} \end{array}$$

$$\begin{array}{r} \overset{1\to 12}{1}\,\overset{2\to 14}{2}\,\cancel{3}\,4_{10} \\ -\ 1\,8\,6_{10} \\ \hline 0\,4\,8_{10} \end{array}$$

$1+11+10 = 22_{10} - 16_{10} = 6_{10}$

$8+9 = 17_{10} - 16_{10} = 1_{10}$

$$\begin{array}{r} 3\,B\,8_{16} \\ +\ 1\,A\,9_{16} \\ \hline 5\,6\,1_{16} \end{array}$$

A	10_{10}
B	11_{10}
C	12_{10}
D	13
E	14
F	15

$A \to 16_{10}$ $\quad 8+16 = 24_{10} - 9_{10} = 15_{10}$

$$\begin{array}{r} 3\,\cancel{B}\,8_{16} \\ -\ 1\,A\,9_{16} \\ \hline 2\,0\,F_{16} \end{array}$$

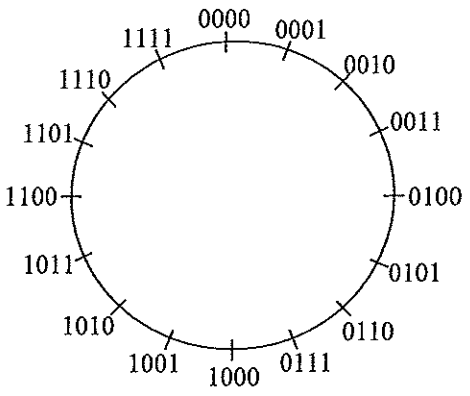
1. Consider 4-bit BINARY numbers, because of "roll-over" the number line wraps around.

a) List unsigned decimal values on the outside

b) List signed (two's complement) decimal values on the inside

c) Mark the point of unsigned overflow

d) Mark the point of signed overflow



Perform the following additions:

e) for unsigned numbers:

0100_2 (4_{10}) + 0110_2 (6_{10})

1001_2 (9_{10}) + 1010_2 (10_{10})

f) for signed numbers: (two's compliment)

0100_2 (4_{10}) + 0110_2 (6_{10})

0100_2 (4_{10}) + 1010_2 (-6_{10})

1100_2 (-4_{10}) + 1010_2 (-6_{10})

2. For *4-bit unsigned numbers*, when do we have overflow and get the wrong result during addition? (Hint: think about the carry bits into and/or out of the most-significant bit)

3. a) For *4-bit signed numbers*, complete the following table about signed overflow:

Sign of Operands for addition		Expected Sign of Result	Wrong Sign of Result (indicates overflow)
Operand 1	Operand 2		
+	+		
+	−	These two rows cannot cause signed overflow in addition	
−	+		
−	−		

b) For *4-bit signed numbers*, when do we have overflow and get the wrong result during addition? (Hint: think about the carry bits into and/or out of the most-significant bit)

4. Use Booth's algorithm to calculate the 8-bit product of $0110_2 \times 1101_2$.

Multiplicand

0 1 1 0

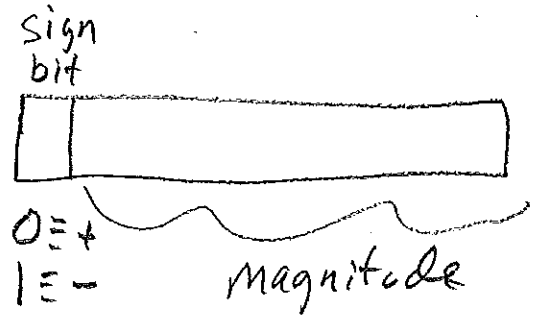
– Multiplicand

1 0 1 0

"Initial Product"	"Multiplier"	"Previous bit"
0 0 0 0	1 1 0 1	0

Lecture 3 Video

Sign-Magnitude



-19_{10} 8-bits

	64	32	16	8	4	2	1
1	0	0	1	0	0	1	1

$$\begin{array}{r} 19 \\ -16 \\ \hline 3 \end{array}$$

10010011

$+23_{10}$

0	0	0	1	0	1	1	1

Range with 8-bits

128 64 1 1

0 1 1 1 1 1 1 1 $= +127_{10}$

+1

128 = 1 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1

−127

8-bit unsigned: $0000\ 0000_2 \Rightarrow 0_{10}$

256 128 64

$1111\ 1111_2 \Rightarrow +255_{10}$

+1

1 0 0 = +256 Lec 3-(1)

one's complement

0 or +: normal binary #

− : ① start with + value

② Flip all bit 0 → 1
1 → 0

8-bit -19_{10}

128 64 32 16 8 4 2 1

① $+19 = 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1$

② $1\ 1\ 1\ 0\ 1\ 1\ 0\ 0$

↑ "sign bit"

$+23_{10}$:

128 64 32 16 8 4 2 1

$0\ 0\ 0\ 1\ 0\ 1\ 1\ 1$

↑ sign bit

---

two's complement:

0 or +: normal binary #

− : ① start with + value
② flip bits
③ add 1

8-bit -19_{10}:

128 64 32 16 8 4 2 1

$0\ 0\ 0\ 1\ 0\ 0\ 1\ 1$

$1\ 1\ 1\ 0\ 1\ 1\ 0\ 0$

$+1$

neg. → $1\ 1\ 1\ 0\ 1\ 1\ 0\ 1$

two's complement:

-19_{10} 1 1 1 0 1 0 1

↑ neg.

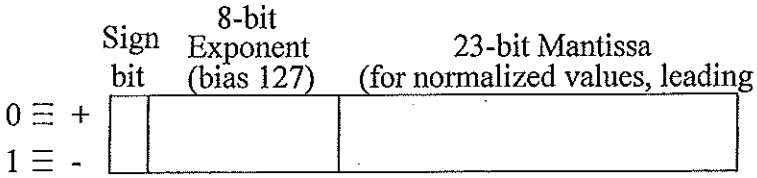
① flip bits: 0 0 0 1 0 0 1 0

② add 1: +1

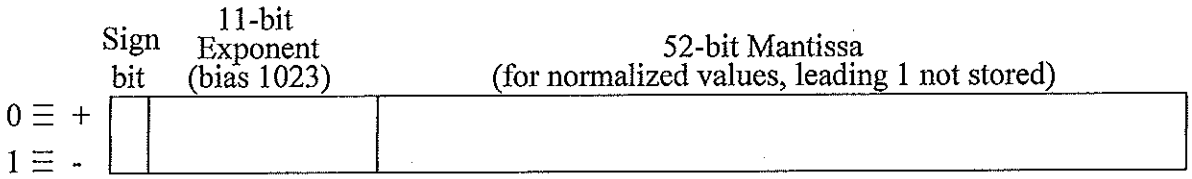
0 0 0 1 0 0 1 1

16 8 4 2 1 = +19

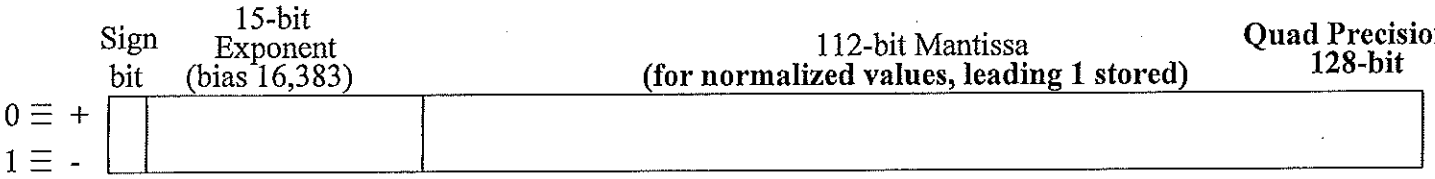
IEEE 754 Standard Floating Point Representation



Single Precision 32-bit



Double Precision 64-bit



Quad Precision 128-bit

Single Precision		Double Precision		Object
Exponent	**Mantissa**	**Exponent**	**Mantissa**	**Represented**
1-254	any value	1-2046	any value	normalized #
0	0	0	0	0
0	nonzero	0	nonzero	denormalized #
255	0	2,047	0	infinity
255	nonzero	2,047	nonzero	NaN (not a #)

1) Convert the value 23.625_{10} to its binary representation.

64 32 16 8 4 2 1 . .5 .25 .125 .0625

2) Normalize the above value so that the most significant 1 is immediately to the left of the radix point. Include the corresponding exponent value to indicate the motion of the radix point.

$1.\square \times 2^{\square}$

3) Write the corresponding 32-bit IEEE 754 floating point representation for 23.625_{10}.

4) Write the corresponding 128-bit IEEE 754 floating point representation for 23.625_{10}.

5) How would you add two normalized IEEE 754 floating point numbers?

6) Consider adding 1.011×2^{40} and 1.01×2^{5}.

a) How many places does the second number's mantissa get shifted?

b) After we add these two numbers and store the results back into a 32-bit IEEE 754 value, what would be the result?

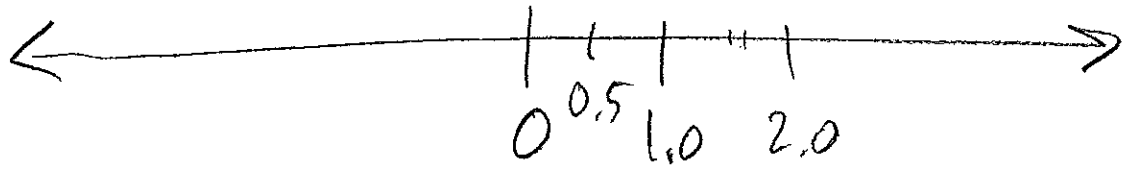
7) How would you multiply two normalized IEEE 754 floating point numbers?

8) What would be the smallest positive normalized 32-bit IEEE 754 floating point value?

9) What would be the largest positive denormalized 32-bit IEEE 754 floating point value?
(denormalized: exponent is still bias 127, but no implied "1." (i.e., "leading 1 not stored"), but "0." instead)

10) What would be the smallest positive denormalized 32-bit IEEE 754 floating point value?

Lecture 4 Floating Point

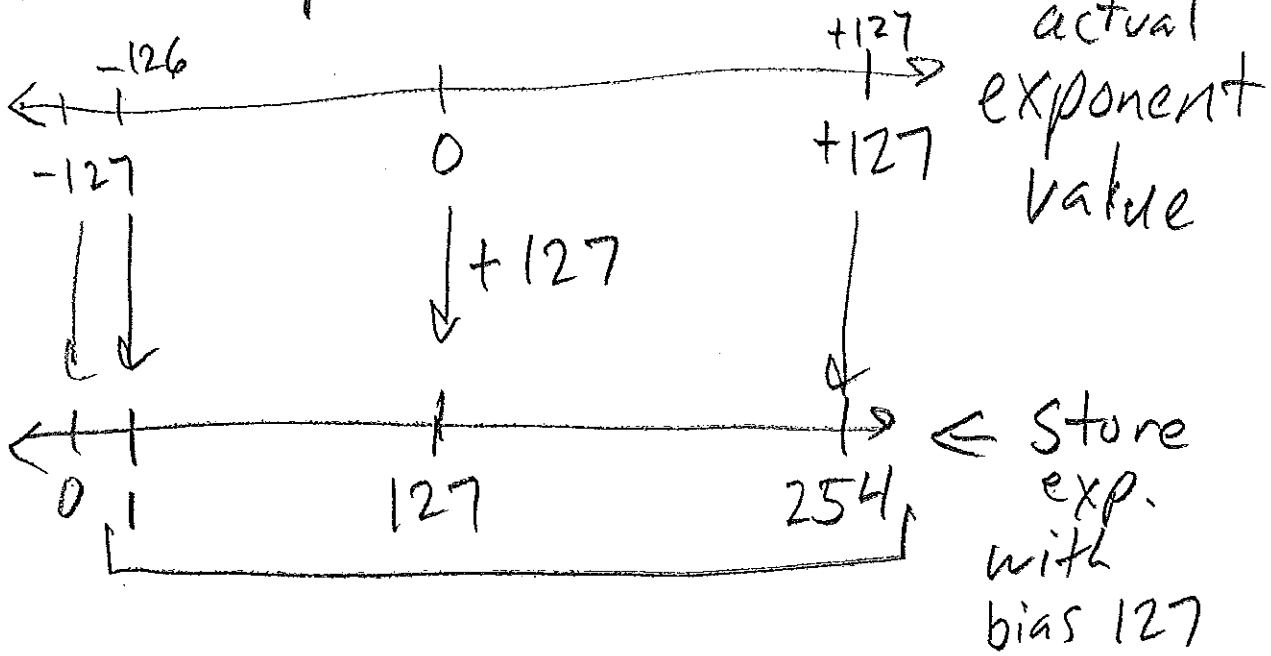


Decimal.

$23,456_{10} = 2.3456 \times 10^{1}$

2345.6×10^{-2}

8-bits exponent: $2^8 = 256$ values



73.3125_{10}

radix

64 32 16 8 4 2 1 . .5 .25 .125 .0625

1 0 0 1 0 0 1 . 0 1 0 1

133
6+127
6+1023
1029

1.0 0 1 0 0 1 0 1 0 1 X 2

32-bit 128 64 32 16 8 4 2 1

| 0 | 1 0 0 0 0 1 0 1 | 0 0 1 0 0 1 0 1 0 1 0 0 0 ... 0 |

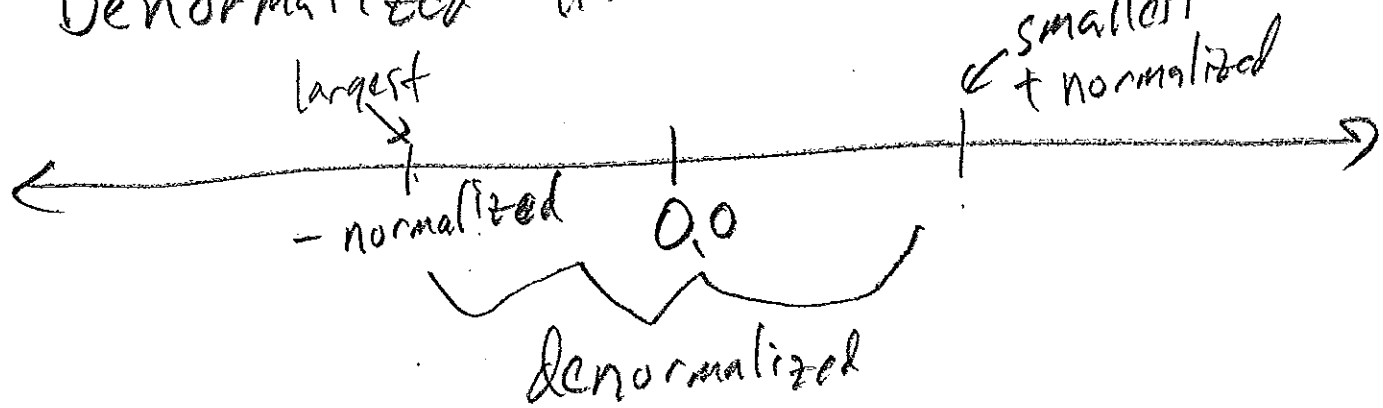
64-bit 1024 512 256 128 64 32 16 8 4 2 1

| 0 | 1 0 0 0 0 0 0 0 1 0 1 | 0 0 1 0 0 1 0 1 0 1 0 0 0 ... 0 |

128 bit

| 0 | 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 | 1 0 0 1 0 0 1 0 1 0 1 0 0 ... 0 |

Denormalized #s



ASCII Character Representation

0 NUL	16 DLE	32	48 0	64 @	80 P	96 `	112 p
1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
4 EOT	20 DC4	36 $	52 4	68 D	84 T	100 d	116 t
5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
7 BEL	23 ETB	39 '	55 7	71 G	87 W	103 g	119 w
8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
9 HT	25 EM	41)	57 9	73 I	89 Y	105 i	121 y
10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
11 VT	27 ESC	43 +	59 ;	75 K	91 [	107 k	123 {
12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124 |
13 CR	29 GS	45 -	61 =	77 M	93]	109 m	125 }
14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
15 SI	31 US	47 /	63 ?	79 O	95 _	111 o	127 DEL

Abbreviations

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell (beep)	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed, new line	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed, new page	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift In	US	Unit separator
		DEL	Delete/Idle

1) The ASCII code for character 'A' is 65_{10}, 'B' is 66_{10}, ... and 'a' is 97_{10}, 'b' is 98_{10}, ...

a) What would be the 7-bit binary value used to represent 'A'?

b) What would be the 7-bit binary value used to represent 'a'?

c) How does an upper-case letter differ from its corresponding lower-case letter?

d) *Even parity* prepends a 0 or 1 so as to make the total number of 1's be even. What is the 8-bit ASCII value for"

'A':

'a':

e) What error(s) cannot be detected by even parity?

2 a) For the 8-bit data 01001011_2 develop the Hamming codeword for one-bit error detection and correction:

12	11	10	9	8	7	6	5	4	3	2	1
D_7	D_6	D_5	D_4	P_8	D_3	D_2	D_1	P_4	D_0	P_2	P_1
0	1	0	0		1	0	1		1		
4+8	1+2+8	2+8	1+8	8	1+2+4	2+4	1+4	4	1+2	2	1

Check bit P_1 looks at bit positions 1, 3, 5, 7, 9, and 11
Check bit P_2 looks at bit positions 2, 3, 6, 7, 10, and 11
Check bit P_4 looks at bit positions 4, 5, 6, 7, and 12
Check bit P_8 looks at bit positions 8, 9, 10, 11, and 12

b) If bit D_5 gets flipped (an error), then how would we be able to detect an error?

