CS240: Computer Organization and Assembly Language

Darshan Patel

Spring 2017

Contents

1 Computer Organization and Design

1.1 Introduction

Definition 1.1. Computer Architecture: concerned with the structure and behavior of the computer as seen by the user; includes the information formats, the instruction set and techniques for addressing memory

Definition 1.2. Computer Organization: concerned with the way the hardware components operate and the way they are connected together to form the computer system

Internal Parts of a Computer:

- Case
- CD-ROM/DVD-ROM/CDRW/DVD+RW
- CPU or processor

- Case fan
- CPU fan
- Hard drive
- Keyboard and mouse
- Memory
- Modem
- Monitor
- Power supply
- Motherboard
- Network card NIC

Definition 1.3. John von Neumann Architecture: memory, control unit and arithmetic logic unit are all connected to each other; in the arithmetic logic unit is the accumulator which gets inputs and gives output

Definition 1.4. Generic Computer Architecture: Input and output are connected to memory and memory passes information to the processor (control and datapath) which is then received back to the memory

Milestones in Computer Architecture

- Zeroth Generation: Mechanical computers
  - Blaise Pascal: addition and subtraction calculator
  - Charles Babbage
    - Difference Engine: addition and subtraction calculator
    - Analytic Engine: general purpose algorithms, 4 components: store (memory), mill (computation), input section (punched-card reader) and output section (punched and printed output)
  - Early 20th century using mechanical relays
- First Generation: Vacuum Tubes (1945 - 1955)
  - John von Neumann: mathematician and physicist
  - Von Neumann developed the basic computer architecture used to this day in most digital computers
- Second Generation: Transistors (1955 - 1965)
  - Transistors: a semiconductor device used to amplify or switch electronic signals or electrical power

  - Much smaller and reliable than vacuum tubes
  - Digital Equipment Corporation built the PDP-8 using a single bus (omnibus, a set of parallel wires) to connect the components of a computer; has individual buses for data, address and control and they all connect to CPU, memory and I/O

- Third Generation: Integrated Circuits (1965 - 1980)
  - Transistors are used to build integrated circuit chips
  - CPUs are one type of integrated circuits
  - Integrated circuits allowed computers to shrink in size and increase in computing power
  - The IBM 360 (based on integrated circuits) introduced multiprogramming, having several programs in memory at once

- Fourth Generation: Very Large Scale Integration VLSI (1980 - ?)
  - VLSI made it possible to put tens of thousands, then hundreds of thousand, and finally millions of transistors onto a single chip
  - VLSI continues to shrink the size and increase the power of computing
  - Lead to the Personal Computer era
  - Graphical User interface (GUI)
  - Field Programmable Gate Array (FPGA): a computer chip with a large collection of generic logic gates that is programmed to a circuit to perform a needed task

- Fifth Generation: Low-Power and Invisible Computers
  - Personal Digital Assistants that evolved into today's smartphones
  - Ubiquitous/Pervasive Computing: computers so small they can be imbedded into everything

Typical Multilevel Machine

- Level 0: Digital Logic Level
  - Logic Gate: made from transistors with one or more inputs performing some simple logical operation such as AND, OR, or NOT
  - 1 Bit Memory: implemented by several logic gates
  - Register: combining groups (groups of 16, 32, or 64) of 1-bit memories
  - Main Computing Engine: implemented by logic gates

- Level 1: Microarchitecture Level
  - 8 to 32 registers

  - Arithmetic Logic Unit (ALU)
  - Data Path: connects the registers to the ALU
  - Microprogram: controls the operation of the data path; either implemented via software of hardware; if implemented via software, then the microprogram is an interpreter for the instructions at level 2

- Level 2: Instruction Set Architecture Level
  - The machine's set of instructions to perform operations such as addition, Branch, etc.
  - Each instruction is interpreted by the microprogram or hardware execution circuits

- Level 3: Operating System Machine Level
  - Hybrid level providing ISA level instructions; these instructions are interpreted directly by the microprogram
  - Some instructions are identical to ISA level instructions; these instructions are interpreted directly by the microprogram
  - The remaining instructions are interpreted by the operating system

- Level 4: Assembly Language Level
  - Programs are translated rather than interpreted
  - The languages at level 1 to 3 are difficult to code for humans
  - Assembly language is a symbolic form of the numeric languages
  - An assembler performs translations of an assembly language program to a level 1 to 3 language and then is interpreted by the appropriate interpreter for that level's language

- Level 5: Problem-Oriented Language Level
- Programs are translated rather than interpreted
- The languages at level 5 are called high level languages and are translated by a compiler to a level 3 to 4 language

Language Levels:

- High level languages
- Assembly languages
- Binary language

Components of a CPU

- Control Unit: fetching instructions from main memory and determining their type
- Arithmetic Logic Unit: performing operations such as addition, Boolean AND, etc. needed to carry out instructions
- High speed memory: small set of registers
  - All the registers have the same number of bits
  - Hold a single number
  - Program Counter (PC): special register containing the address of the next instructions to execute
  - Instruction Register (IR): holds the current instruction being executed

CPU Organization

- Data Path: the register, ALU, and the buses connecting them together
- Instructions fall into one of two types:
  - register to memory
  - register to register
- Data Path Cycle: the process of running 2 operands through the ALU and sending the result back into a register

Instruction Execution - Fetch - Decode - Execute Cycle:

1. Fetch the next instruction from memory into the instruction register
2. Change the program counter to the following instruction
3. Determine the type of instruction just fetched
4. If the instruction uses a word in memory, determine its location
5. Fetch the word, if needed, into a CPU register
6. Execute the instruction
7. Go back to step 1 to begin executing the following instruction

Definition 1.5. Moore's Law: an observation by Gordon Moore that processing power for computing CPU doubles every 18 months at the same cost

1.2 Number Systems

Definition 1.6. Fixed Point Numbers: Fixed $<$ _____, _____ $>$ where the first number is number of digits and the second number is how many numbers is to the right of the point

Parts of a Number

- Integer part
- Fractional part

Digits

- Least Significant Digit: the rightmost digit in a number
- Most Significant Digit: the leftmost digit in a number

Number Systems

- Binary Number System: 2 binary digits - 0, 1
- Octal Number System: 8 digits - 0 through 7
- Decimal Number System: 10 digits - 0 through 9
- Hexadecimal Number System: 16 digits - 0 through 9, A through F

Definition 1.7. Unsigned Number: non-negative integer

Expanded form of $(429)_10$

$$4 \cdot 10^2 + 2 \cdot 10^1 + 9 \cdot 10^0$$

Example 1.1. What's $(429)_{16}$ in base 10?

$$4 \cdot 16^2 + 2 \cdot 10^1 + 9 \cdot 10^0 = 1065$$

Example 1.2. What's $(1011)_2$ in base 10?

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$$

Conversion:

- Algorithm to convert a non-negative integer from binary, octal or hexadecimal number system to decimal system
  1. Initialize result = 0
  2. For each digit from most to least significant, compute result = result $\cdot$ base + digit

- Algorithm to convert a non-negative integer from decimal to binary, octal or hexadecimal number system
  1. Initialize quotient = 0
  2. Compute quotient = quotient / base and remainder = quotient mod base, until quotient is zero
  3. The remainders in reverse order gives the number in the base number system

Example 1.3. Convert $(429)_{16}$ into base 10.

$$\begin{aligned} \text{result} &= 0 \\ 0 \cdot 16 + 4 &= 4 \\ 4 \cdot 16 + 2 &= 66 \\ 66 \cdot 16 + 9 &= 1065 \end{aligned}$$

$(1065)_{10}$

Example 1.4. Convert $(1065)_{16}$ to base 16.

$$\begin{aligned} \text{quotient} &= 1065 \\ \frac{1065}{16} &= 66 \text{ remainder } 9 \\ \frac{66}{16} &= 4 \text{ remainder } 2 \\ \frac{4}{16} &= 0 \text{ remainder } 4 \end{aligned}$$

The remainders were 9, 2, 4. Thus the number in base 16 is 429.

Octal to Binary

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Hexadecimal to Binary:

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Example 1.5. Convert 429 from base 16 to base 2.

In base 2: 4 is 0100, 2 is 0010, 9 is 1011. So 429 in base 16 is 010000101001 in base 2.