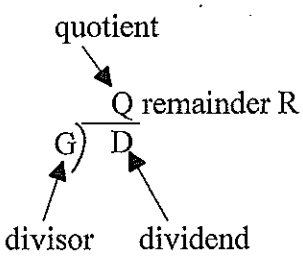
c) If bit D_5 gets flipped (an error), then how would we be able to know which bit to correct?

d) For the 8-bit data 11001001_2 develop the Hamming codeword for one-bit error detection and correction:

12	11	10	9	8	7	6	5	4	3	2	1
D_7	D_6	D_5	D_4	P_8	D_3	D_2	D_1	P_4	D_0	P_2	P_1
1	1	0	0		1	0	0		1		
4+8	1+2+8	2+8	1+8	8	1+2+4	2+4	1+4	4	1+2	2	1

3) The CRC, Cyclic Redundancy Check, are used to detect errors in long data transmissions/blocks that are subject to burst errors (several bits in sequence that are corrupted). The main idea is to append to the data-bits a small (16 or 32 bits) amount of information to help detect errors. Retransmission of the data is done if an error is detected.

The basic calculation of the CRC is integer division.



$$5\overline{)134} \quad 26 \text{ remainder } 4$$

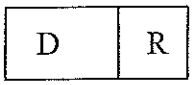
a) What is the quotient of (D - R) / G?

b) What is the remainder of (D - R) / G?

c) What is the range of the remainders of D / G?

Assume the sender and receiver agree on G and the sender sends:

If the receiver performs the calculation (D - R)/G, what would the remainder be if no transmission error occurred?



To simplify calculations all CRC calculations are done in modulo-2 arithmetic without carries in addition or borrows in subtraction. Therefore, both addition and subtraction are identical and equivalent to bitwise XOR (exclusive OR). For example,

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

$$\begin{array}{r} 1100 \\ -\,1010 \\ \hline 0110 \end{array} \qquad \begin{array}{r} 1100 \\ +\,1010 \\ \hline 0110 \end{array}$$

Let
D = d-bit data
R = n-bit remainder
G = degree n polynomial generator, e.g., G = $x^5 + x^2 + 1$ is 100101_2
C = (d+n)-bit codeword to be transmitted

The goal is to generate C such that C / G would have no remainder. To do this, we generate R as $\frac{D\times 2^n}{G}=Q\oplus\frac{R}{G}$, where Q is the quotient and R is the remainder.

The sender sends the codeword $C = D\times 2^n \oplus R$. When the receiver receives C, it divides by G, $\frac{C}{G}=\frac{D\times 2^n\oplus R}{G}=\frac{D\times 2^n}{G}\oplus\frac{R}{G}=Q\oplus\frac{R}{G}\oplus\frac{R}{G}=Q\oplus\frac{R\oplus R}{G}$.

e) What is $R \oplus R$?

Let
D = 10100101_2 (8-bit data),
G = $x^5 + x^2 + 1$ is 100101_2 (degree 5 polynomial)

f) Determine the remainder of $\frac{D\times 2^n}{G}$.

```
                    1
  1 0 0 1 0 1 ) 1 0 1 0 0 1 0 1 0 0 0 0 0
                1 0 0 1 0 1
                ---------
                    1 1 0 0
```

append n 0's since we start with $Dx2^n$

```
                    1 0 1
  1 0 0 1 0 1 ) 1 0 1 0 0 1 0 1 0 0 0 0 0
                1 0 0 1 0 1 ↓ ↓
                ---------
                    1 1 0 0 0 1
                    1 0 0 1 0 1
                    -----------
                      1 0 1 0 0
```

g) The codeword sent is the data appended with the remainder, so what codeword is sent by the sender in part f?

h) Divide the codeword by the generator $G = x^5 + x^2 + 1$ (100101_2) to check for an error. Remainder should be zero if no errors.

i) Introduce some random error into the codeword and check for an error by dividing by the generator $G = x^5 + x^2 + 1$ (100101_2)

Lecture 5

Characters - ASCII 7-bits 0_{10} - 127_{10}

A-Z contiguous 'A' = 65_{10}

a-z 'a' = 97

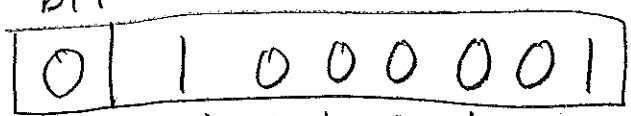
0-9

Unicode: 16-bit

'A' = 65_{10}

---

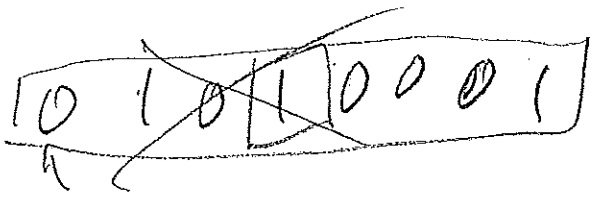
parity bit | 7-bit

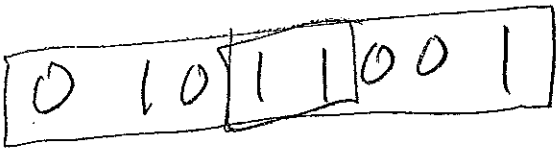


A = 65

even parity - total # of 1's is even

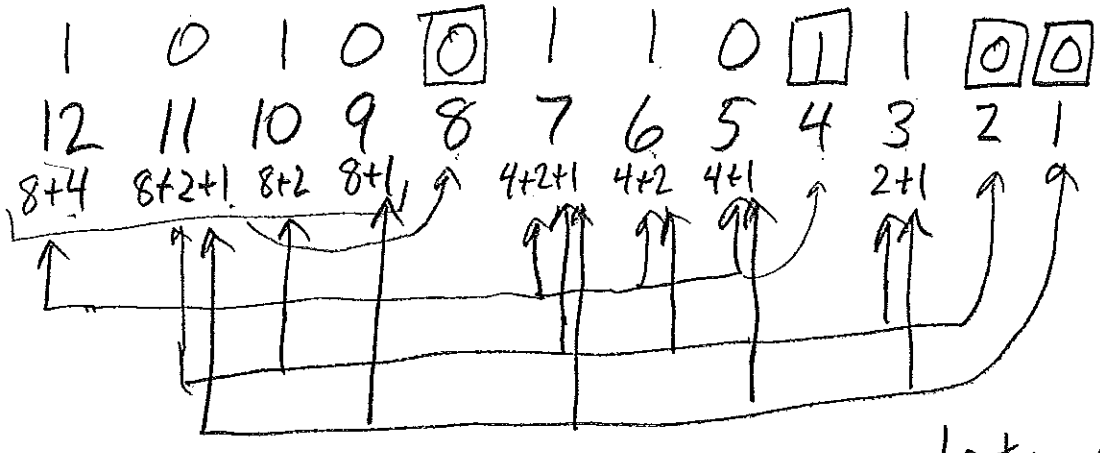
odd parity - total # of 1's is odd





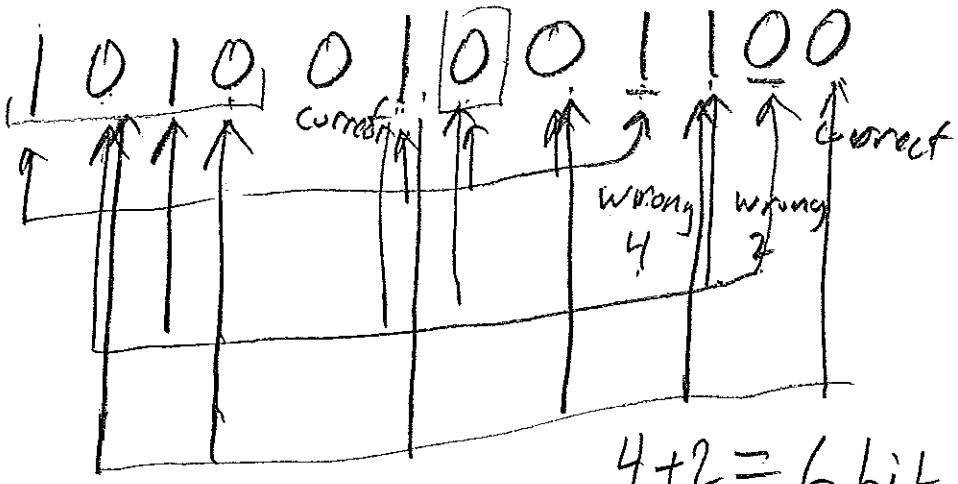
Hamming Code 8-bit data 10101101_2

even parity



Hamming Code Word:

12	11	10	9	8	7	6	5	4	3	2	1
1	0	1	0	0	1	1	0	1	1	0	0



4+2 = 6 bit incorrect

8-bit

$1010\ 1101_2$

1) Complete the truth tables for the logical gates and draw the gates.

OR

A	B	A+B
0	0	
0	1	
1	0	
1	1	

AND

A	B	A·B
0	0	
0	1	
1	0	
1	1	

NOT

A	$\overline{A}$
0	
1	

NOR

A	B	$\overline{A+B}$
0	0	
0	1	
1	0	
1	1	

NAND

A	B	$\overline{A \cdot B}$
0	0	
0	1	
1	0	
1	1	

(exclusive-OR) **XOR**

A	B	A⊕B
0	0	
0	1	
1	0	
1	1	

2) Draw the logic circuit using ANDs, ORs, and NOT gates for F = $\overline{A}\overline{B}C + A\overline{B}\overline{C} + \overline{A}BC + \overline{B}C$.

a) What is the complexity (sum of # inputs and # gates)?

b) How many gate delays for your circuit?

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0+x = x$
Null (or Dominance) Law	$0x = 0$	$1+x = 1$
Idempotent Law	$xx = x$	$x+x = x$
Inverse Law	$x\bar{x} = 0$	$x+\bar{x} = 1$
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy + xz$
Absorption Law	$x(x+y) = x$	$x+xy = x$
DeMorgan's Law	$(\overline{xy}) = \bar{x}+\bar{y}$	$(\overline{x+y}) = \bar{x}\bar{y}$
Double Complement Law	$\bar{\bar{x}} = x$	

General Simplification Technique

1) Look for two produces that have one variable negated

$$\ldots + X\bar{Y}\bar{Z} + \ldots + XY\bar{Z} + \ldots$$

2) "Factor out" common terms (distr. law)

$$\ldots + X\bar{Z}(\bar{Y} + Y) + \ldots$$

3) Apply inverse law

$$\ldots + X\bar{Z}\,1 + \ldots$$

4) Apply identity law

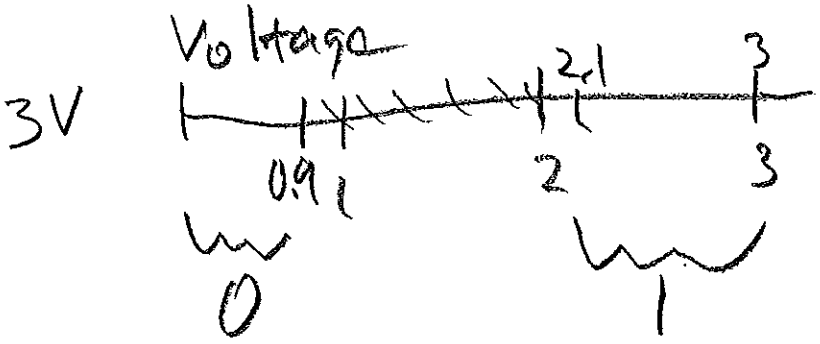
$$\ldots + X\bar{Z} + \ldots$$

3) Using Boolean Algebra simplify F= $\bar{A}\bar{B}C + A\bar{B}\bar{C} + \bar{A}BC + \bar{B}C$.

4) Draw the simplified logic circuit using ANDs, ORs, and NOT. What is the complexity (sum of # inputs and # gates)?

Lecture 6: Boolean logic and Gates

Digital / Binary : Boolean T, F
Binary 1, 0



Boolean algebra to describe functions for logical circuit

Gate: OR, AND, NOT, XOR, NOR, NAND

OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

AND

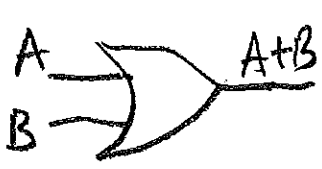
A	B	A·B
0	0	0
0	1	0
1	0	0
1	1	1

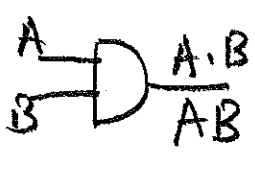
NOT

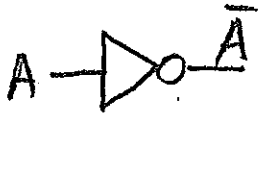
A	$\bar{A}$
0	1
1	0

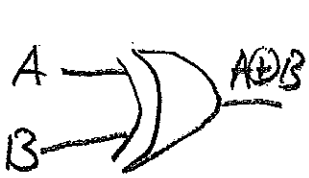
XOR

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0







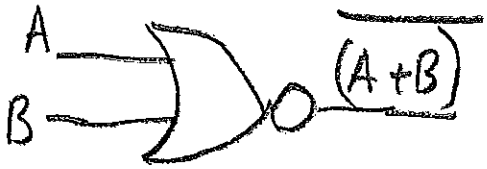


$((-x)\cdot y) + (z \cdot 3)$

NOT 1st
AND 2nd
OR 3rd

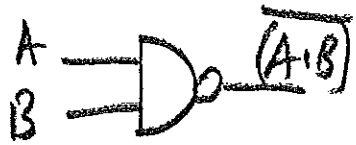
NOR

A	B	$\overline{(A+B)}$
0	0	1
0	1	0
1	0	0
1	1	0



NAND

A	B	$\overline{(A \cdot B)}$
0	0	1
0	1	1
1	0	1
1	1	0

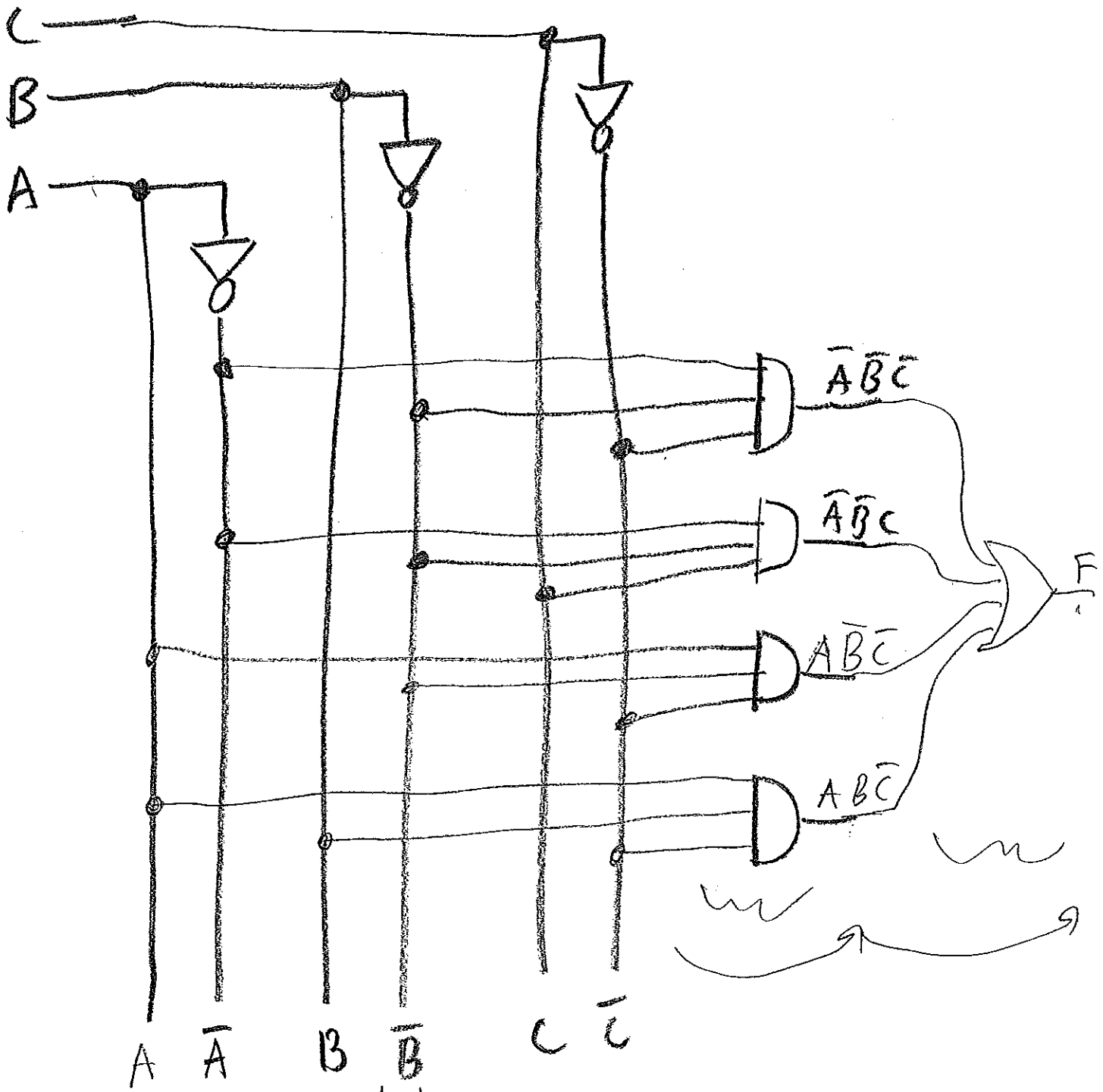


$$F = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,B\,C + A\,\overline{B}\,\overline{C} + A\,B\,\overline{C}$$

"Sum-of-Products" SOP

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

$$F = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}\bar{C} + ABC$$



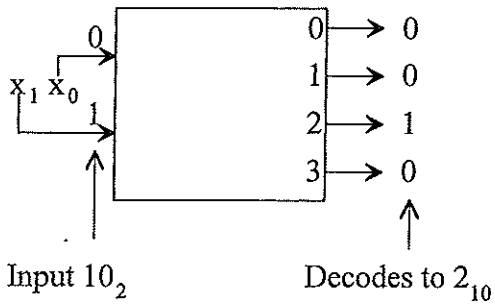
# of gate delay = 2

circuit complexity = $5 + (3\times4) + 4 = 21$

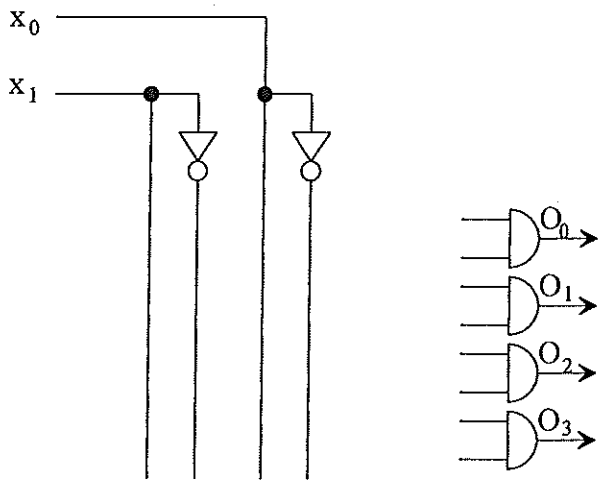
(#gates + #inputs ...)

1. A *decoder* takes an n-bit binary number (range of values 0_{10} to 2^n-1) and determines ("decodes") it's value by *asserting* (i.e., output of 1) only the output corresponding the binary number's value.

Consider a 2-to-4 decoder:



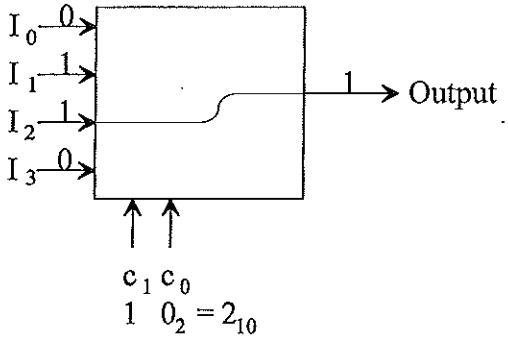
a) Connect input of AND-gates to generate the correct values.



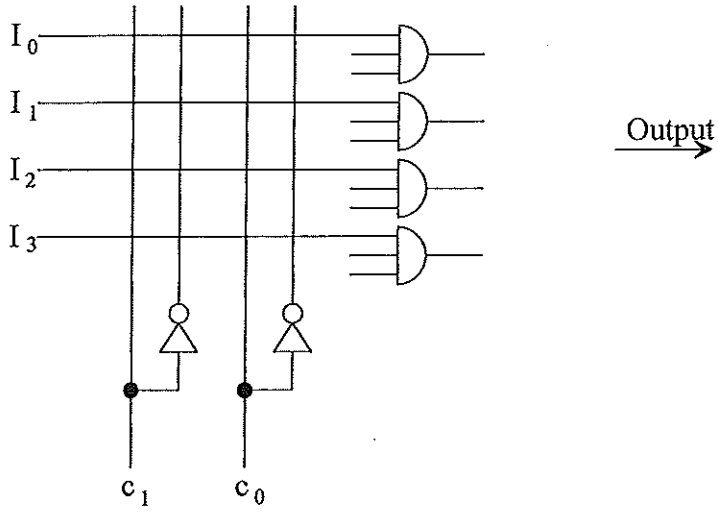
b) Draw a 3-to-8 decoder circuit below:

2. A *multiplexer* (MUX) allows one of 2^n inputs to be switched to a single output based on an n-bit binary number (range of values 0_{10} to 2^n-1) on the *control lines*.

Consider a 4-to-1 multiplexer:



a) Connect input of AND-gates and decide on how to merge AND-gate outputs into a single Output.



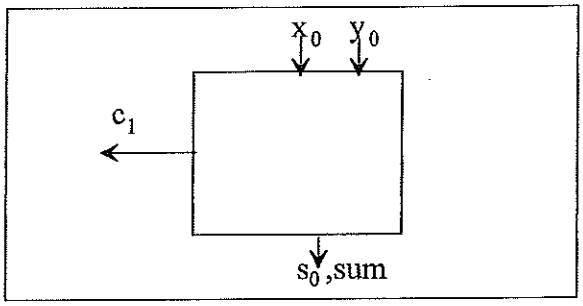
b) Draw a 8-to-1multiplexer circuit below:

3. Sum the following binary (base 2) numbers

$$\begin{array}{r} 10011_2 \\ +10110_2 \\ \hline \end{array} \qquad \begin{array}{r} 101101_2 \\ +110111_2 \\ \hline \end{array}$$

4. Complete the Half-Adder truth table and circuit diagram.

x_0	y_0	sum s_0	carry-out c_1
0	0		
0	1		
1	0		
1	1		



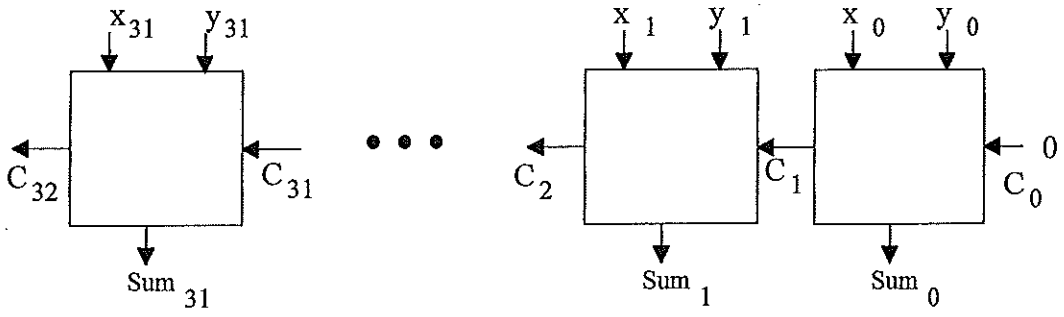
5. a) Complete the Full-Adder truth table for the sum (s_i) and carry-out (c_{i+1}) functions.

x_i	y_i	carry-in c_i	sum s_i	carry-out c_{i+1}
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

b) Use k-maps to minimize the sum (s_i) and carry-out (c_{i+1}) functions of the Full-Adder:

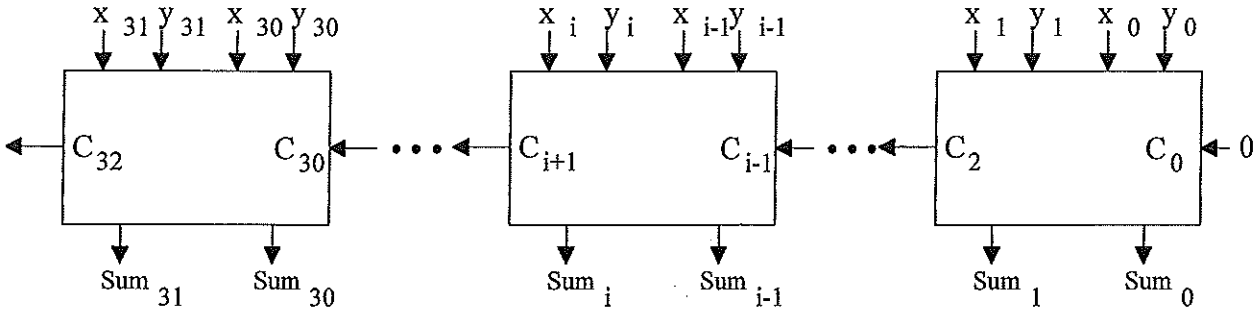
c) For the one-bit Full-Adder, how many gate delays are needed before the carry-out (c_{i+1}) wire is correct?

d) A 32-bit, ripple-adder is made up of a collection of single-bit Full-Adders connected together as shown below



e) How many gate delays are needed before c_{32} is correct?

6. To speed up the calculation of the carry-out (C_{i+1}) signals, consider constructing a 32-bit adder using two-bit adders as shown in:



a) If c_{i+1} is calculated directly from the inputs as $c_{i+1} = x_i y_i + x_i x_{i-1} y_{i-1} + x_i x_{i-1} c_{i-1} + x_i y_{i-1} c_{i-1} + y_i x_{i-1} y_{i-1} + y_i x_{i-1} c_{i-1} + y_i y_{i-1} c_{i-1}$, then how many gate delays would be needed to calculate the c_{i+1} signal in a two-bit adder?

b) What would be the total number of gate delays in a 32-bit adder before the c_{32} signal is generated correctly if <u>two-bit adders</u> were used?

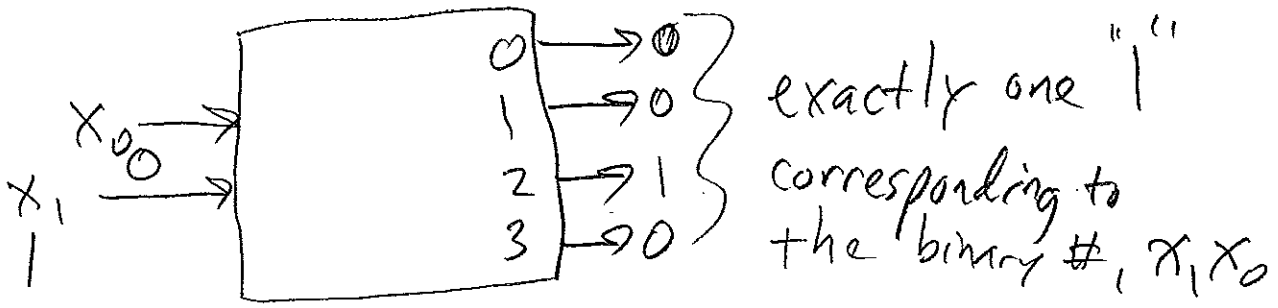
7. What would be the total number of gate delays in a 32-bit adder before the c_{32} signal is generated correctly if <u>three-bit adders</u> were used (10 three-bit adders and a 2-bit adder)?

8. What would be the total number of gate delays in a 32-bit adder before the c_{32} signal is generated correctly if <u>four-bit adders</u> were used?

Lecture 7 Common Circuits for Computers

Decoder n-inputs (n-bit binary #)

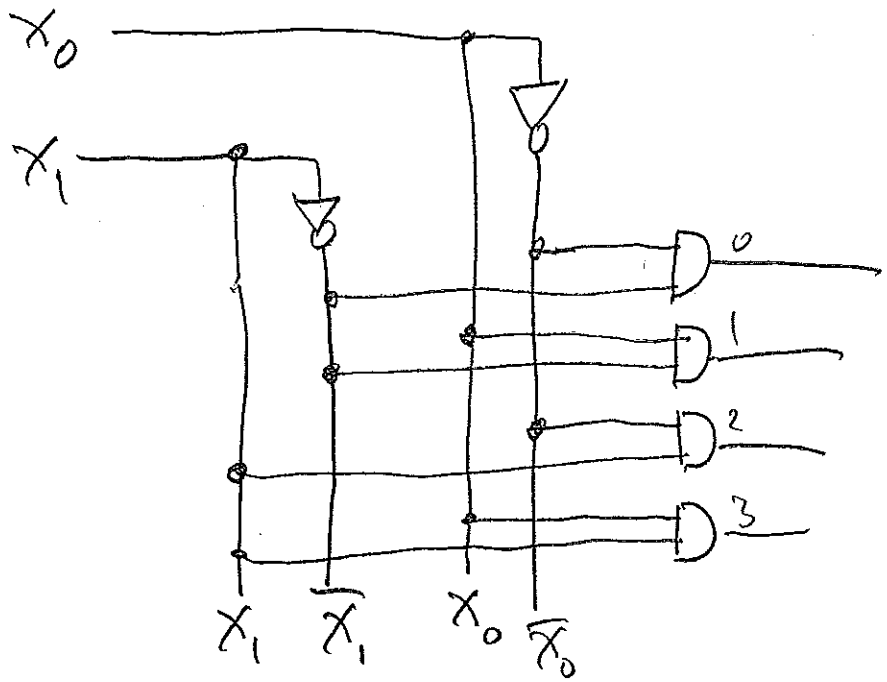
2-to-4 decoder



10_2

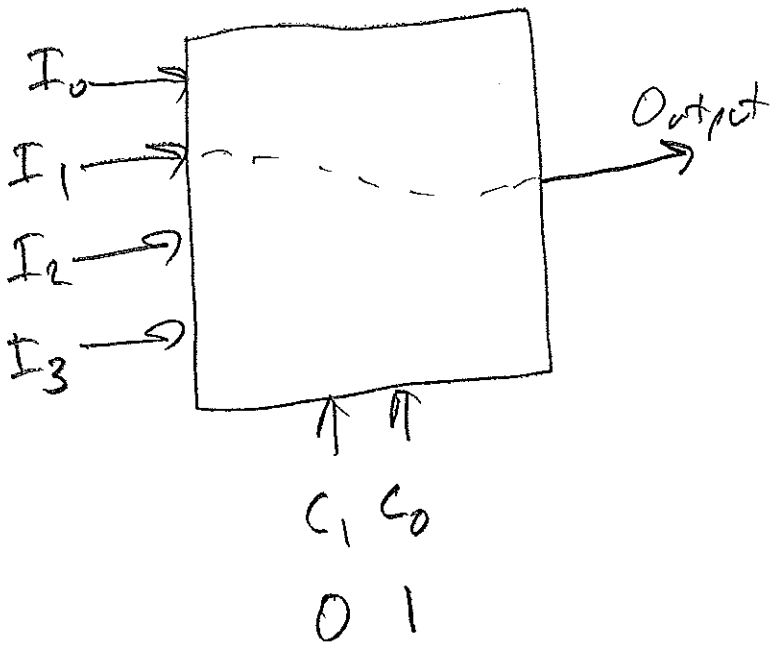
2_{10}

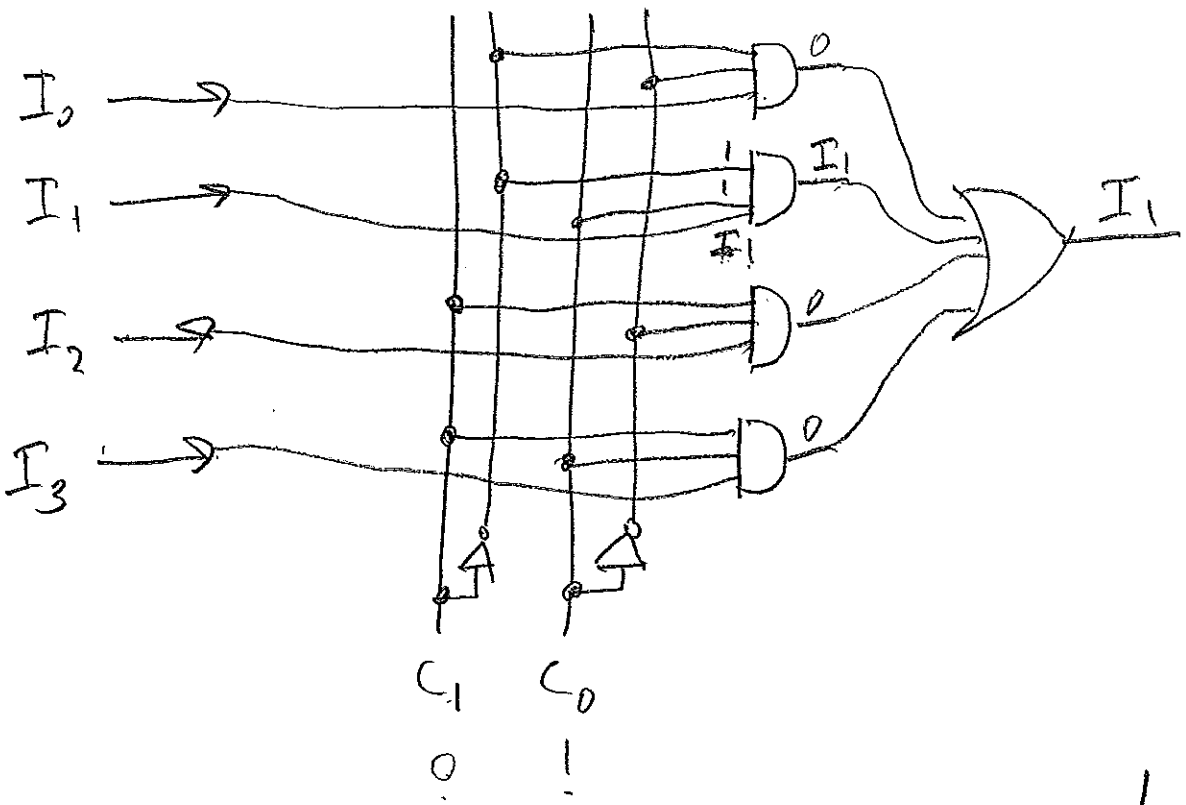
2-to-4 decoder



Multipler, MUX - swith one of the inputs to a single output based on a binary # on the control wires

4-to-1 MUX





Adder

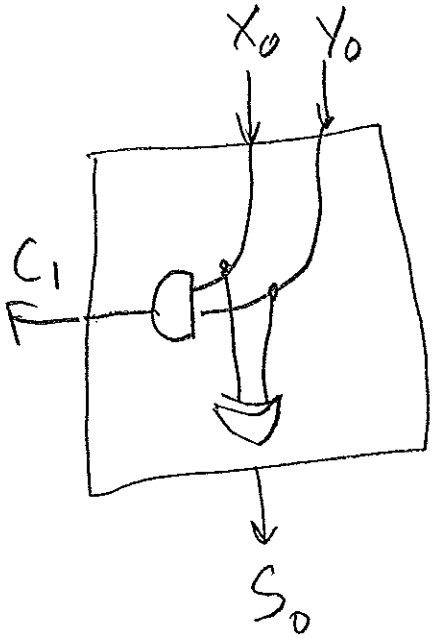
32-bit #s

$$
\begin{array}{cccccc}
 & & & c_{i+1} & c_i \text{ carry}_i & \\
X: & x_{31} & x_{30} & \cdots\ x_i & x_1 & \boxed{x_0} \\
Y: & y_{31} & y_{30} & \cdots\ y_i & y_1 & \boxed{y_0} \\
\hline
 & & & S_i & &
\end{array}
$$

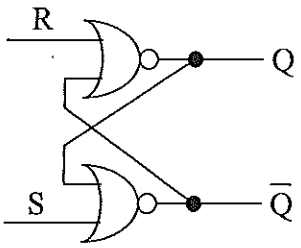
1-bit adder

half-adder (least-sign. bit only)

x_0	y_0	(sum) S_0	C_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1
		↑ XOR	↑ AND



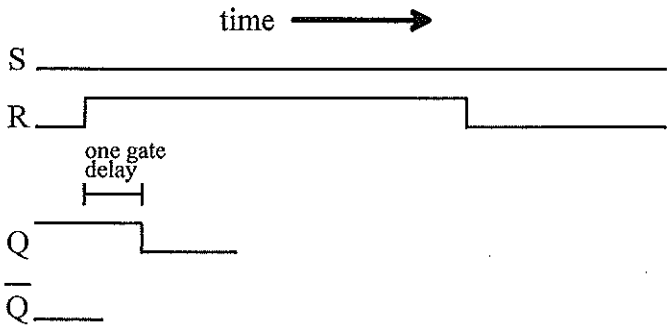
1. a) If R = 0 and S = 1, then what will be the output on Q and $\overline{Q}$?