Note: For hexadecimals, group numbers into 4s. For octals, group numbers into 3s.

Example 1.6. Convert 010000101001 from base 2 to base 8.

010 is 2, 000 is 0, 101 is 5, 001 is 1. So the number is 2051 in base 8.

Note: If we want to go to octal to hexadecimal, or vice versa, use binary in between.

Signed Binary Numbers Implementations

- Signed Magnitude
- One's Complement
- Two's Complement
- Biased Notation

Note: The bias value in Biased notation is $2^{2-1} - 1$, where n is the number of bits used to store the number.

Unsigned Numbers in 4 bits

Unsigned Number	Binary Combination	Unsigned Number	Binary Combination
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Range of Unsigned Numbers: 0 - $2^n - 1$

Signed Magnitude Numbers in 4 bits

Signed Number	Binary Combination	Signed Number	Binary Combination
0	0000	-0	1000
1	0001	-1	1001
2	0010	-2	1010
3	0011	-3	1011
4	0100	-4	1100
5	0101	-5	1101
6	0110	-6	1110
7	0111	-7	1111

Range of Signed Magnitude: $-(2^{n-1} - 1) - (2^{n-1} - 1)$

One's Complement Numbers in 4 bits

Signed Number	Binary Combination	Signed Number	Binary Combination
0	0000	-7	1000
1	0001	-6	1001
2	0010	-5	1010
3	0011	-4	1011
4	0100	-3	1100
5	0101	-2	1101
6	0110	-1	1110
7	0111	-0	1111

Range of One's Complement: $-(2^{n-1} - 1) - (2^{n-1} - 1)$

Two's Complement Numbers in 4 bits

Signed Number	Binary Combination	Signed Number	Binary Combination
0	0000	-8	1000
1	0001	-7	1001
2	0010	-6	1010
3	0011	-5	1011
4	0100	-4	1100
5	0101	-3	1101
6	0110	-2	1110
7	0111	-1	1111

Range of Two's Complement: $-(2^{n-1}) - (2^{n-1} - 1)$

To switch from One's Complement to Two's Complement or vice versa, invert the number in binary and add 1.

Example 1.7. Convert +3 from one's complement to two's complement.

$$+3 \rightarrow 0011 \rightarrow 1100 \rightarrow 1100 + 1 \rightarrow 1101 \rightarrow -3$$

Example 1.8. Convert -3 from two's complement to one's complement.

$$-3 \rightarrow 1101 \rightarrow 0010 \rightarrow 0010 + 1 \rightarrow 0011 \rightarrow +3$$

Example 1.9. Convert 0 from one's complement to two's complement.

$$0 \rightarrow 0000 \rightarrow 1111 \rightarrow 1111 + 1 \rightarrow 10000$$

We have run out of bits. Not possible.

Bias Notation Number in 4 bits

Signed Number	Binary Combination	Signed Number	Binary Combination
-7	0000	1	1000
-6	0001	2	1001
-5	0010	3	1010
-4	0011	4	1011
-3	0100	5	1100
-2	0101	6	1101
-1	0110	7	1110
0	0111	8	1111

Range of Bias Notation: $-(2^{n-1} - 1) - 2^{n-1}$

Bias Value: $[(2^{n-1}) - 1]$
Bias Value of $n = 4$:

$$(2^{n-1}) - 1 = (2^{4-1}) - 1 = 2^3 - 1 = 7$$

Convert from bias notation to signed and then subtract 7 to get binary.

Example 1.10. Convert 0100 to binary

$$0100 \rightarrow 4 \rightarrow 4 - 7 = -3$$

Example 1.11. Signed to binary

$$-3 + 7 = 4 \rightarrow 0100$$

Signed Binary Numbers in 4 bits

Bit Combination	0000	0001	0010	0011	0100	0101	0110	0111
Unsigned Number	0	1	2	3	4	5	6	7
Signed Magnitude	0	1	2	3	4	5	6	7
One's Complement	0	1	2	3	4	5	6	7
Two's Complement	0	1	2	3	4	5	6	7
Biased Notation	-7	-6	-5	-4	-3	-2	-1	0

Bit Combination	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned Number	8	9	10	11	12	13	14	15
Signed Magnitude	0	-1	-2	-3	-4	–5	-6	-7
One's Complement	-7	-6	-5	-4	-3	-2	-1	0
Two's Complement	-8	-7	-6	-5	-4	-3	-2	-1
Biased Notation	1	2	3	4	5	6	7	8

1.3 Fixed Point Arithmetic

1. Addition

   - Overflow - unsigned number, carry indicator
   - Overflow - signed number, overflow indicator

     Example 1.12. Unsigned:

     $$1000 + 0111 = 01111$$

     Example 1.13. 9 + 7 = 1001 + 0111 = (1)000

   - The 1 at the beginning says that it has overflowed.

     Example 1.14. 15 + 15 = 1111 + 1111 = (1)1110

   - For unsigned, carry out a value to the most significant digit. If there's a 0 in the carry out value, there's no overflow. If there's a 1, there's overflow.
   - If the signs of both numbers are the same, there may be an overflow

     Example 1.15. -3 + -1 = 1101 + 1111 = (1)1100 = -4

   - If the signs of the operands are the same as the result, there is no overflow.

     Example 1.16. -8 + -1 = 1000 + 1111 = (1)0111

- If signs of the operands are different, then there is never overflow.

2. Subtraction
   - Signed Numbers: $A - B = A + (-B)$

     Example 1.17. 7 - 6 = 0111 - 0110 = 0111 + 1001 + 1 = 0111 + 1010 = (1)0001
   - If the signs of the operands are the same, then there will be no overflow
   - If the signs of the operands are different, there may be overflow
   - If the signs of the operands and result are the same, there is overflow

     Example 1.18. 7 - (-7) = 0111 - 1001 = 0111 + 0110 + 1 = 0111 + 0111 = 1110

Addition and Subtraction

- Overflow: an integer overflow occurs when an arithmetic operation attempts to create a numeric value which is outside of the range represented with a given number of bits, either larger than the maximum representation value or lower than the minimum representable value

Carry (Flag) Indicator:

- Used for unsigned integers
- For addition, the Carry Flag is set to 1 if there is a carry out of 1 of the most significant bit and set to 0 otherwise
- For subtraction: the Carry Flag is set to 1 if the first number is less than the second number and set to 0 otherwise

Overflow Flag

- Used for signed numbers
- For addition
  - If the signs of both operands are the same, then there may be overflow
    1. If the sign of the result is the same as the signs of the operands then overflow did not occur and the overflow flag is set to 0
    2. If the sign of the result is different from the signs of the operands then overflow occurred and the overflow flag is set to 1
  - If the signs of both operands are different, then overflow did not occur and the overflow flag is set to 0
- For subtraction
  - If the signs of both operands are different, then there may be overflow
    1. If the sign of the result is the same as the sign of the first operand then overflow did not occur and the overflow flag is set to 0

2. If the sign of the result is different from the sign of the second operand then overflow did occur and the overflow flag is set to 1

– If the signs of both operands are the same then overflow did not occur and the overflow flag is set to 0