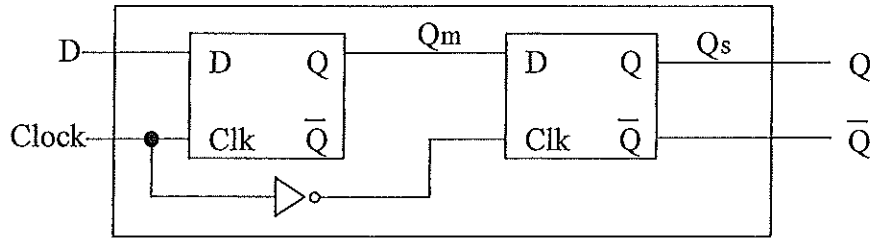
A	B	$\overline{(A+B)}$
0	0	1
0	1	0
1	0	0
1	1	0

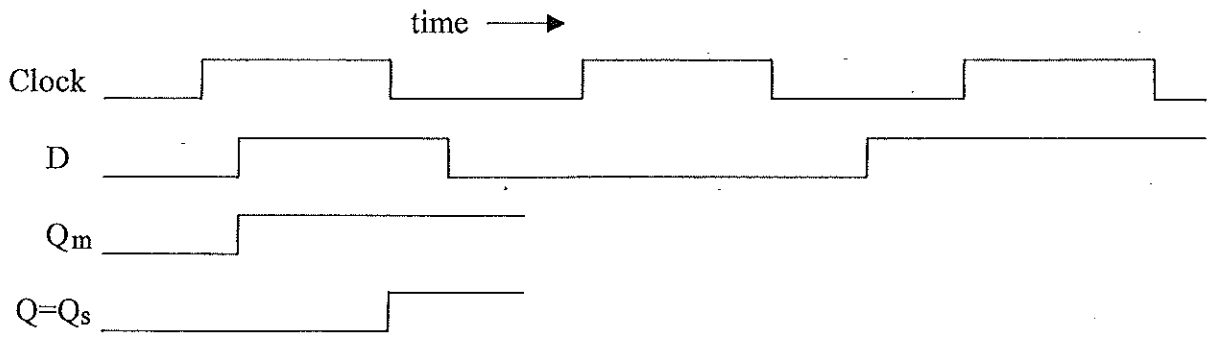
b) Now, if S goes to a 0 value, what happens to the output on Q and $\overline{Q}$?

c) Complete the following timing diagram for the SR latch:

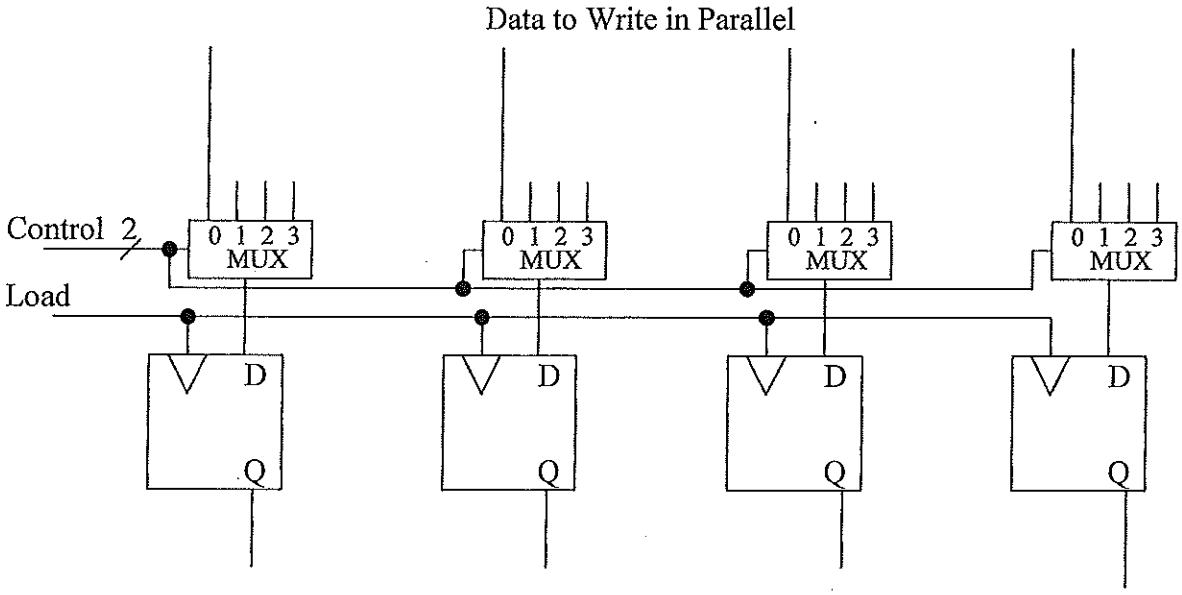


2. Complete the following timing diagram for the Master-Slave D flip-flop:





3. Complete the below diagram of a 4-bit register so that it is able to perform the following operations:
- parallel read/output of all bits (just look at the Q values)
- parallel write/input of all bits
- logical shift left one bit position (value shifted out of most-sign.bit is lost and a "0" is shifted into the least-significant bit)
- circular shift right one bit position (value shifted out of least-sign. bit is shifted into the most-significant bit)
- arithmetic shift right (sign-extend the most-significant bit)



4. Suppose we have a register file with the following specifications:
- 8 registers numbered from 0 to 7
- each register has 16-bits
- one write port
- two read ports

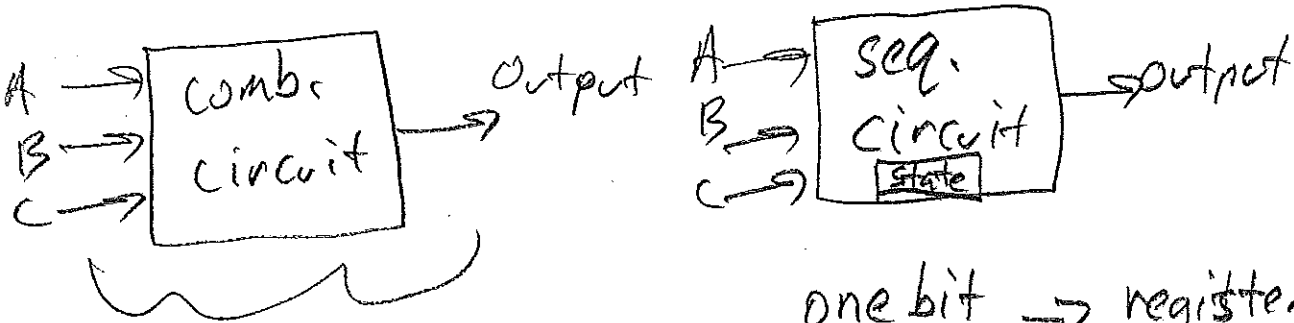
a) What how many bits(/"wires") would be need for each of the following?
- data to be written for a write port
- specifying the register number of a read port
- specifying the register number of a write port
- output read from a read port

b) How many write enable wires would be needed for the whole register file?

c) How many decoders would be needed in the implementation of the whole register file (like Figure 3.32)? Explain how you arrived at that number and specify the type of decoders (i.e., number of inputs and number of outputs for each decoder)

d) How many multiplexers would be needed in the implementation of the whole register file (like Figure 3.32)? Explain how you arrived at that number and specify the type of multiplexers.

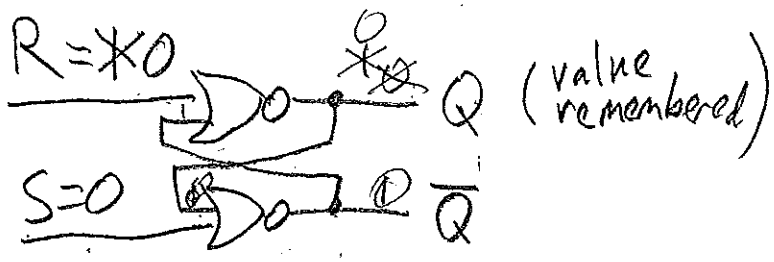
Lecture 8 - 1-bit SR-latch/memory



decoder, MUX, adder

one bit memory ⇒ register ⇒ reg. file

SR latch 1-bit memory



XOR gate

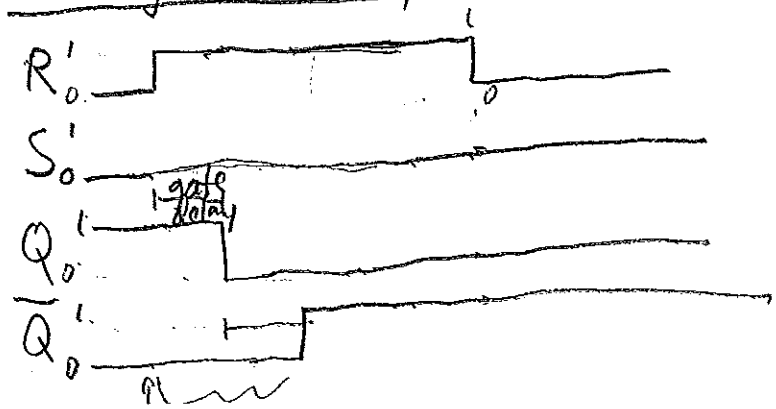
A	B	(A+B)
0	0	1
0	1	0
1	0	0
1	1	0

(*) If R change from 1 to 0, then keep remembering a 0.

R (Reset) to 0
S (set) to 1

} SR-latch remembers whether R or S had a "1" most recently.

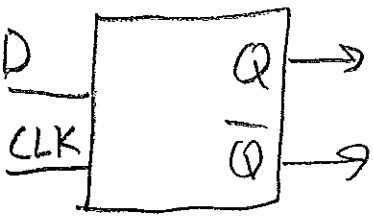
Timing diagram time →



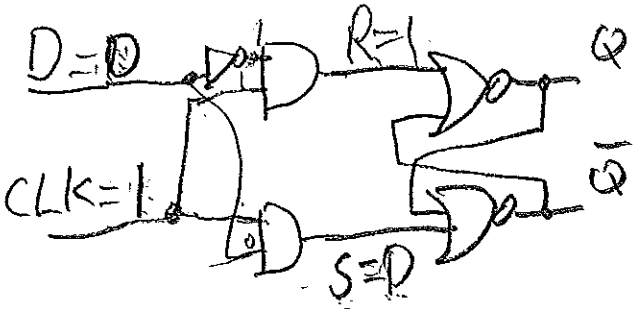
SR latch Characteristic Table

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	– don't do

D latch



When CLK=1, remember D's value;
otherwise CLK=0 - ignore D but keep remembering

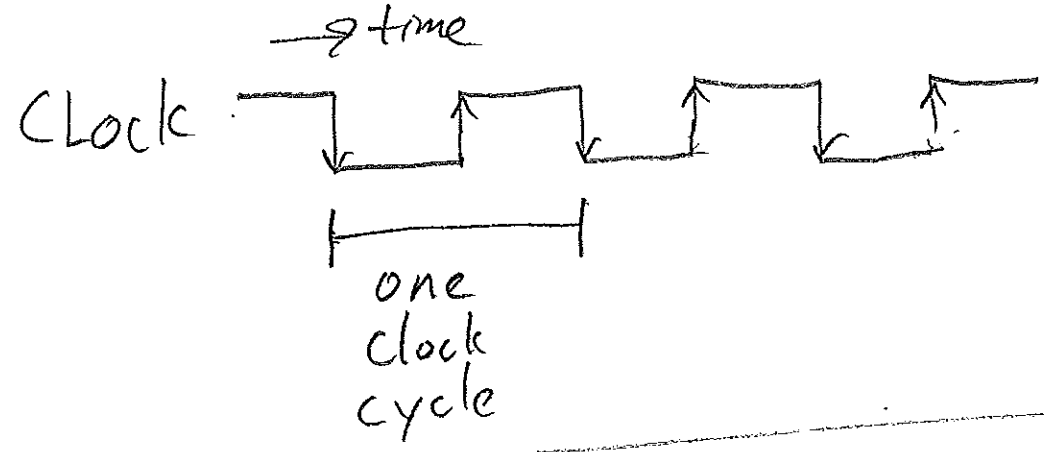


CLK	D	Q_{n+1}
0	0	Q_n
0	1	Q_n
1	0	0
1	1	1

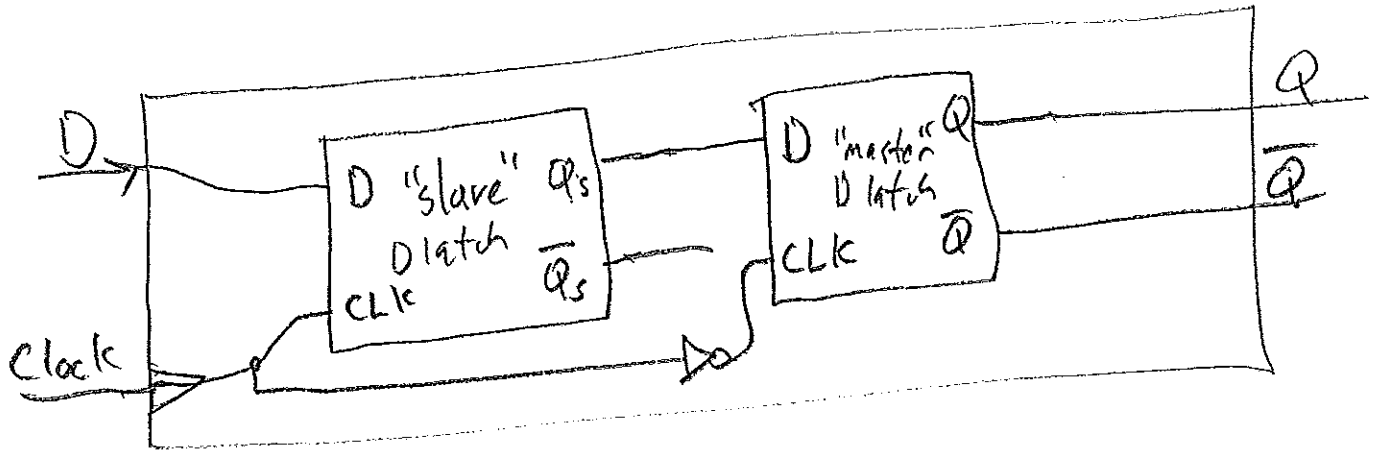
Flip-flops

1-bit memory only change state (not based on gate delays), but with respect to either the falling or rising edge of the clock pulse.

Clock – wire with a regular changing signal

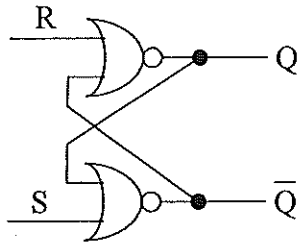


Master-Slave D flip-flop



"Memory" Supplement for Section 3.6 of the textbook

The most basic 1-bit memory is the *SR-latch* with consists of two cross-coupled NOR gates.