Bit Shifting

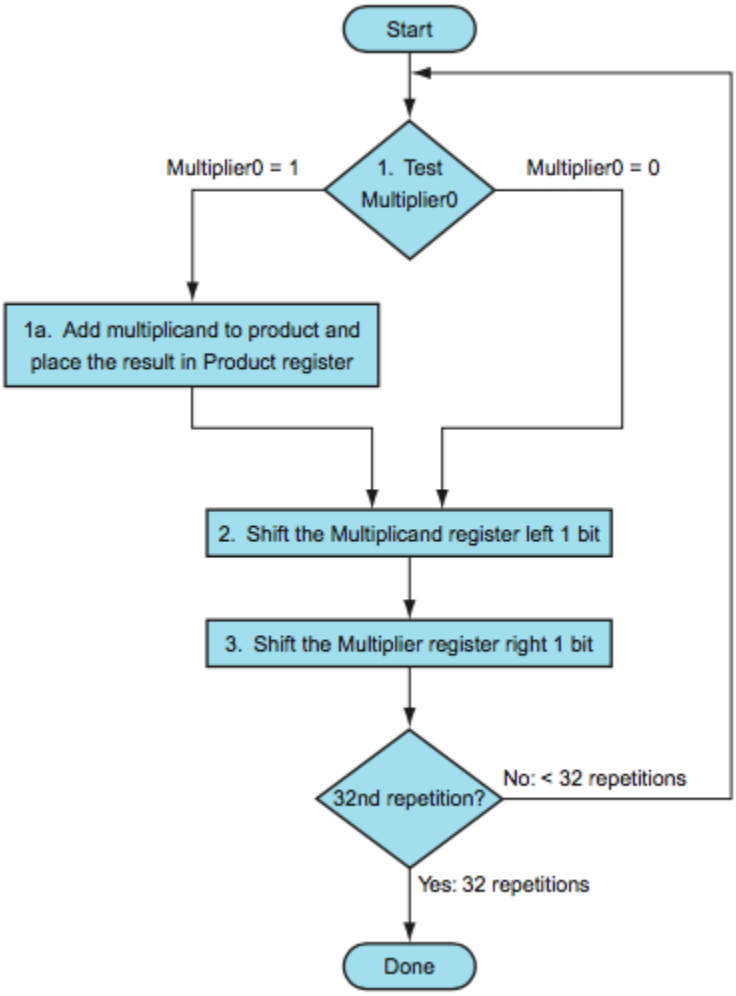
- Logical Shift
- Arithmetic Shift

Direction Shifts

- Left Shift: shift each bit to the left; the most significant bit is dropped and the least significant bit is a zero; similar to multiplying by 2
- Right Shift: shift each bit to the right; the least significant bit is dropped and the most significant bit is a zero if logical shift; if arithmetic shift, it is the sign of the next bit; similar to dividing by 2

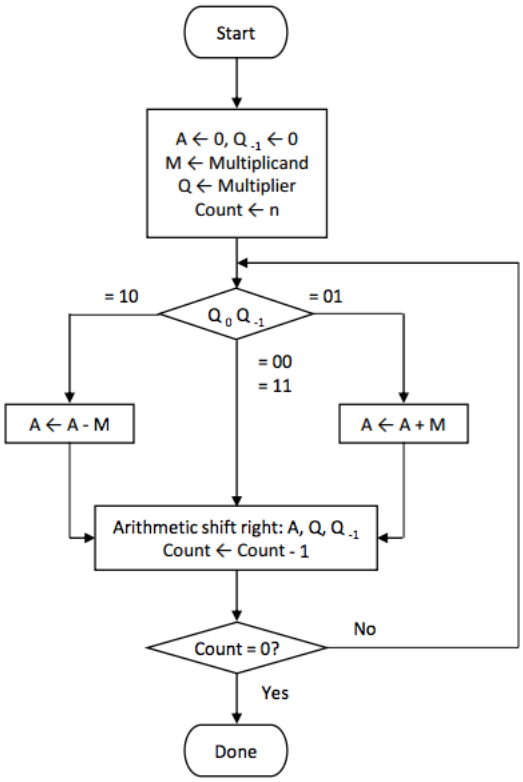
First Multiplication Algorithm

- For unsigned n-bit point numbers
- The Product Register is 2n bits in length, initialized to all zeros
- The Multiplication Register is 2n bits in length with the multiplicand placed to the right half and all zeros in the left half
- The Multiplier Register is n bits in lengths, initialized with the multiplier



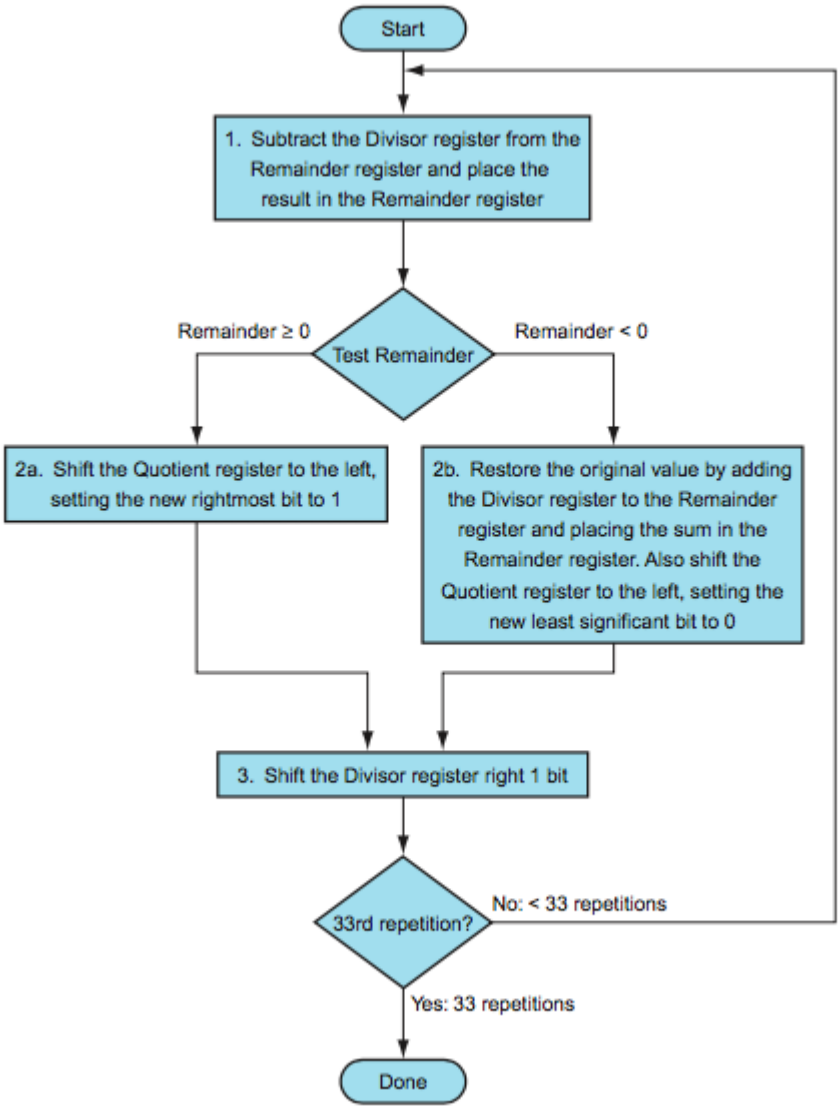
Booth's Algorithm

- For n bit signed point numbers
- n is the number of bits in the Multiplicand
- n is also the number of bits in the Multiplier
- A is the left half of the Product Register
- Q is the right half of the Product Register
- Q_0 is the rightmost (least significant) bit of Q
- Q_{-1} is an extra bit to the right of Q
- A and Q each have n bits
- Therefore the Product Register has 2n bits



Division Algorithm

- For n-bit unsigned fixed point numbers
- The Remainder Register is $2n$ bits in length with the dividend placed in the right half and all zeros in the left half
- The Divisor Register is $2n$ bits in length with the divisor placed in the left half and all zeros in the right half
- The Quotient Register is n bits in length initialized to all zeros



For Division of Signed Binary Integers

- If the signs of the operands (dividend and divisor) are different, the quotient is negative.
- If the signs of the operands are the same, the quotient is positive.

1.4 Floating Point Numbers (Fixed Point Numbers with a Fractional Part)

Definition 1.8. Fixed $< w, b >$: defines the format of a fixed point number where w is the total number of bits to represent the fixed point number and b is the number of those w bits which are to the right of the binary point.

Example 1.19. Binary fixed point numbers given with their format:

- 011.1, 011.0 and 100.1 have format fixed$< 4, 1 >$
- 11.00, 00.01 and 01.01 have format fixed$< 4, 2 >$
- 1100, 0001 and 0101 have format fixed$< 4, 0 >$

Two's complement implements signed fixed point numbers with a fractional part.

Signed Binary Numbers in 4 Bits

Bit Combinations	0000	0001	0010	0011	0100	0101	0110	0111
Fixed$< 4, 1 >$	0	0.5	1	1.5	2.0	2.5	3.0	3.5
Fixed$< 4, 2 >$	0	0.25	0.50	0.75	1.00	1.25	1.50	1.75
Fixed$< 4, 0 >$	0	1	2	3	4	5	6	7

Signed Binary Numbers in 4 Bits

Bit Combinations	1000	1001	1010	1011	1100	1101	1110	1111
Fixed$< 4, 1 >$	-4	-3.5	-3	-2.5	-2	-1.5	-1	-0.5
Fixed$< 4, 2 >$	-2.00	-1.75	-1.50	-1.25	-1.00	-0.75	-0.50	-0.25
Fixed$< 4, 0 >$	-8	-7	-6	-5	-4	-3	-2	-1

Conversion between the Integer Part Separately from the Fractional Part:

Algorithm to convert a nonnegative integer fractional part from decimal to the binary, octal or hexadecimal number system:

1. Initialize, num = "fractional part"
2. Compute num = num $\cdot$ base
3. Integer part of num is the next (binary, octal, hexadecimal) digit of the result
4. Discard the integer part of num
5. Repeat step 2 through 5 until num is zero

Warning: num might not reach zero; in that case, repeat steps 2 through 5 until the result consists of a reasonable number of digits.

Algorithm to convert a nonnegative fractional part from binary, octal, or hexadecimal number system to decimal

1. Initialize, result = 0
2. For each digit from least to most significant, compute result = (result + digit) / base

Algorithm to convert a nonnegative fractional part from binary, octal or hexadecimal number system to one of the other two number system

1. Group fractional digits into groups of 2, 3 or 4 (binary, octal, hexadecimal respectively) and convert into integer values from left to right of the point
2. Convert the nonnegative integer fractional part attained into binary, octal or hexadecimal using algorithm previously written

Definition 1.9. Normalized Form: a number in scientific notation

Definition 1.10. Denormalized: a floating number that has a 0 in the non-fractional part of the integer in scientific notation

Floating Point Binary Numbers (IEEE 754 standard)

- Single Precision - 32 Bits - Bias value of 127
  - Signed - 1 bit - 0 if positive, 1 if negative
  - Exponent - 8 bits - range: 126 to 127
  - Significand / Mantissa - 23 bits
- Double Precision - 64 Bits - Bias value of 1023
  - Signed - 1 bit
  - Exponent - 11 bits
  - Significand / Mantissa - 52 bits

Example 1.20. Express 12.3125 in decimal into binary normalized.

$$0.3125 \cdot 2 = 0.625$$
$$0.625 \cdot 2 = 1.25$$
$$0.25 \cdot 2 = 0.5$$
$$0.5 \cdot 2 = 1.0$$

12.3125 in binary is (1100.0101).

$$1100.0101 \cdot 2^0 = 1.1000101 \cdot 2^3$$

In single precision, the significand is 10010100.. The exponent bit is 127 + 3 = 130 or 10000010. The sign bit is 0. Thus 12.3125 is 01000001010010100 in binary form.

Note: To move the point to the right, decrease the exponent; to move the point to the left, increase the exponent.

IEEE 754 encoding of floating point numbers (a separate sign bit determines the sign.)

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Floating Point Arithmetic Operations

- Addition or Subtraction
  1. In order to add the two numbers, the exponents of the two numbers need to be the same. Convert the number with the smaller exponent to the equivalent number whose exponent is the same as the other number.
  2. Add or subtract the mantissa of the two numbers
  3. Normalize the result

$$1.110 \cdot 2^2 + 1.000 \cdot 2^0 = 1.110 \cdot 2^2 + 0.01 \cdot 2^2 = 10.000 \cdot 2^2 = 1.0000 \cdot 2^3$$

- Overflow and Underflow
  - Overflow occurs when the result of a floating point operation is larger than the largest positive value or smaller than the smallest negative value; in other words, the magnitude (exponent) is too large to represent
  - Underflow occurs when the result of a floating point operation is smaller than the smallest positive value or larger than the largest negative value; in other words, the magnitude (exponent) is too small to represent
- Multiplication
  1. Add the exponents; if the exponents are in biased form, then subtract the bias from the sum
  2. If exponent overflow or underflow, then report and stop
  3. Multiply the significands considering their signs
  4. Normalize the result
  5. Round the result

- Division
  1. Subtract the exponents; if the exponent is in biased form, then add the bias to the difference
  2. If exponent overflow or underflow, then report and stop
  3. Divide the significands
  4. Normalize the result
  5. Round the result

2 MIPS Assembly Language Programming

2.1 MIPS Assembly Language Programming

MIPS Subroutines: subprograms (procedure, function, method)

Definition 2.1. Subroutines: a tool programmers use to structure programs to both to make them easier to understand and to allow code to be reused

Each subroutine is placed after the main section of the program, one after another. Each starts with a label and ends with a jr $ra.

The Steps Needed for the Execution of a Subroutine:

1. Put parameters in a place where the subroutine can access them.
2. Transfer control to the subroutine.
3. Acquire the storage resources needed for the subroutine.
4. Perform the desired task.
5. Put the result value in a place where the calling (sub)program can access it.
6. Return control to the point of origin, since a subroutine can be called from several points in a program.

Subroutine Resources

- Registers used for calling subroutines
  - $a0 - $a3: Four argument registers in which to pass parameters
  - $v0 - $v1: Two value registers in which to return values
  - $ra: One return register to return to the point of origin
- The jump-and-link instruction (jal) jumps to an address and simultaneously saves the address of the following instruction in register $ra
- At the end of a subroutine, the jump register jr is used at the end of a subroutine to return to the calling (sub)program, jr $ra

Program in C++:

```
int leafExample(int g, int h, int k, int j){      int f;
    if (i == j) f = g + h
    else f = g - h
    return f;
}
```

Program in MIPS:

```
    $a0 → g
$a1 → h, $a2 → i, $a3 → j, $s0 → f
b $a2, $a3, else
bne $a2, $a3
add $s0, $a0, $a1
j endif #only needed if there's an else statement
else sub $s0, $a0, $a1
endif:

leafExample:
    bne #a2, #a3, else
    add $s0, $a0, $a1
    j endif
else sub $s0, $a0, a1
endif move $v0, $s0
    jr $ra
```

MIPS Runtime Stack

- Often, a subroutine does call either subroutine or even itself.
- Therefore, to ensure the program and all called subroutines work properly, registers used by the current called subroutine need to be saved before the current called subroutine can begin executing its own code.
- The runtime stack and the Stack Pointer register $sp are the tools for saving register contents
- Stack is implemented LIFO

Program in C++:

```
int leafExample(int g, int h, int k, int j){      int f;
    f = (g + h) - (i + j);
    return f;
}
```

Program in MIPS leafExample:
pushes
add $s0, $a0, $a1
add $t0, $a2, $a3
sub $s0, $t0, t1
move $v0, $s0
pop
jr $ra

int leafExample(int g, int h, int i, int j) int f; f = (g + h) - (i + j); return f;
$a0 - g. $a1 - h, $a2 - i, $a3 - j

In MIPS with Stack:
leafExample
addi $sp, $sp, 0
sw $t1, 8($sp)
sw #t0, 4($sp)
sw $
—

At the Beginning of a Subroutine:

- the $sp register starts at a high address and moves towards lower addresses
- To push registers onto the stack, subtract (the number of registers times 4) from $sp
-
- Remove each register from the stack
- Add (the number of registers times 4) to $sp

Register Saving Conventions for MIPs Programmers

- Caller (sub)program - the main part of the program or subroutine calling another subroutine
- Callee subroutine - the subroutine called
- If the callee uses temporary registers ($t0 - $t9), it is the caller (sub)program which should save them on this stack
- If the callee uses saved registers ($s0, $s7), it is the callee subroutine which should save them on the stack