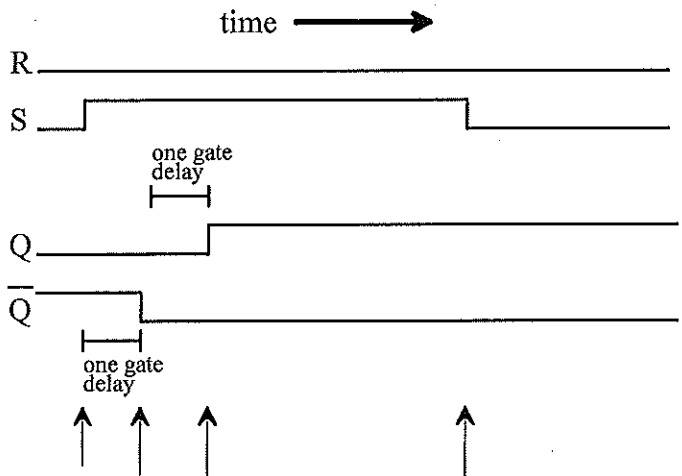
Recall the NOR gate truth table:

A	B	$\overline{(A+B)}$
0	0	1
0	1	0
1	0	0
1	1	0

The "S" stands for Set to remember 1, and the R for Reset to remember 0. For example, if the input values of R = 0 and S = 1 are held constant, then the bottom NOR gate will start outputting 0. This 0 is fed as an input to the upper NOR with the R = 0, so both of the upper NOR gate's inputs are 0. Thus, the upper NOR gate will start outputting a 1 which is fed back to the bottom NOR as an input. Now, the bottom NOR has an input of two 1's, but it continues to output a 0 because that's what NORs do (see truth table).

If the S input's value is now changed back to 0 (both R =0 and S=0), then bottom NOR will continue to output 0 (i.e., $\overline{Q} = 0$) since the top NOR is still outputting a 1. The top NOR has two 0 inputs, so it continues to output 1 (i.e., $Q = 1$). This is a stable state where the SR-latch is remembering a 1.

A timing diagram for the above scenario would look like:

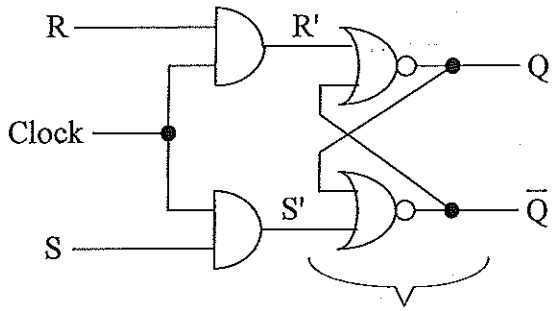


1. Initial state of R = 0 and S = 1
2. After a gate delay, the bottom NOR outputs a 0 which is fed to the upper NOR (the upper NOR has two 0s for inputs)
3. After a gate delay, the top NOR outputs a 1 which is fed to the bottom NOR (the bottom NOR has two 1s for inputs)
4. When S returns to 0, no change in outputs

The SR-latch is actual symmetric from top-to-bottom, so a similar argument can be made for the inputs of R = 1 and S = 0 causing a stable state of it remembering a 0.

The main problem with the SR-latch is that its output changes are determined by some number of gate delays after its inputs are changed. This makes it difficult for a collection of one-bit memories to work in a lock-step fashion to hold a multiple bit value. It also has the problem that inputs of R = 1 and S = 1 lead to an undefined state.

We can solve these problems in a series of steps. First we'll add a Clock input to prevent the memory from changing states if the Clock = 0. This leads us to a *clocked-SR latch*:



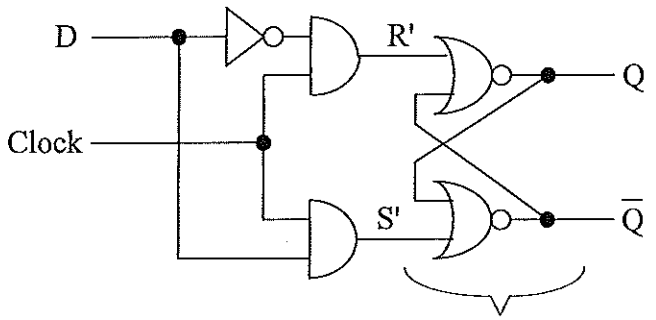
Original SR-latch

When the Clock = 0, both AND gates will output 0s. This guarantees that the original SR-latch has inputs of R'= 0 and S'= 0, so it cannot change its memory state.

When the Clock = 1, the AND gates allow the R and S inputs to be passed through without change. This does not exactly solve either of the previously mentioned problems, but we can at least prevent the *clocked-SR latch* from changing by having the Clock = 0.

To solve the problem of inputs of R = 1 and S = 1 leading to an undefined state. We'll consider a slightly different clocked latch called a *clocked D-latch*. The clocked D-latch remembers the value on the D input when the Clock = 1. If the Clock = 0, then the D input is ignored and the one-bit memory is not changed.

The clocked D-latch circuit is very similar to the clocked SR latch.

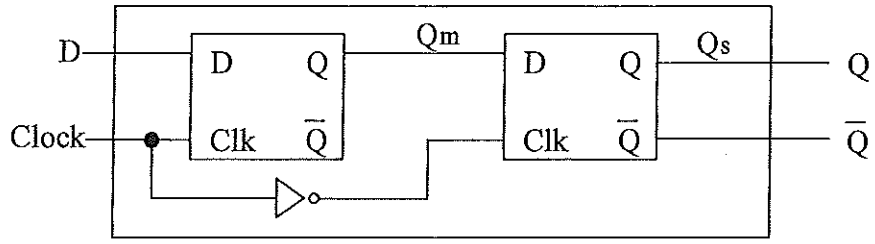


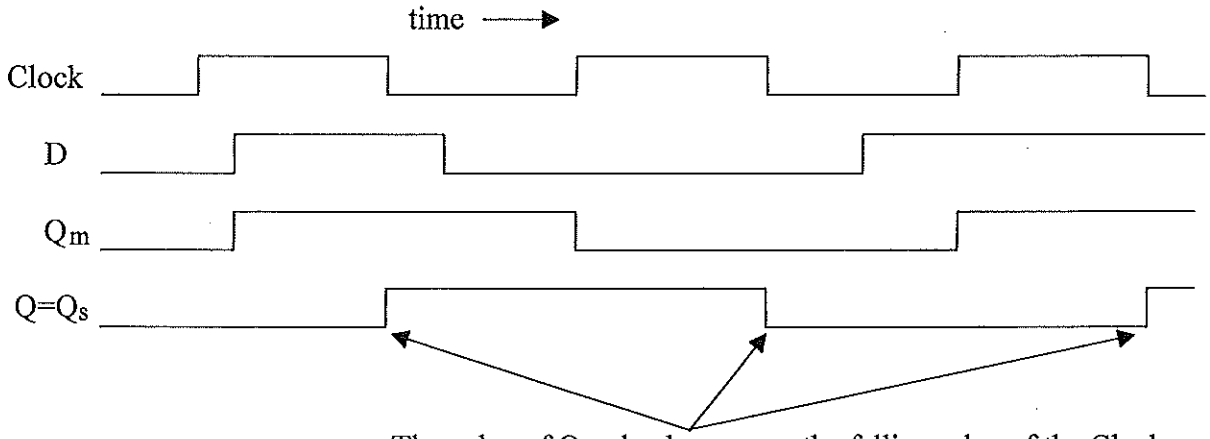
Original SR-latch

To fix the problem of the one-bit memory having its output changing being determined by some number of gate delays after its inputs are changed. We can use two clocked D-latches to build a *Master-Slave D flip-flop* which changes state only on the falling edge of the clock pulse. The below timing diagram shows how the internal clocked D-latches change within the master-slave flip-flop to achieve this.

When the Clock = 1, the master D-latch can change to D's value, but the slave D-latch cannot change since it receives the negated Clock signal of 0. When the Clock = 0, the master D-latch cannot change, but the slave D-latch can change to remember the value being remembered in the master D-latch (Q_m).

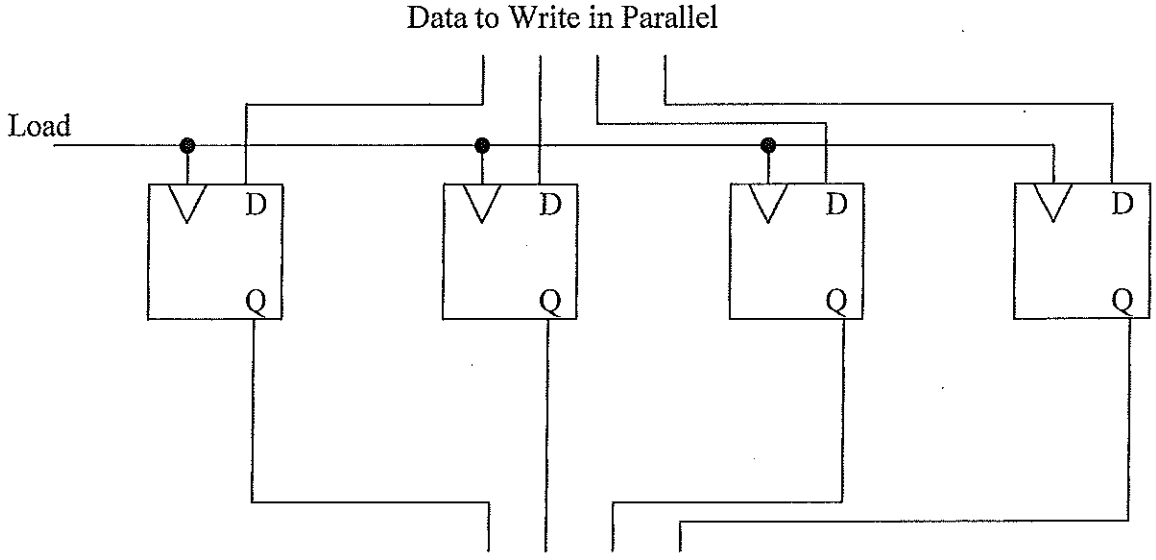
"Memory" Supplement for Section 3.6 of the textbook





The value of Q only changes on the falling-edge of the Clock

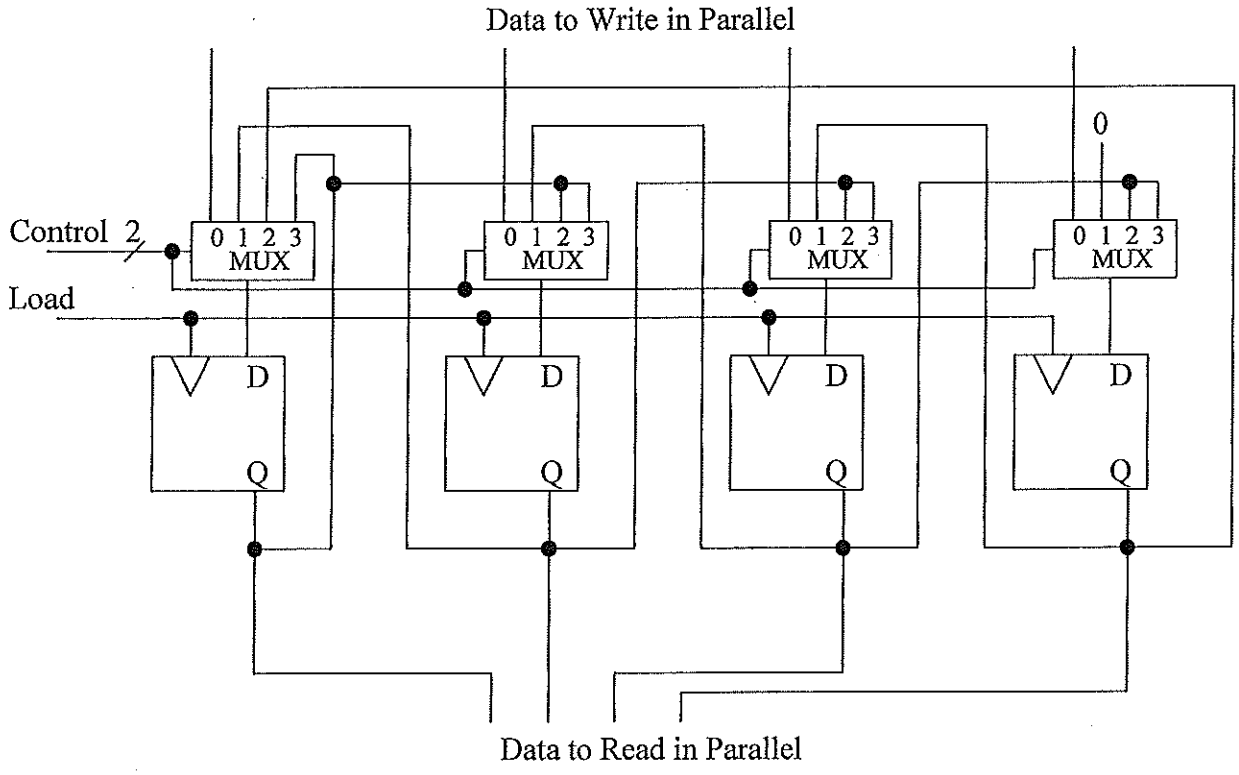
This master-slave D flip-flop allows for a collection of one-bit memories to work in a lock-step fashion to hold a multiple bit value. The value currently contained in a D flip-flop can to read (from the slave's Q) in the same clock pulse as a new value is written to it (into the master's D input). The following block diagram shows how four D flip-flops can be used to load and read 4-bit value in parallel. The "triangle" input of each flip-flop connected to the Load wire is the "Clock" input in the above diagram. This triangle input indicates that a flip-flop is being used instead of a latch.



Another more complex example would be a *shift-register* that allows the value stored in the register to also be shifted by one bit position several ways. The below diagram is a 4-bit shift-register that is able to perform the following operations:

- parallel read/output of all bits (just look at the Q values)
- (Control of 00_2) parallel write/input of all bits
- (Control of 01_2) logical shift left one bit position (value shifted out of most-significant bit is lost and a "0" is shifted into the least-significant bit)
- (Control of 10_2) circular shift right one bit position (value shifted out of least-significant bit is shifted into the most-significant bit)
- (Control of 11_2) arithmetic shift right (sign-extend the most-significant bit)

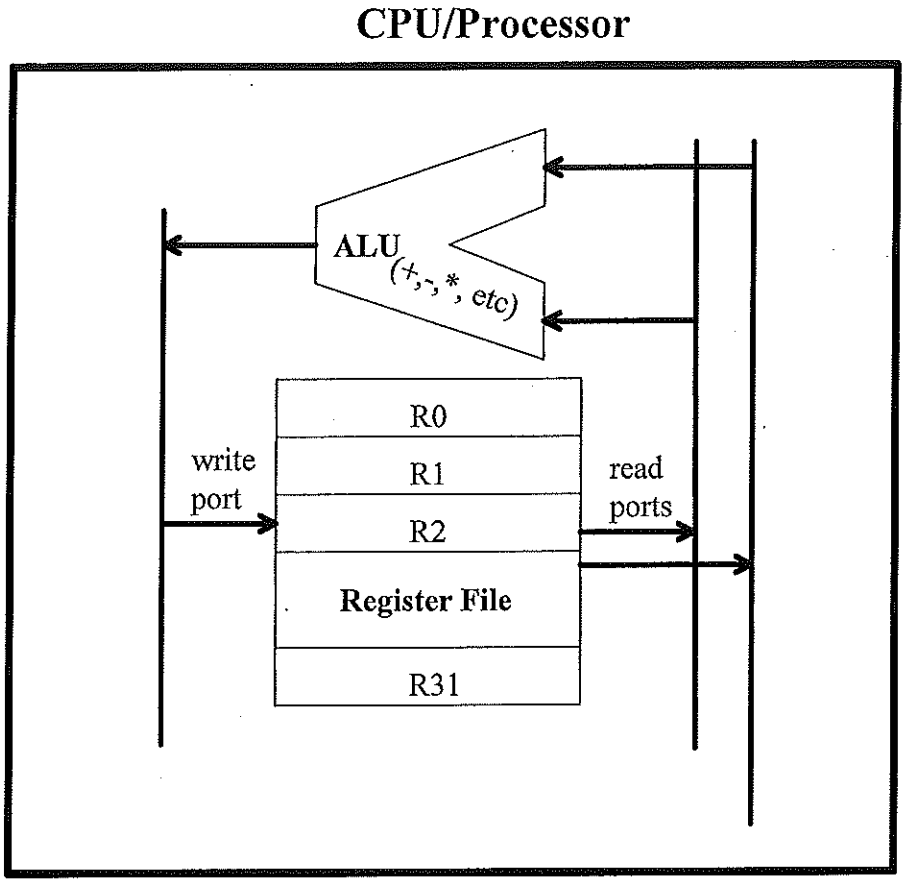
For each D-flip flop, the output of a MUX is used as the D-input. The Control wires control the MUXs to vary the value routed to the D inputs.



Register File:

Computers typically have a collection of registers, called a *register file*, to store data values currently being calculated with by the CPU. Individual registers are numbered (register 0 or R0, register 1 or R1, etc.) and assigned by the compiler or the assembly-language programmer to hold variable values.

Since most arithmetic operations need two operands (e.g., addition: Y + Z), most register files have at least two *read/output ports* and one *write/input port* to accommodate sending two values to the ALU and receiving one result.