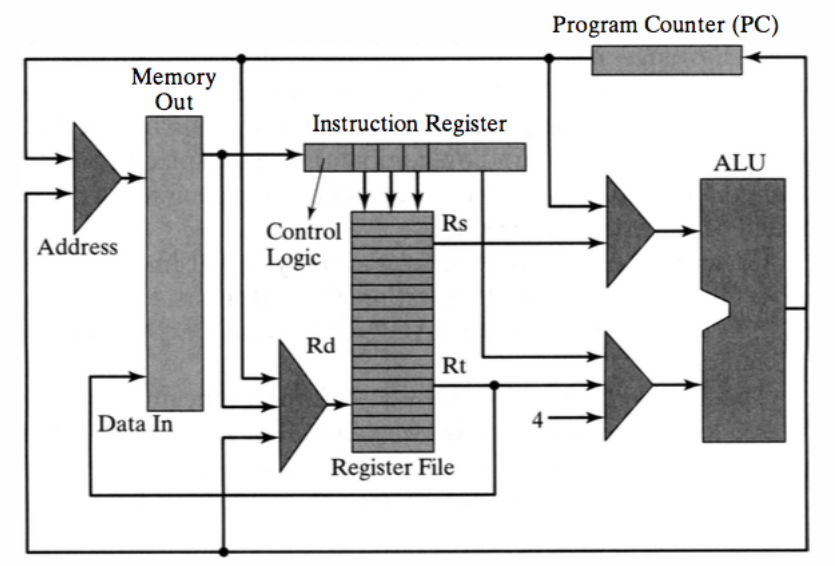
Recursion

- Often, a subroutine does call other subroutine and even itself
- Therefore, the callee subroutine needs to store the argument registers it uses as well as the return address in $ra onto the stack

2.2 MIPS Architecture

2.3 MIPS ALU Design

MIPS Datapath Diagram



Register File

Register	Number	Usage
zero	0	Constant 0
at	1	Reserved for the assembler
v0	2	Used for return values from function calls
v1	3	
a0	4	Used to pass arguments to functions
a1	5	
a2	6	
a3	7	
t0	8	Temporary (Caller-saved, need not be saved by called functions)
t1	9	
t2	10	
t3	11	
t4	12	
t5	13	
t6	14	
t7	15	
s0	16	Saved temporary (Callee-saved, called function must save and restore)
s1	17	
s2	18	
s3	19	
s4	20	
s5	21	
s6	22	
s7	23	
t8	24	Temporary (Caller-saved, need not be saved by called function)
t9	25	
k0	26	Reserved for OS kernel
k1	27	
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address for function calls

Note: All registers have a $ in front of it.

Definition 2.2. .data represents the memory area of the program

Definition 2.3. .text: code area of the program

Assembler Directives:
Macros:
Syntax for MIPS Assembly Language Instructions:

[label:] Op-code [operand,] [operand,] [Operand] [#command]

The first operand is usually the destination and the others are sources.

Types of R format: Op-code $rd, $r

- add $rd, $rs, $rt
- addu (unsigned) $rd, $rs, $rt
- addi (immediate, numerical literal): $rd, $rs, Imm
- addiu (immediate unsigned): $rd, $rs, Imm
- sub $rd, $rs, $rt
- subu (unsigned) $rd, $rs, $rt
- and: $rd, $rs, $rt
- or: $rd, $rs, $rt
- XOR (exclusive or): $rd, $rs, $rt
- NOR: $rd, $rs, $rt
- mult: $rd, $rs, $rt (64 bits)
- mfhi: $rd - get upper half of product
- mfo: $rd - get lower half of product
- div: $rs, $rt
- divu (unsigned): $rs, $rt
- mfhi: $rd - get remainder
- mflo $rd - get quotient

Types of I Format - Data Transfer

- lw (load word): $rt, offset($rs)
- sw (store word): $rt, offset($rt)

A word is a certain number of bits. A byte is composed of 8 bits. In MIPS, a word is 32 bites, or 4 bytes. Variables are represented as .word, each 4 bytes. The first variable's address is 0; the next one's address is 4, the next one's address is 8, etc. The memory address is found by adding $rs and offset. Main is represented as .text

Macro Instructions

- Load Address la: $rs, Label
- Load Immediate li: $rs, Imm

Conditional Branches in I Format

- Branch If Equal beq: $rs, $rs, Label
- Branch if Not Equal bne: $rs, Label

J Format

- Jump j: Label
- Jump and Link jal: Label
- Jump and Link Register jalr: $rd, $rs
- Jump Register jr: $rs

Input and Output

- Read an integer:
  li $v0, 5
  syscall
  #The integer read in is placed in register $v0
- Print an integer:
  $v0, a0
  li $v0, 1
  syscall
  # Place the integer to print in register $a0
- Read a String:
  # Place the address of the string's storage into register $a0. Place the maximum number of chars to read in into register $a1
  la $a0, myString
  li, $a1, 100
  li, $10, 8
  syscall
- Print a String:
  la $a0, inputPrompt
  li, $v0, 4
  syscall
  # Place the address of the string's storage into register

String Literal

- .ASCII
- .ASCIIz - null terminated - add a 0 after the least character in a string
- new line \n

String Variables

- .space n - allocates n number of bytes for storage (use one additional byte for null) myString .space 101

Vector - Arrays
an array of Integers
C: .space n
n must be a multiple of 4
Note: The memory address is $rs + offset. Therefore
lw $rt, offset($rs) will load a value into the array.
sw $sw, $rt, offset($rt) will save a value into the array

Branch of Equal: beq $rs, $rd, Label
Branch of not Equal: bne $rs, $rd, Label
Set if Less than: slt, $rd, $rs, $rt
if rs > rt, then the first operand gets a value of 1, else 0
Branch if less than: ble, $rd, $rs, $rt

2.4 Algorithm Development in Pseudocode

3 Combinational Logic

Logic Design
Boolean Values: 0, 1 - boolean literals

Binary Values

- High voltage - true, 1, or asserted
- Low voltage - false, 0, or deasserted

Logic Blocks

- Logic Blocks are categorized as one of two types based on whether or not the block contains memory
- Combinational Logic Blocks: logic blocks without memory using the processes of combinational logic; the output of a combinational logic block depends only on it current inputs
- Sequential Logic Blocks: logic blocks with memory using the processes of sequential logic; the output of a sequential logic block can depend on both the inputs and the value stored in memory, which is called the state of the logic block

Truth Tables

- A table to define the output value for each combination of the inputs
- For a logic block of n inputs, there are 2^n entries in the truth table
- Defines any combinational logic block but grows in size quickly
- A shorthand lists only the input combinations which have an output value of 1

Boolean Logic

- Developed by George Boole, a nineteenth century English Mathematician
- Defines a combinational logic block using logic equations and Boolean algebra
- All inputs and outputs (Boolean variables) have a value of 0 or 1
- Three Boolean Opeartors
    1. OR: written as +; the result of OR is 1 if either input is 1; it is also called logical sum
    2. AND: written as $\cdot$: the result of AND is 1 only if both values are 1; it is also called logical product
    3. NOT: written as $\bar{A}$: the result of NOT is 1 if the input value is 0 and is 0 if the input value is 1; the NOT operator produces the negation or inversion of the input value

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

Boolean Algebra Laws

- Identify Law: $A + 0 = A$ and $A \cdot 1 = A$
- Zero and One Law: $A \cdot 0 = 0$ and $A + 1 = 1$
- Inverse Law: $A + \bar{A} = 1$ and $A \cdot \bar{A} = 0$
- Commutative Laws: $A + B = B + A$ and $A \cdot B = B \cdot A$
- Associative Laws: $A + (B + C) = (A + B) + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive Laws: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ and $A + (B \cdot C) = (A + B) \cdot (A + C)$
- DeMorgan's Law: $\overline{A + B} = \bar{A} \cdot \bar{B}$ and $\overline{A \cdot B} = \bar{A} + \bar{B}$

Example 3.1.

$$\begin{aligned}
(\bar{x} \cdot \bar{y} \cdot z) + (\bar{x} \cdot y \cdot z) + (x \cdot \bar{y}) &= (\bar{x} \cdot z \cdot \bar{y}) + (\bar{x} \cdot z \cdot y) + (x \cdot \bar{y}) \\
&= ((\bar{x} \cdot z) \cdot \bar{y}) + ((\bar{x} \cdot z) \cdot y) + (x \cdot \bar{y}) \\
&= (\bar{x} \cdot \bar{z}) \cdot (\bar{y} + y) + (x \cdot \bar{y}) \\
&= (\bar{x} \cdot z) \cdot (1) + (x \cdot \bar{y}) \\
&= (\bar{x} \cdot z) + (x + \bar{y}) \\
(\bar{x} \cdot \bar{y} \cdot \bar{z}) + (\bar{x} \cdot y \cdot \bar{z}) + (x \cdot \bar{y} \cdot \bar{z}) + (x \cdot y \cdot \bar{z}) &= \Big((\bar{x} \cdot \bar{y}) + (\bar{x} + y) + (x + \bar{y}) + (x + y)\Big) \cdot \bar{z} \\
&= \Big(\Big(\bar{x} \cdot (\bar{y} + y)\Big) + \Big(x \cdot (\bar{y} + y)\Big)\Big) \cdot \bar{z} \\
&= \Big((\bar{x} \cdot 1) + (x \cdot 1)\Big) \cdot \bar{z} \\
&= (\bar{x} + x) \cdot \bar{z} \\
&= 1 \cdot \bar{z} \\
&= \bar{z}
\end{aligned}$$