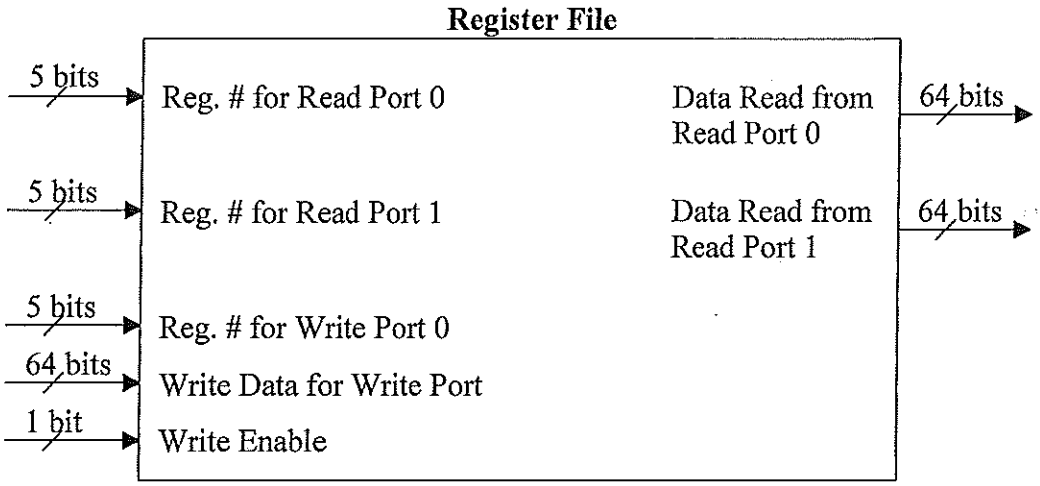


To control each read port we need to be able to specify a register number for the register to be read. The width/number of bits read equals the number of bits per register. For example, 32 register (R0 to R31) need 5-bits to represent this range of values 00000_2 to 11111_2, i.e., 0_{10} to 31_{10}.

To control a write port we need to be able to specify a register number for the register to be written, the data to be written (equal to the number of bits in a register), and a write-enable signal. The write-enable signal indicates if we are writing a register, i.e., indicates if we should ignore the write register # and data.

A register-file block-diagram assuming:

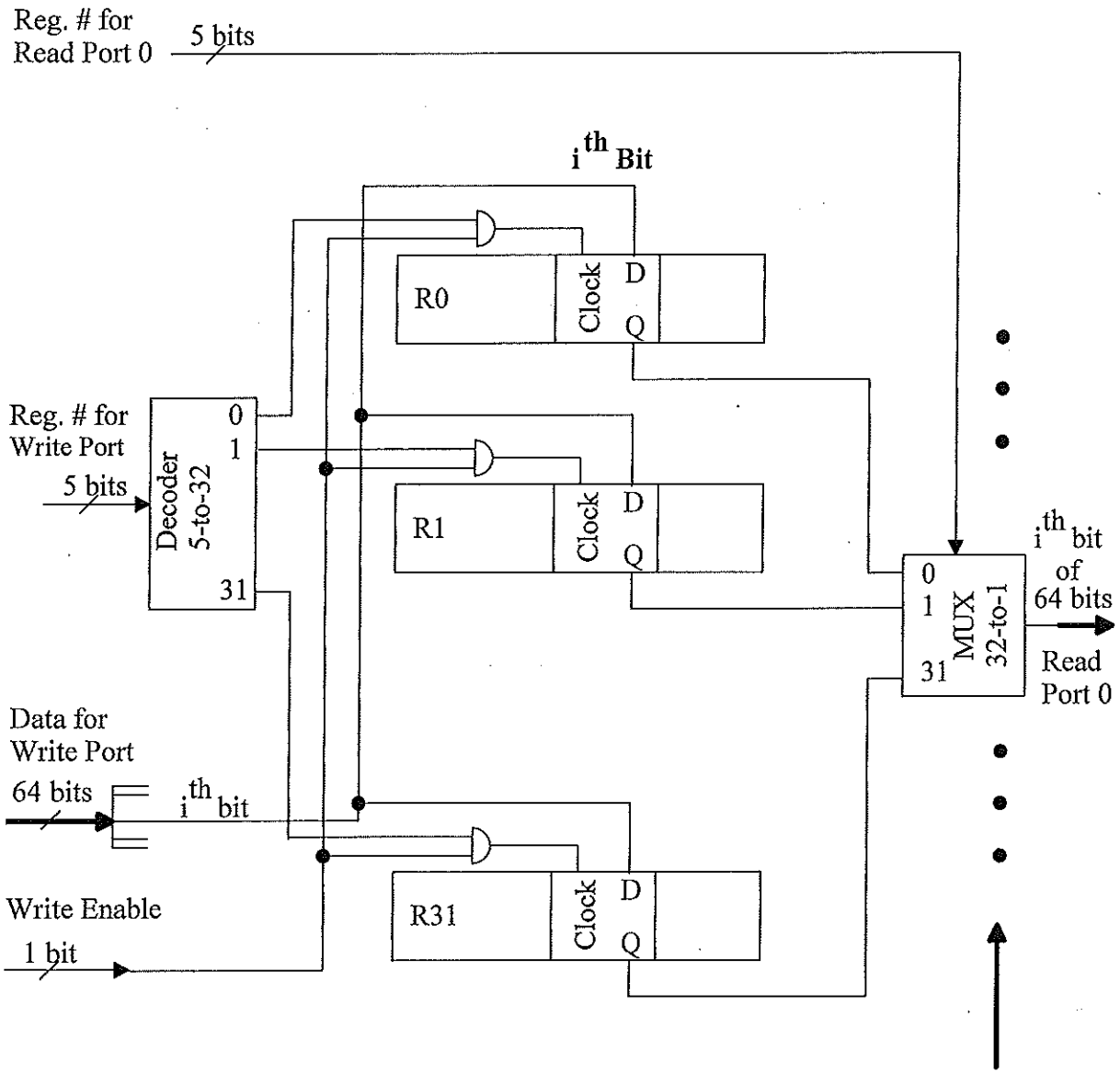
- 32 registers numbered R0 to R31 (00000_2 to 11111_2 in binary)
- 64-bit registers
- two read ports
- one write port



One-bit (i^{th} bit) Slice of a Register File Implementation Using D-latches:
The diagram below shows only a one-bit slice for the ith bit of the register file to simplify the diagram.

The write-port register number (5-bit number from R0 to R31) is input to a decoder to "wake-up" the corresponding register to be written by sending a 1 into the clock signal for register to be written. The decoder sends 0s to all the other registers' clock signals. Notice that the selected register only receives the 1 from the decoder if the write-enable signal is also a 1.

When a register is read at a read-port, the whole register is read in parallel (i.e., all 64-bits at the same time). For example, 64-bit registers requires 64 data wires of output. Each output wire is fed by a multiplexer which uses the register number to be read as its Control to select the bit from the appropriate register. In this example 64 MUXs each with 32-inputs (one for each register) are need per read port. Since we have two read ports, a total of 128 MUXs would be needed.



Note: To write a D flip-flop, we want its reg. # to be specified and the "Write Enable" to be asserted.

Note: 64 MUXs are need per Read Port - one per bit

"Memory" Supplement for Section 3.6 of the textbook

To help clarify the above discussion, a complete (not just a one-bit slice) register file that has:

- 4 registers (numbered R0 to R3, i.e., register numbers 00_2 to 11_2)
- 2-bits per register
- one write-port
- two read-ports

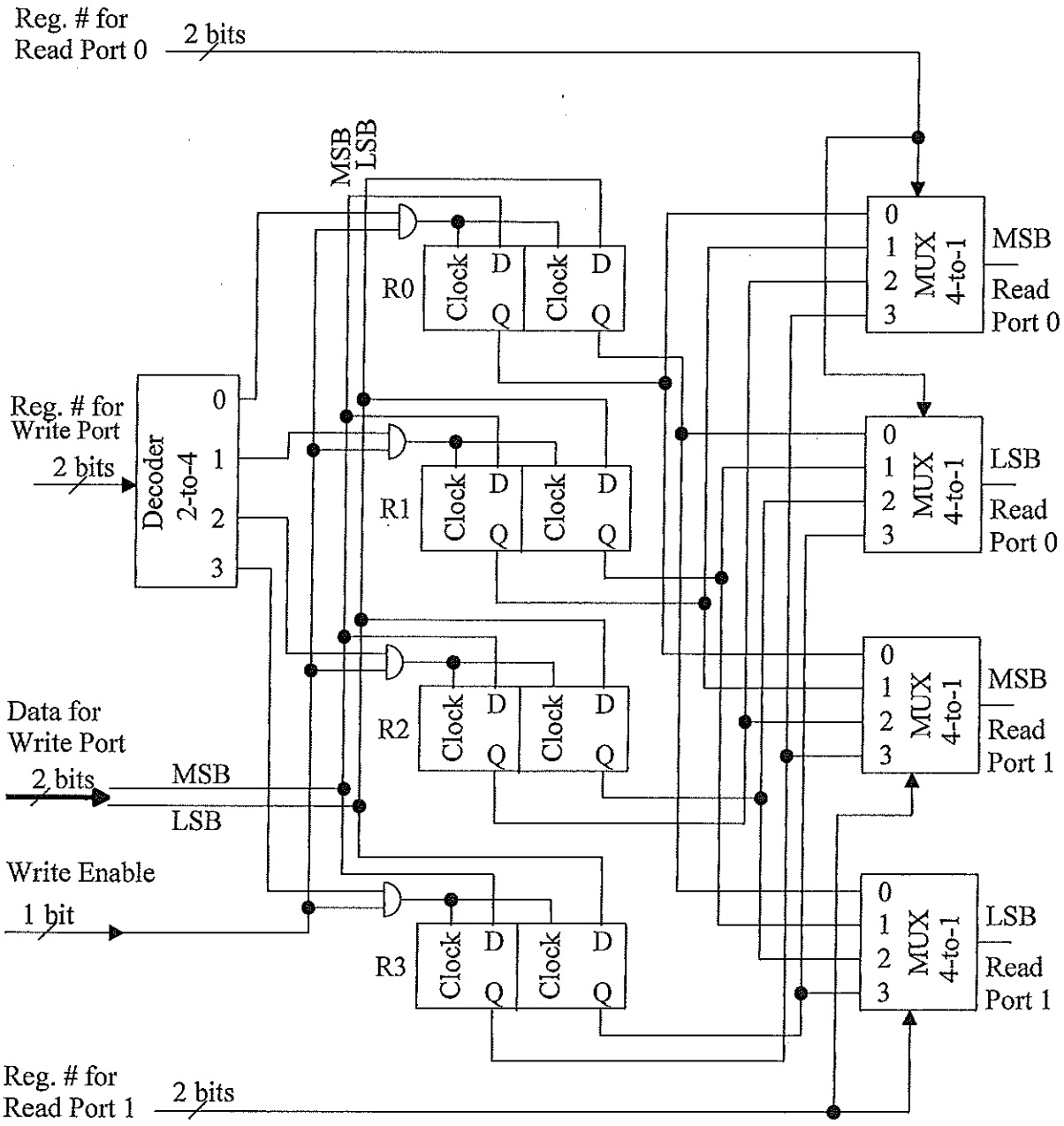


Figure 3.32 on page 143 is a similar to the above picture, except it is for a 4 word x 3 bit *memory*. Memories unlike register files typically have only a single combined read/write port. So only a read or a write can be performed at the same time. That's why Figure 3.32 has only one set of address (S_0 and S_1) lines that act like the register # in the above diagram. Notice that Figure 3.32 shows the AND gates and OR gates need to implement the decoder (along the bottom) and the MUXs (along the right and above the D flip-flops).

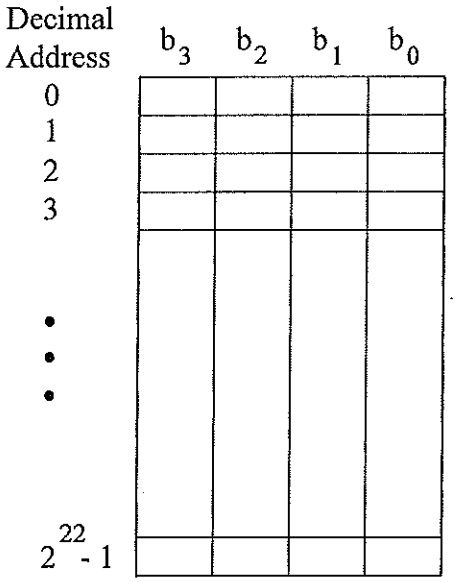
Implementation of Large Memory Chips:

The above register-file design does not scale well for large (realistic) memories for several reasons:
- the number of gates in the address decoder (and MUXs) grows exponentially with the number of bits in the address.
- lots of wires into/out of the memory chip for address, data, and control

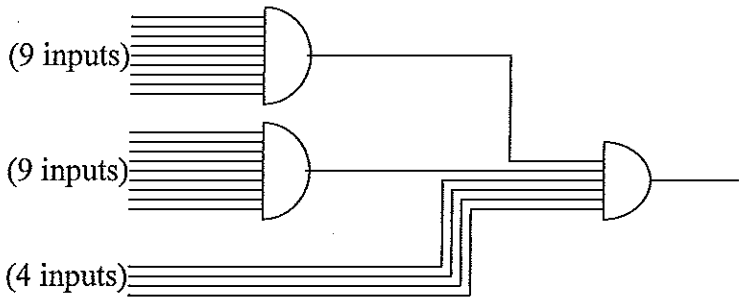
These problems are solved by
- using square-array of bits and decoding the address in two parts (row then column number)
- eliminate MUX's by using tri-state buffers
- single-port RAM memory - data wires shared for reading and writing, and one address specified

Consider a larger memory chip with 4M word x 4-bit words, i.e., a 4M x 4 memory chip. This chip would have a 22-bit addresses since $4M = 2^{22}$.

Conceptually (logically/externally) we view a 4M x 4-bit memory as pictured below with:
- each memory word made up of four bits: $b_3\ b_2\ b_1\ b_0$



When we want to write a word, we supply a 22-bit address and the corresponding new 4-bit word for that row. If we actually implemented this memory using a register-file design, we'd need a 22-to-2^{22} decoder for the write-port. If we think about how a decoder is implemented, each decoder output is fed by an AND gate looking for its corresponding address. Thus, we'd need 2^{22} (or 4 M) 22-input AND gates. Since 22-input AND gates exceed the limit of 9-inputs or less for an AND, we would need to build each 22-input AND gate using 3 smaller ANDs as:



A total of 3 x 2^{22} AND gates are needed to implement just the write-port decoder.

As in Figure 3.32, the read port could use the same decoder as the write port. Still, each bit in the memory requires a 2-input AND gate. Additionally, each of the four output wires is fed by a really BIG 2^{22}-input OR gate. Because of the 9-input limit, each 2^{22}-input OR must be implemented by multiple levels of smaller OR gates. The number of actual OR gates needed to implement each 2^{22}-input OR would be about:

$$\left\lceil \frac{2^{22}}{9} \right\rceil + \left\lceil \frac{2^{22}}{9^2} \right\rceil + \left\lceil \frac{2^{22}}{9^3} \right\rceil + \ldots + 1 = 524{,}289$$

The total for number of gates for the read port would be 4 x 2^{22} ANDs + 4 x 524,289 ORs.

Assuming D-latches to store each bit of the memory, the total number of gates needed for storing all 2^{22} x 4 bits of memory is 2^{22} x 4 x 4 = 2^{26} gates.

Overall, the number of gates in the whole memory chip is about (3 x 2^{22}) + (4 x 2^{22} + 4 x 524,289) + (2^{26}) .

Since the D-latch gates are the only ones that are used for actually storage, the gates associated with the read and write ports can be considered overhead. The percentage of gates used for actual storage (the D-latch gates) of bits is:

$$\frac{2^{26}}{(3\times 2^{22})+(4\times 2^{22}+4\times 524{,}289)+(2^{26})} \times 100 = 68.1\ \%$$

In practice memory chips do not use SR-latch based 1-bit memories, called *static memory*, because each bit requires multiple gates per bit of storage (e.g., 4 gates per bit for a D-latch). To increase the amount of memory on a chip, *dynamic memory* is used that only requires **one gate per bit** of storage. The one gate used in one bit of dynamic memory acts like a capacitor. You might recall from Physics that a capacitor is a device for storing a charge across two nearby "plates." The electrons from one plate are forces from one plate to the other plate. Because the plates are insulated from each other, the charge is stored. The magnitude of the charge difference between the plates determines whether a "0" is being stored (say a voltage difference of 0 to 1 volts) or a "1" is being stored (say a voltage difference of 2 to 5 volts) in the memory.

Dynamic memory (the "D" in DRAM) can store about four times more bits on a chip than static memory, but is also has some problems:

- reading the value being stored is relatively slow since the volt difference must be sensed to determine the binary value being stored.
- chips are made from semiconductor material, so a dynamic-memory gate storing a "1" with a high voltage difference (say 5 volts) will have electrons leak across the "plates" and eventually be storing a "0" (voltage difference of 0). To solve this problem, dynamic memory chips must be *refreshed* which means that periodically the whole memory must be read and rewritten to boost the "1" values up to a high voltage difference. Usually, the memory refreshing only takes a fraction of a percent of the memory's overall execution time.