Example 3.2.

$$A + (A \cdot B) = (A \cdot 1) + (A \cdot B) = A \cdot (1 + B) = A \cdot 1 = A$$

$$A \cdot (A + B) = (A + 0) \cdot (A + B) = A + (0 \cdot B) = A + 0 = A$$

Operator Precedence: NOT, AND, OR

Logic Gates

- And Gate (capital D with 2 inputs and 1 output): implements the And logical operator
- Or Gate (curved capital D with 2 inputs and 1 output): implements the Or logical operator
- Not Gate (triangle followed by small circle with 1 input and 1 output): implements the Not logical operator, also called inverter

Functional Completeness:

- The original set of Boolean operations is (And, Or, Not)
- A set of Boolean operators is functionally complete if all other Boolean operators can be constructed from this set
- The set of Boolean operators (And, Not) is functionally complete
- Because, the OR operator can be defined in terms of And and Not
- How? Using two Boolean properties

  - Double negation
  - DeMorgan's Law
  - $\overline{A \cdot B} = \bar{A} + \bar{B}$
- $\overline{\bar{A} \cdot \bar{B}} = \bar{\bar{A}} + \bar{\bar{B}} = A + B$
- This set of Bollean operation (Or, Not) is functionally complete
- Because, the And operator can be defined in terms of Or and Not
- How? Using 2 Boolean operations
  - Double Negation $\bar{\bar{A}} = A$
  - DeMorgan's Law $\overline{A + B} = \bar{A} \cdot \bar{B}$
- $\overline{\bar{A} + \bar{B}} = \bar{\bar{A}} \cdot \bar{\bar{B}} = A \cdot B$

Exclusive OR ($\oplus$, double curved capital D): returns a 1 if both values are different or a 0 if both values are the same

$$A \oplus B = (\bar{A} \cdot B) + (A \cdot \bar{B})$$

A	B	$A + B$	$A \oplus B$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

Logic Gates:

- Nand Gate (capital D followed by a small circle, has 2 input an 1 output): $\overline{A \cdot B}$; is a functionally complete set
- Not Gate: implemented using a Nand gate (square, capital D, small circle; has 1 input and 1 output)

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

- And gate implemented using Nand gates (capital D, small circle followed by a square, capital D and small circle; 2 input and 1 output) A NAND $B = \overline{A \cdot B}$
- Or gate implemented using Nand gates (for each input, 1 square, capital d, small circle to a capital D followed by a small circle, 1 output) $\bar{A}$ NAND $\bar{B} = \overline{\bar{A} \cdot \bar{B}} = \bar{\bar{A}} + \bar{\bar{B}} = A + B$
- Nor Gate: $\overline{A + B}$, functionally complete set (2 inputs into a curved D followed by a small circle)

- Not Gate implemented using a Nor gate (square, curved D, small circle)
- Or Gate implemented using Nor gates (2 inputs into a curved D, small circle giving 1 output which goes into a square, curved D and a small circle)
- And Gate implemented using Nor Gates (2 of square, curved d, small circles into a curved D and small circle, 1 output)

FIGURE B.2.1 Standard drawing for an AND gate, OR gate, and an inverter, shown from left to right. The signals to the left of each symbol are the inputs, while the output appears on the right. The AND and OR gates both have two inputs. Inverters have a single input.

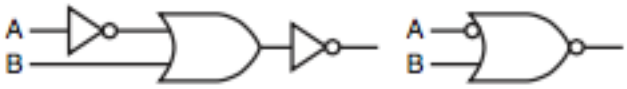
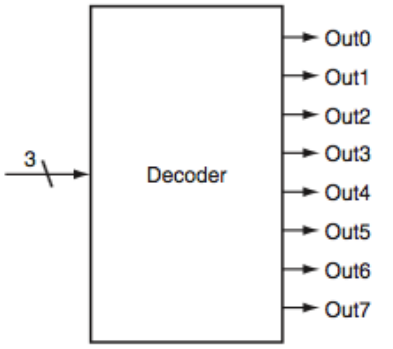


FIGURE B.2.2 Logic gate implementation of $\overline{\overline{A} + B}$ using explicit inverts on the left and bubbled inputs and outputs on the right. This logic function can be simplified to $A \cdot \overline{B}$ or in Verilog, `A & ~ B`.

Decoder

- Decoder: a combinational logic circuit that converts a binary integer value to an associated pattern of output bits
- For the decoders discussed in this class, only output bit is asserted for each binary integer input value
- With n inputs, the decoder has 2^n outputs
- They are used in a wide variety of applications, including data demultiplexing, seven segment displays and memory address decoding

- A 3 to 8 decoder has 3 inputs and up to 8 outputs



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

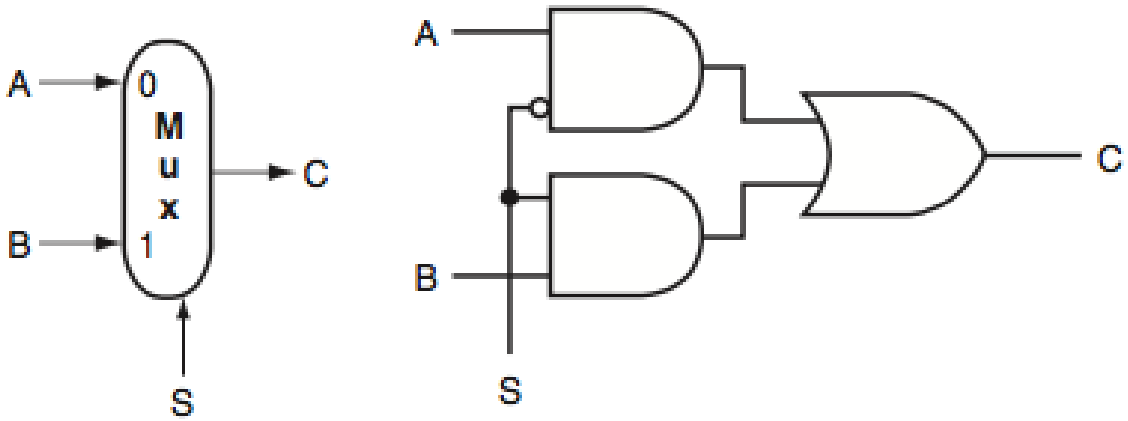
FIGURE B.3.1 A 3-bit decoder has 3 inputs, called 12, 11, and 10, and $2^3 = 8$ outputs, called Out0 to Out7. Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.

Simple Encoder

- A simple encoder circuit is a one-hot to binary converter. That is, if there are 2^n input lines, and at most only one of these will ever be asserted, the binary code of this 'hot' line is produced on the n-bit output lines
- Works the opposite of a decoder
- 8 to 3 Encoder - works the opposite as where the 8 outputs on the right are inputs and the 3 inputs on the left are outputs

Multiplexer

- Called a selector because its output is one of the inputs selected by a control, selector bit(s); only has one output
- 2 to 1 Multiplexer



N-Bit Multiplexer

- An n-input multiplexer needs $\lceil \log_2 n \rceil$ selector (control) bits
- The multiplexer consists of
  - A decoder with n outputs signals, each matched to a unique multiplexer input
  - An array of n AND gates, each combining one of the multiplexers inputs with an output signal from the decoder
  - A single large OR gate which combines the outputs of the AND gates
- The multiplexer inputs are numbered from 0 to $n - 1$
- The selector bits represents a binary number
- Each selector bit binary combination selects the multiplexer input corresponding to the binary number represented by that selector bit binary combination
- The currently selected multiplexer input connects to the multiplexer output
- If the selector bits changes, then the multiplexer input connected to the multiplexer output changes

4 to 1 Multiplexer

	Inputs		Outputs	
A	B	C	D	E
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Karnaugh Maps (K-Maps)

- Used to generate a simplified version of the sum of products expressions of a logic function

- Kmap for 3 inputs

	Inputs		Output
A	B	C	D
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

- Kmap for 4 inputs

	Inputs			Outputs
A	B	C	D	E
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

In K-maps, the size of rectangles of 1s must be powers of 2 (1, 2, 4, 8, etc) that don't have to be right next to each other but can overlap. Bigger size is better. Using less inputs is also better. Each rectangle gets a term and whichever input has all 1s goes in the equation. Simplified form of Sum of Products for E:

$$E = (A \cdot C) + (A \cdot \bar{B}) + (B \cdot C \cdot \bar{D})$$

Two Level Logic

- Any logic function can be written as a logic equation in its canonical form where each input variable can be written as itself ad its complement $(A, \bar{A})$ and there are only two levels of operators, one being AND and the other being OR

- This form is a two-level representation and there are two versions:
  - Sum of Products: a logical sum (OR) of products, product terms
  - Product of Sums: a logical product (AND) of sums, sum terms

Logic Function

	Inputs			Outputs	
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

D is true if at least 1 input is true; E is true if exactly 2 inputs are true; F is true only if all 3 inputs are true.

$$
\begin{aligned}
D &= (\bar{A} \cdot \bar{B} \cdot C) + (\bar{A} \cdot B \cdot \bar{C}) + (\bar{A} \cdot B \cdot C) + (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot \bar{B} \cdot C) + (A \cdot B \cdot \bar{C}) + (A \cdot B \cdot C) \\
&= 001 \;\; 010 \;\; 011 \;\; 100 \;\; 101 \;\; 110 \;\; 111 \\
&= (A + B + C) \\
&= 000 \\
E &= (A + B + C) \cdot (A + B + \bar{C}) \cdot (A + \bar{B} + C) \cdot (\bar{A} + B + C) \cdot (\bar{A} + \bar{B} + \bar{C}) \\
&= 000 \;\; 001 \;\; 010 \;\; 100 \;\; 111 \\
&= (\bar{A} \cdot B \cdot C) + (A \cdot \bar{B} \cdot C) + (A \cdot B \cdot \bar{C}) \\
&= 011 \;\; 101 \;\; 110 \\
F &= (A \cdot B \cdot C)
\end{aligned}
$$

Sum of Products

- Construct the sum of products logic equation from the logic function's truth table
- Each truth table entry whose output value is 1 is a product term in the sum of products equation
- A product term is also called a minterm
- A product term is the logic product of all the input variables where the input variable is itself (A) if its value in the truth table is 1 or the complement of itself ($\bar{A}$) if its value in the truth table is 0
- The logic function's sum of products equation is also called the logical sum of minterms

Product of Sums

- Construct the product of sums logic equation from the logic functions's truth table
- Each truth table entry whose output value is 0 is a sum term in the product of sums equation
- A sum term is also called a maxterm
- A sum term is the logic sum of all the input variables where the input variable is itself (A) if its value in the truth table entry is 0 or the complement of itself ($\bar{A}$) if its value n the truth table entry is 1
- The logic function's product of sums equation is also called the logical product of its maxterms

Sum of Products:

$$G = (\bar{A} \cdot B \cdot \bar{C}) + (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot B \cdot \bar{C})$$

$$G = 010100110$$

If this was product of sums, invert the 0s and 1s and puts the 1s in those rows.

$$H = (\bar{A} + B + \bar{C}) \cdot (A + \bar{B} + \bar{C}) \cdot (A + B + \bar{C})$$

$$H = 101011001$$

A	B	C	G	H
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	1	0
1	1	1	0	0

PLA Integrated Circuit

- Programmable Logic Array (PLA): a common structured logic implementation of the sum of products representation of a logic function
- A PLA has a et of inputs and the corresponding inputs complements (implemented with a set of inverters) and two stages of logic
- The first stage is an array of AND gates that form the set of product terms (minterms)

- The second stage is an array of OR gates, each of which forms a logical sum of any number of the product terms

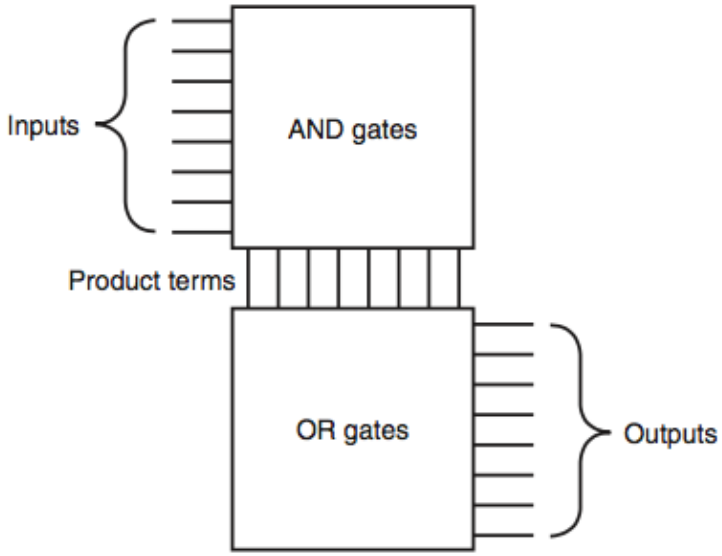
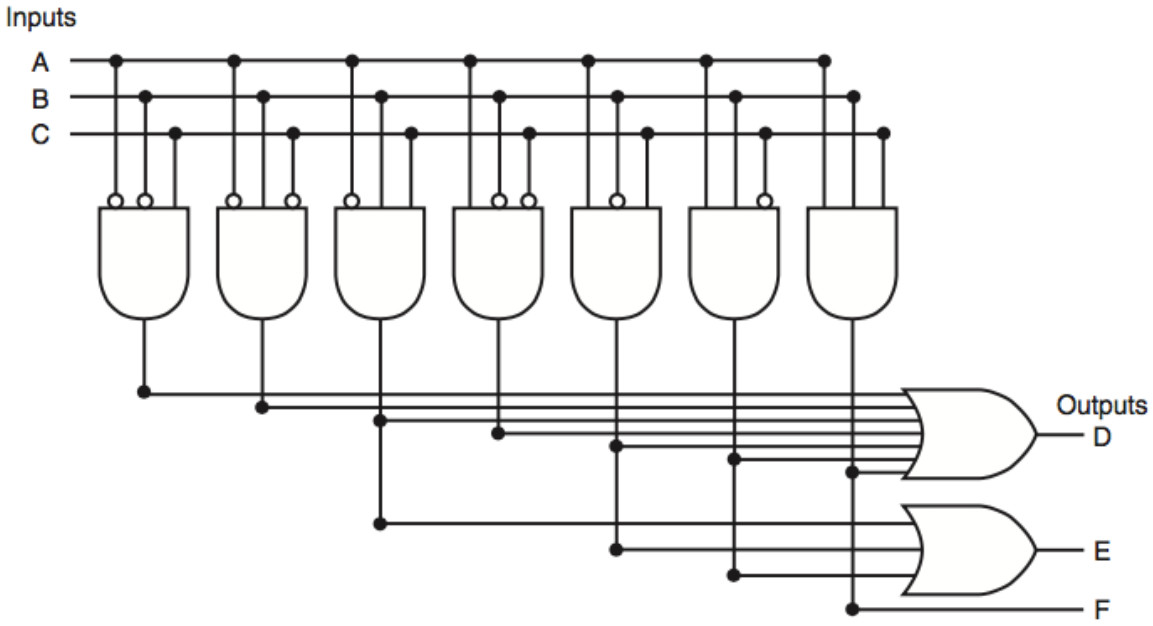
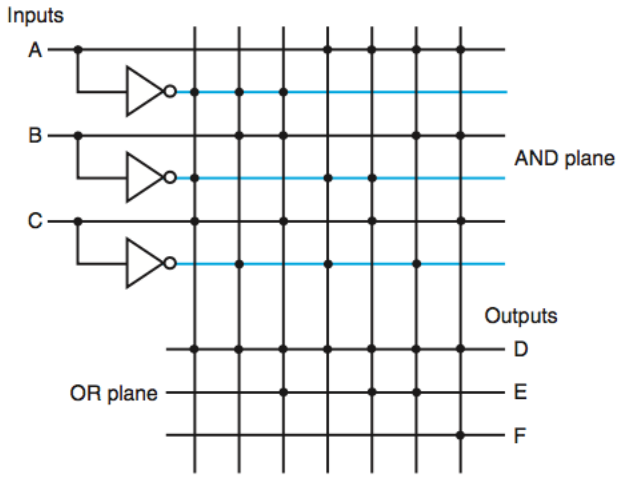


FIGURE B.3.3 The basic form of a PLA consists of an array of AND gates followed by an array of OR gates. Each entry in the AND gate array is a product term consisting of any number of inputs or inverted inputs. Each entry in the OR gate array is a sum term consisting of any number of these product terms.