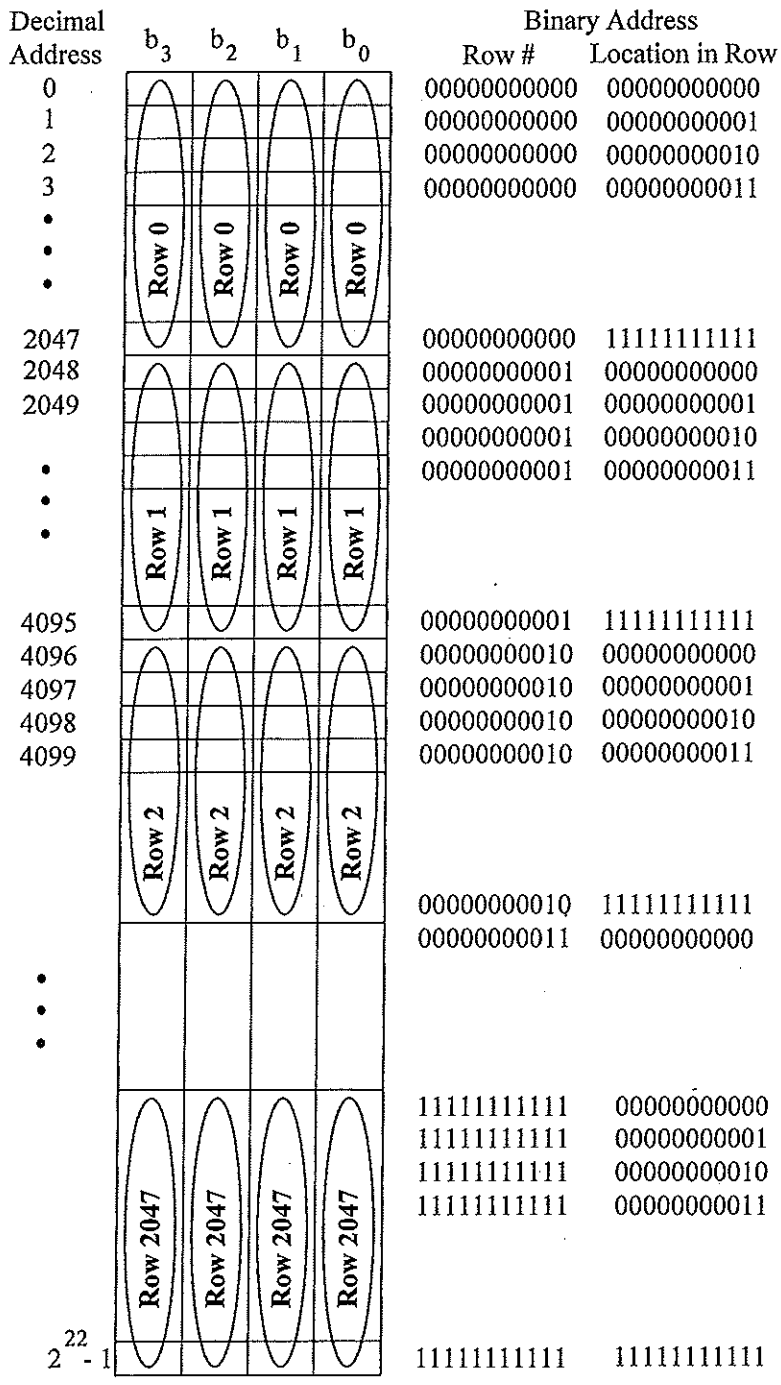
"Memory" Supplement for Section 3.6 of the textbook

If we recalculate the percentage of gates used for actual storage assuming dynamic memory (one gate per bit), we get:

$$\frac{2^{24}}{(3\times 2^{22})+(4\times 2^{22}+4\times 524{,}289)+(2^{24})} \times 100 = 34.8\ \%$$

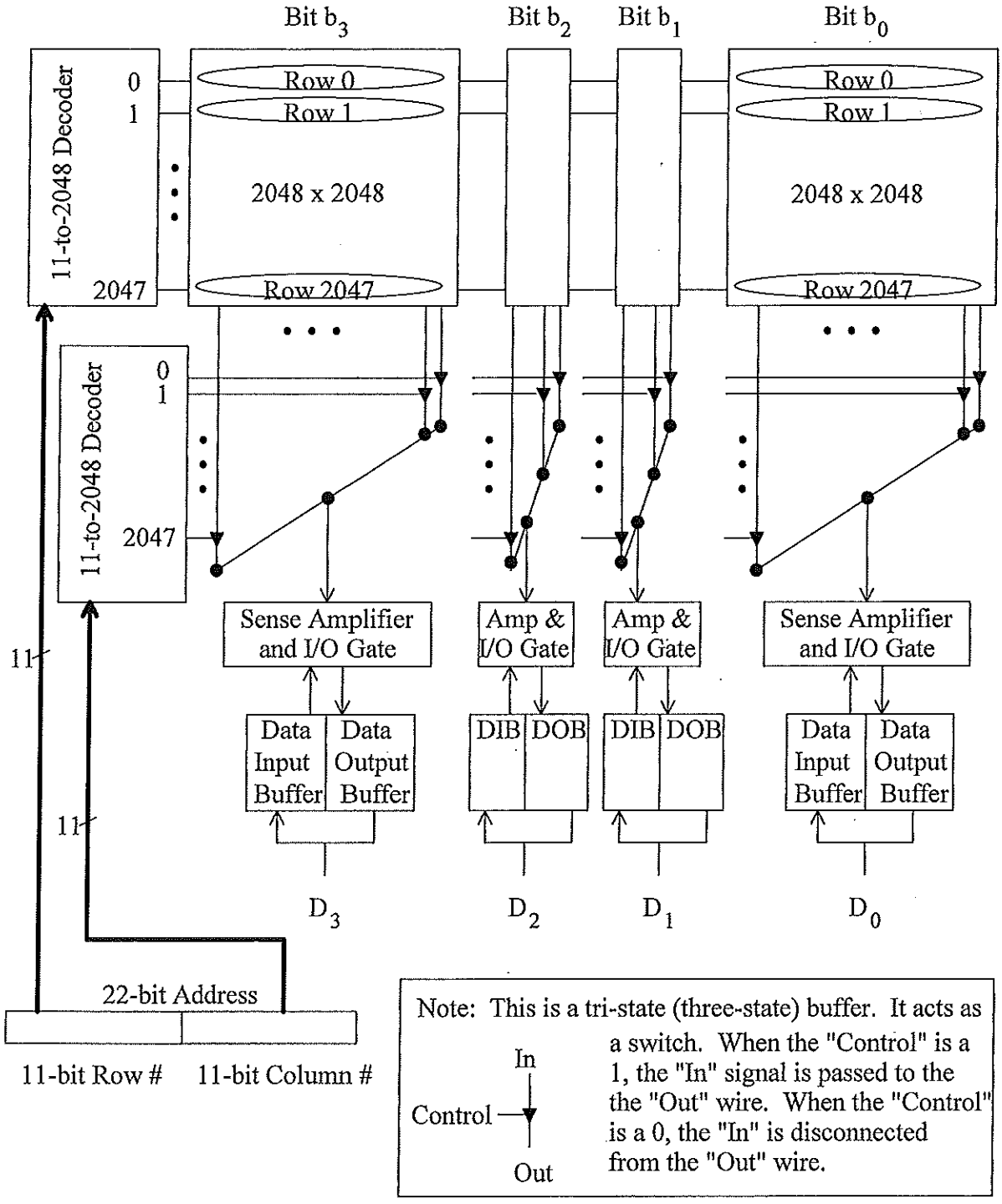
Thus, 65.2 % of the gates on the chip are being "wasted" in overhead for decoding, MUXs, etc.

To help us see how the 4M x 4-bit memory gets mapped to the 2048 x 2048 x 4 memory array on the next page consider splitting memory into 2048 word blocks as shown below. Notice that the most-significant 11 bits of the binary address correspond to the row number, and the least-significant 11 bits of the address to the offset within the row.



"Memory" Supplement for Section 3.6 of the textbook

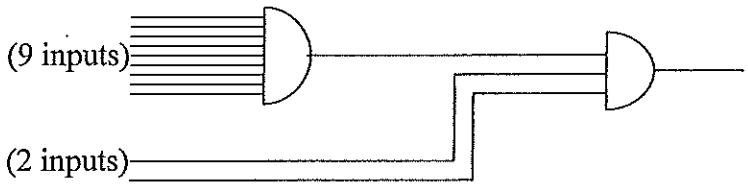
Each bit of a word is split into a separate 2048 x 2048 square memory "array". Each of these square memory arrays is 2048 x 2048 = 2^{11} x 2^{11} = 2^{22} = 4M bits.
The 22-bit address of the 4M memory is split into two 11-bit parts. The upper 11-bits is first used to activate the correct row within the square memory arrays. Of the 2048-bit row that is read from each memory array, we are interested in only one bit. The lower 11-bits of the address specifies the location of the desired bit within the 2048-bit row.



Note: This is a tri-state (three-state) buffer. It acts as a switch. When the "Control" is a 1, the "In" signal is passed to the the "Out" wire. When the "Control" is a 0, the "In" is disconnected from the "Out" wire.

"Memory" Supplement for Section 3.6 of the textbook

For this "square-memory" design of memory, we can recalculate the percentage of gates used for actual storage (the dynamic-memory "capacitor" gates) of bits. We now have two, smaller, 11-to-2^{11} decoders. Each decoder output is fed by an AND-gate looking for its corresponding address. Thus, we'd need 2^{11} (or 2 K) 11-input AND gates. Since 11-input AND gates exceed the limit of 9-inputs or less for an AND, we would need to build each 11-input AND gate using 2 smaller ANDs as:



A total of $2 \times 2 \times 2^{11} = 2^{13}$ AND gates are needed to implement both the row and column decoders.

No MUXs are needed in the square-memory design, but we do need $4 \times 2^{11} = 2^{13}$ tri-state buffers.

Assuming dynamic memory to store each bit of the memory, the total number of gates needed for storing all $2^{22} \times 4$ bits of memory is $2^{22} \times 4 = 2^{24}$ gates.

Overall, the number of gates in the whole memory chip is about $2^{13} + 2^{13} + 2^{24}$.

For the square-memory design, the percentage of gates used for actual storage of bits is:

$$\frac{2^{24}}{2^{13}+2^{13}+2^{24}} \times 100 = 99.9\ \%$$

Thus, only about 0.1 % of the memory chips capacity is wasted on overhead for decoding, MUX, etc.

The square-memory design also offers a couple of other important benefits:

- when refreshing the memory, whole rows can be read and rewritten at a time so refreshing is relatively quick
- since whole rows are read from the slow dynamic memory at a time, a block of memory from contiguous memory locations can be read about as fast as a reading a single memory access. Thus, many real DRAM memory chips provide a *fast-page mode* or *burst* feature that allow accessing a block of memory efficiently.

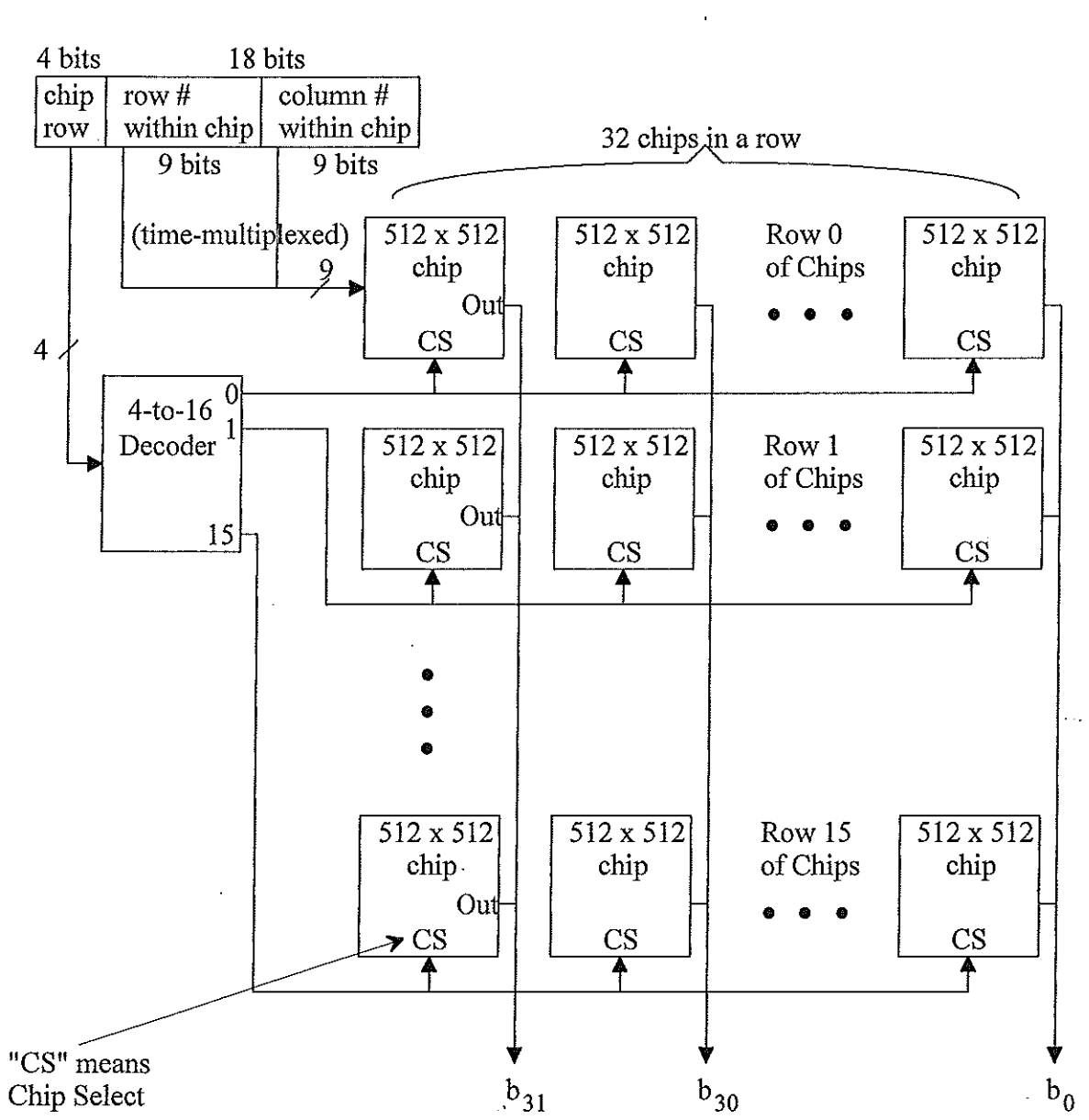
Implementing Large Memory with Smaller Chips

Consider for example, implementing 4M x 32 bits memory with 256KB x 1 bit chips.
The 256KB x 1 chips are implemented as square arrays of 512 x 512.

We will use an two-dimensional array of the 256KB x 1 chips to implement the larger memory.

The number of chips per row would be 32 bits / 1 bit = 32 chips.
The number of chips per column would be 4M / 256K = $2^{22} / 2^{18} = 2^4 = 16$ chips per column.

The 22-bit address of the 4M x 32 bit memory would be split up as follows:



1) Summarize in your own words why a "square-memory" implementation of main memory is used instead of a "register-file" implementation.

2) Why do RAM memory chips use dynamic memory (capacitor based) instead of static memory (SR-latch based) eventhough dynamic memory has the drawback that it needs to be refreshed and is slower?

3) To illustrate how poorly a register-file design scales, consider implementing a 1 G x 8 (1 G registers, each with 8 bits) register file with one write-port and **one** read-port.

a) How many and what type of decoder(s) would be needed?

b) How many total gates (assume 9-input limit on AND & OR gates) would be needed to implement this (these) decoder(s)?

c) How many and what type of MUX(s) would be needed?

d) How many total gates (assume 9-input limit on AND & OR gates) would be needed to implement this (these) MUX(s)?

e) Assuming D flip-flops to store each bit (4 gates/flip-flop). What % of the total gates is used to implement the D flip-flops? The formula for this would be:

$$\frac{\text{Total number of gates for the D flip-flops}}{\text{Total number of gates for D flip-flops + Total for the MUX(s) + Total for the Decoder(s)}} \times 100$$

4) Redo the previous question using the 1 G x 8 square-memory implementation similar to the Memory supplement. **Assume a single shared read/write port and assume dynamic memory is used to store each bit (1 gate/bit) for part (e).**

a) How many and what type of decoder(s) would be needed?

b) How many total gates (assume 9-input limit on AND & OR gates) would be needed to implement this (these) decoder(s)?

c) How many and what type of MUX(s) would be needed?

d) How many total gates (assume 9-input limit on AND & OR gates) would be needed to implement this (these) MUX(s)?

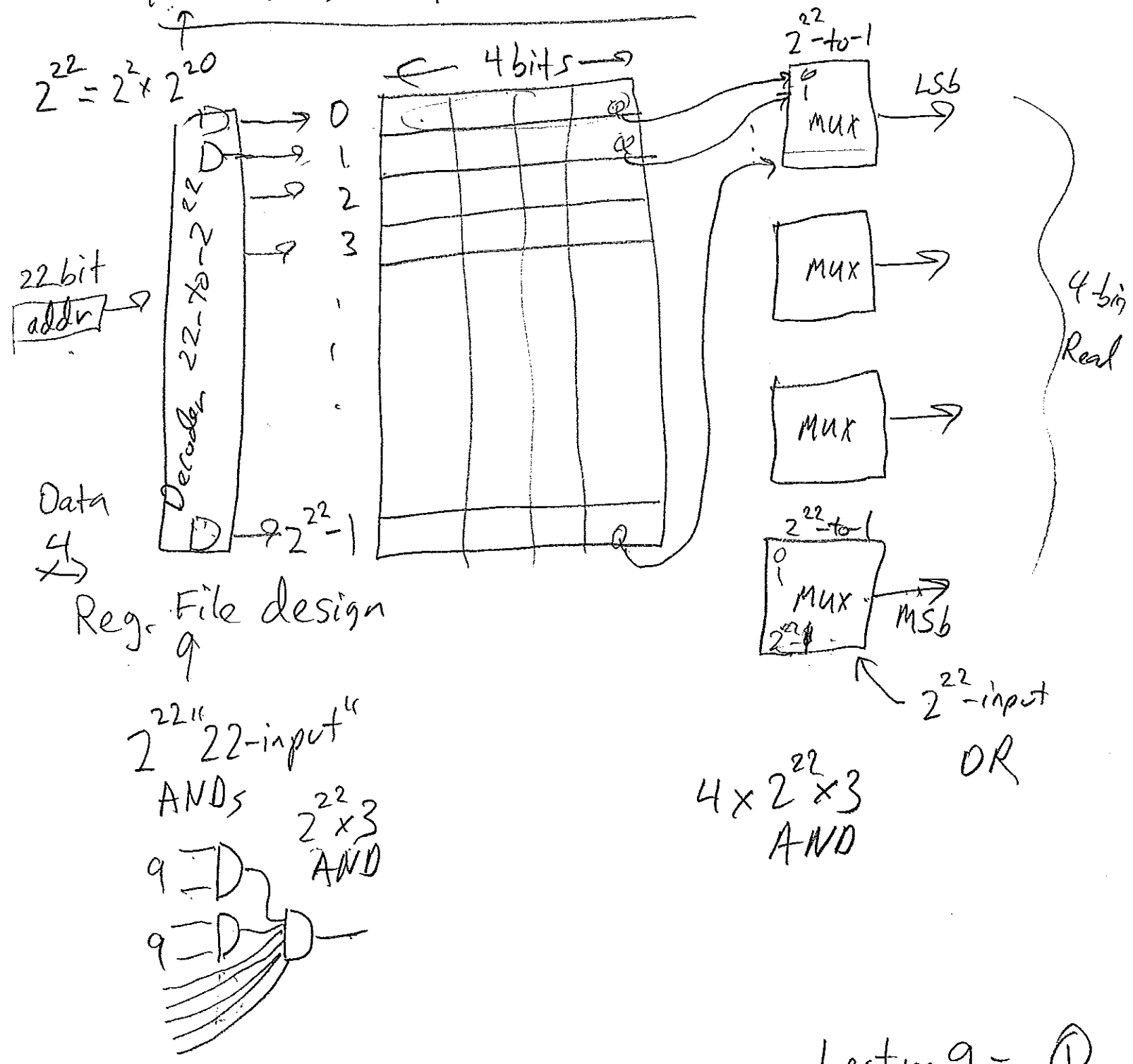
e) Assuming dynamic memory is used to store each bit (1 gates/flip-flop). What % of the total gates is used to implement the dynamic memory? The formula for this would be:

$$\frac{\text{Total number of gates for the dynamic memory}}{\text{Total number of gates for dynamic memory + Total for the MUX(s) + Total for the Decoder(s)}} \times 100$$

Lecture 9: Reg. File v.s. square-memory RAM

Reg. File design <u>does not scale well</u> to RAM (main) memory

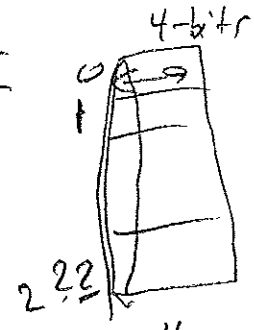
4M words × 4 bit RAM

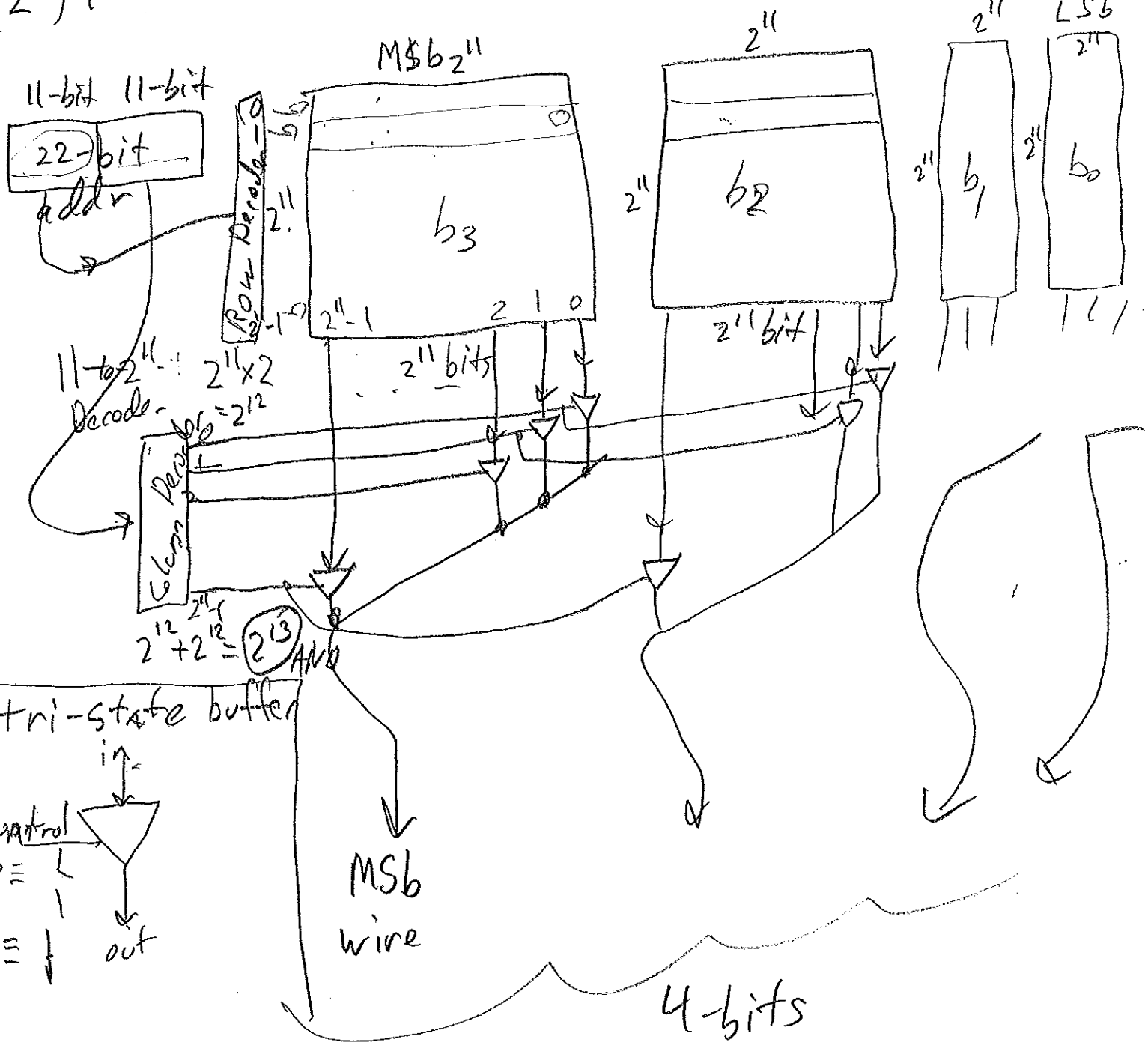


"Square-Memory" RAM - single port

only read or write single value

(2^{22}) 4M * 4bits



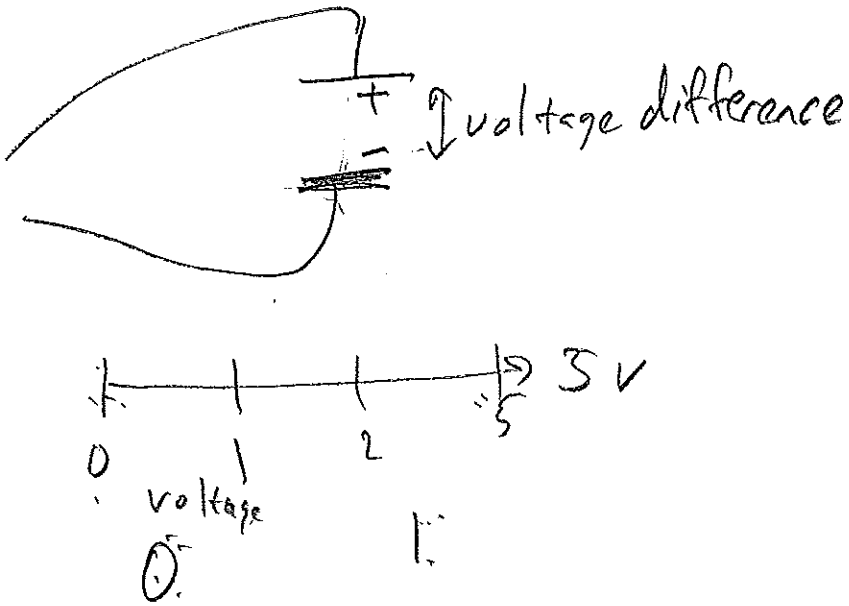


D-latch 4 gates/bit Static Memory

Square Memory RAM

- 1 gate/bit Dynamic Memory

capacitor based



Adv.

\+ 4 times more storage

Disadv.

- periodically read and rewrite whole memory "refresh" memory ⇒ << 1% time
- slower R/w sense whether 1 or 0 stored

Test 1 will be Thursday Sept. 28, in class. It will be closed book and notes, except for one 8.5" x 11" sheet of paper (front and back) with notes. Review topics for Test 1 are:

Chapter 1.
Basic structure and functional units of a computer
Functional units: Input, CPU, Memory, Output, System bus
CPU components: control unit, ALU, regs., internal CPU interconnection
History: Computer Generations: 1st (vacuum tubes), 2nd (transistors, early operating systems, and high-level programming languages), 3rd (small and medium scale integrated circuits), 4th (large and very large scale integrated circuits and microprocessors)
Moore's law, von Neumann model, Instruction/Machine cycle
Programming Languages: High-level language, Assembly Language, and Machine Language
Computer Level Hierarchy

Chapter 2.
Unsigned binary numbers
Conversion between base 10, base 2, and base 16
Signed number representation: sign bit and magnitude, one's complement, two's complement
Addition and subtraction of signed and unsigned numbers
Overflow in integer arithmetic
Booth's Algorithm for integer multiplication
Floating-point Representation: 32-bit, 64-bit, and 128-bit IEEE 754 standard and special values
Addition and multiplication of floating-point numbers
Character Representations: Binary-Coded Decimal (BCD), EBCDIC, ASCII, Unicode
Error Detection and Correction: Hamming Code and CRC

Chapter 3. (SKIP SUBSECTION 3.6.6)
Truth tables for the gates
Boolean algebra notations: sum-of-products
Combinational circuit design (no memory): 1) determine truth table for function, 2) using Boolean identities or K-maps to get minimized sum-of-products function, 3) draw implementation of minimized function (using gates)
Common combinational circuits: decoder, multiplexer (MUX), 1-bit Adders (half and full), ripple-carry adder, faster (2-bit) carry-lookahead adders,
Number of gate delays for a circuit.
Complexity number of a circuit (# of gates + # inputs to those gates)
Sequential Circuits (1-bit memory): SR latch - know how it remembers (two stable states, etc.), know how it changes states;
Gated/Clocked D latches. Master-slave D Flip Flop; their characteristic tables
Timing diagrams for latches and Flip Flops
Register: shifting & rotating
Register file - design and usage
Square-memory implementation of large memories

Fall 2016 Name:_____

Computer Organization Test 1

Question 1. (15 points)

a) Convert 450_{10} to a binary (base 2) value.

b) Convert 450_{10} to a hexadecimal (base 16) value.

c) Convert -450_{10} to a two's complement value.

d) Perform the following arithmetic operations:

1100110_2	1100110_2	$9\ 7\ 6\ C\ 3_{16}$	$9\ 7\ 6\ C\ 3_{16}$
$+\ 101111_2$	$-\ 101111_2$	$+\ 4\ 9\ E\ A\ 8_{16}$	$-\ 4\ 9\ E\ A\ 8_{16}$

Question 2. (8 points)

a) Convert the value $+108.5625_{10}$ to its binary representation.

64 32 16 8 4 2 1 . .5 .25 .125 .0625

[ ] . [ ]

b) Normalize the above value so that the most significant 1 is immediately to the left of the radix point. Include the corresponding exponent value to indicate the motion of the radix point.

$1.$[ ] $\times 2^{[\ ]}$

c) Write the corresponding normalized 32-bit IEEE 754 floating point representation for $+108.5625_{10}$.

Sign bit	8-bit (bias 127) Exponent	23-bit Mantissa (for normalized values, leading 1 not stored)

Question 3. (7 points) Consider adding the values $1.101_2 \times 2^{50}$ and $1.01_2 \times 2^{35}$.

a) How many places does the second number's mantissa get shifted before the addition?

b) After we add these two numbers and store the results back into a 32-bit IEEE 754 variable, what would be the result stored as a 32-bit IEEE 754 variable?

Sign bit	8-bit (bias 127) Exponent	23-bit Mantissa (for normalized values, leading 1 not stored)

Question 4. (8 points) Determine the Hamming codeword if the 8-bits of data (D_7 to D_0) are 0110 1100_2, i.e., fill in the data bits and determine the values of the four even-parity bits (P_8, P_4, P_2, and P_1) to allow for one-bit error detection and correction.

12	11	10	9	8	7	6	5	4	3	2	1
D_7	D_6	D_5	D_4	P_8	D_3	D_2	D_1	P_4	D_0	P_2	P_1
4+8	1+2+8	2+8	1+8	8	1+2+4	2+4	1+4	4	1+2	2	1

Question 5. (12 points) Consider the 12-bit data 101110011001_2 and generator polynomial $G = x^5 + x^2 + 1$ (100101_2). Using the Cyclic Redundancy Check (CRC) method:

a) what code word (data and remainder) would be sent to the receiver?

```
1 0 0 1 0 1 | 1 0 1 1 1 0 0 1 1 0 0 1 0 0 0 0 0
```

b) Upon receiving a codeword, describe how the receiving computer checks for an error in transmission?

Question 6. (20 points) For the Boolean function represented an unsimplified sum-of-products (SOP) Boolean expression $F = A\bar{B}\bar{D} + A\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D} + ABCD + AB\bar{C}D + A\bar{B}C\bar{D}$

a) Using a K-map or the identities of Boolean algebra simplify this function F as much as possible.

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0+x = x$
Null (or Dominance) Law	$0x = 0$	$1+x = 1$
Idempotent Law	$xx = x$	$x+x = x$
Inverse Law	$x\bar{x} = 0$	$x+\bar{x} = 1$
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy + xz$
Absorption Law	$x(x+y) = x$	$x+xy = x$
DeMorgan's Law	$(\overline{xy}) = \bar{x}+\bar{y}$	$(\overline{x+y}) = \bar{x}\bar{y}$
Double Complement Law	$\bar{\bar{x}} = x$	

b) Draw the ***simplified circuit*** for your answer in part (a) using AND, OR, and NOT gates. (NOTE: If you were unable to simplify the circuit, you may alternatively draw the unsimplified circuit instead.)

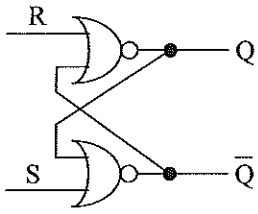
c) Ignoring NOT gates, how many gate delays are in your above circuit?

d) What is the complexity (# of non-NOT gates + # of inputs to those gates) of your circuit?

Question 7. (5 points) Why do RAM memory chips use dynamic memory (capacitor based) instead of static memory (SR-latch based) eventhough dynamic memory has the drawbacks of needing to be refreshed and being slower?

Question 8. (10 points)

a) If R = 1 and S = 0 for a while (> 2 gate delays), then, then what will be the output on Q and $\overline{Q}$?



A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

b) Now, if R goes to a 0 value while keeping S = 0, explain how the SR-latch continues to remember.

Question 9. Recall the sum-of-products (SOP) Boolean formula for the carry-out (c_{i+1}) of a n-bit adder were :

1-bit adder: $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$.

2-bit adder: $c_{i+1} = x_i y_i + x_i (x_{i-1} y_{i-1} + x_{i-1} c_{i-1} + y_{i-1} c_{i-1}) + y_i (x_{i-1} y_{i-1} + x_{i-1} c_{i-1} + y_{i-1} c_{i-1})$
$= x_i y_i + x_i x_{i-1} y_{i-1} + x_i x_{i-1} c_{i-1} + x_i y_{i-1} c_{i-1} + y_i x_{i-1} y_{i-1} + y_i x_{i-1} c_{i-1} + y_i y_{i-1} c_{i-1}$

3-bit adder: $c_{i+1} = x_i y_i + x_i$ (7 product terms of 2-bit adder) $+ y_i$ (7 product terms of 2-bit adder)
$= x_i y_i + x_i x_{i-1} y_{i-1} +$ ... (12 product terms omitted) $+ y_i y_{i-1} y_{i-2} c_{i-2}$

The table summarizing the gate delays for different types of adders **assuming an 9-input limit into any gate.**

Type of Adder	# of product terms in SOP expression to be OR'ed	Most # of inputs in any of the product terms	# gate delays due to product/AND terms	# gate delays due to sum/OR	Gate Delay per Adder	Gate Delays for 32-bit ripple adder using adders of this type
1-bit	3	2	1	1	2	32 x 2 = 64
2-bit	7	3	1	1	2	16 x 2 = 32
3-bit	15	4	1	2	3	10 x 3 + 2 = 32

a) (5 points) Explain why each 3-bit adder required 3 gate delays while each 2-bit adder only requires 2 delays.

Question 10. (10 points) Suppose we have a register file with the following specifications:

- 8 registers (numbered 0 to 7) each with 32 bits
- one write port
- two read ports

a) How many "wires" would be needed to transmit data read from a read port?

b) How many "wires" would be needed to specify the register number of a read port?

c) How many decoders would be needed in the implementation of the whole register file?

d) What type of decoder(s) (i.e., # of inputs-to-# of outputs) are needed in part (c)?

e) How many multiplexers (MUXs) would be needed in the implementation of the whole register file?

f) What type of multiplexer(s) (i.e., # of inputs-to-# of outputs) are needed in part (e)?