PLA Logic Circuit Representation Figure



Alternative Diagram: vertical line represents minterm



Read-Only Memory (ROM):

- m inputs, 2^m outputs
- The m inputs are treated as the address to a entry

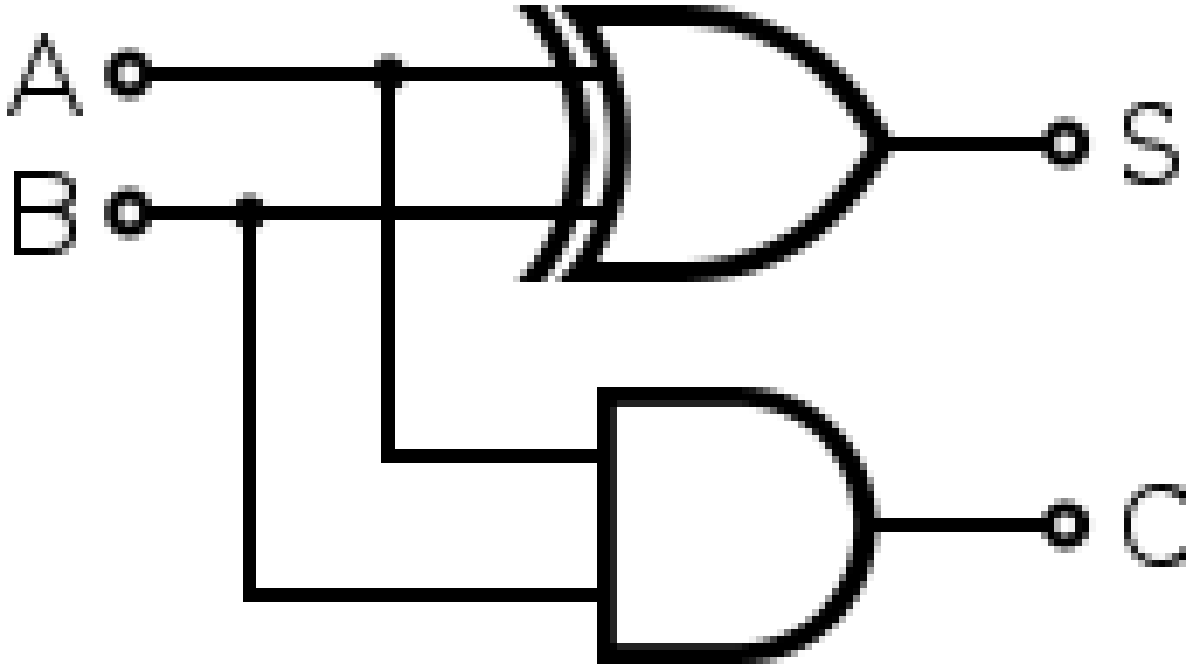
Half Adder Truth Table

Input		Output	
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Half Adder Equations:

$$\begin{aligned} C_{out} &= A \cdot B \\ S &= \bar{A} \cdot B + A \cdot \bar{B} \\ &= A \oplus B \end{aligned}$$

Half Adder Circuit



Full Adder Truth Table

	Input		Output	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

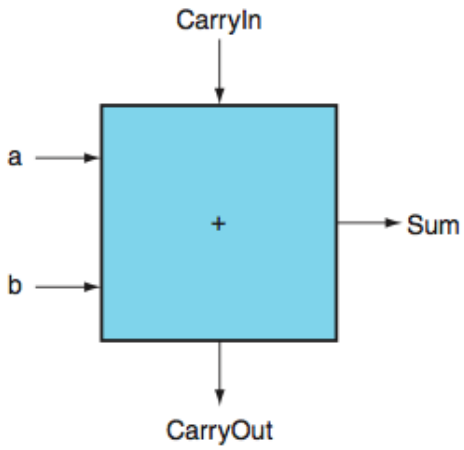
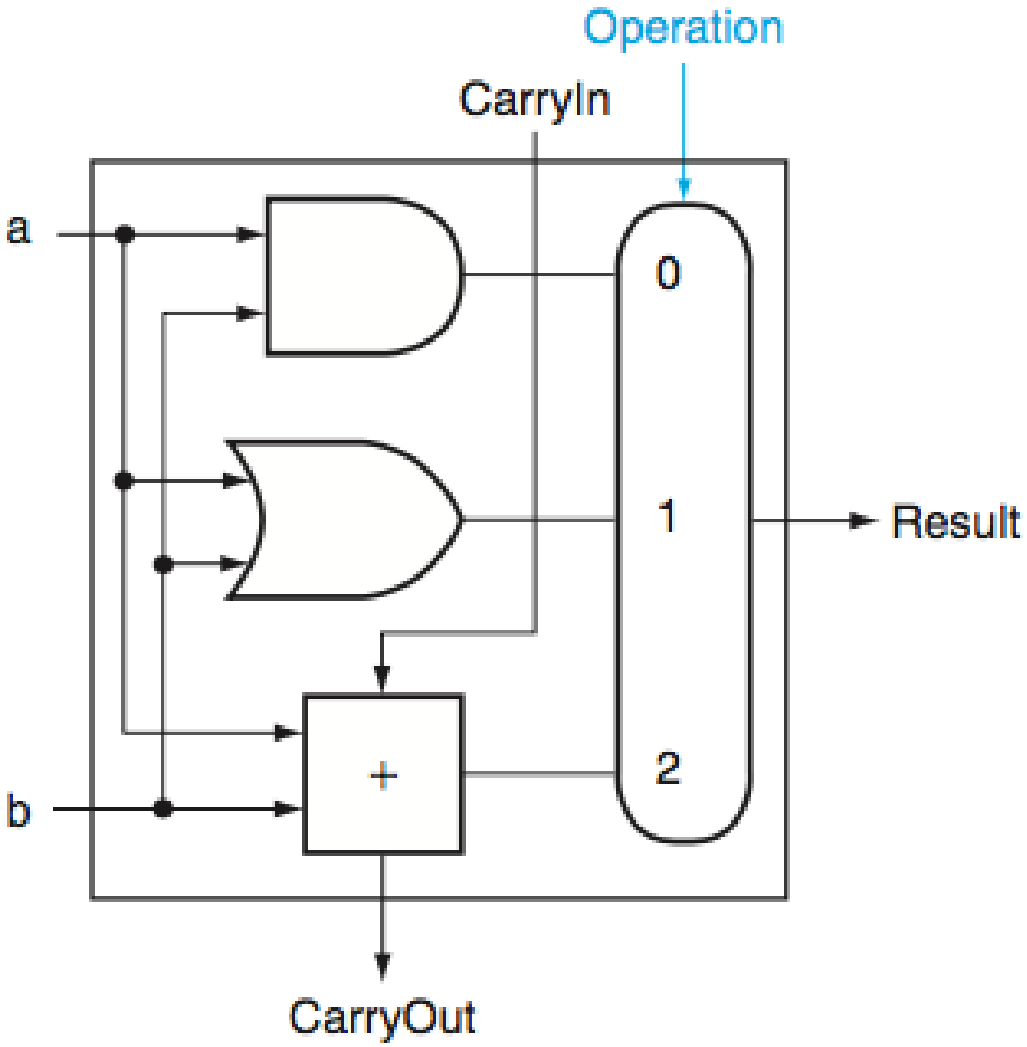
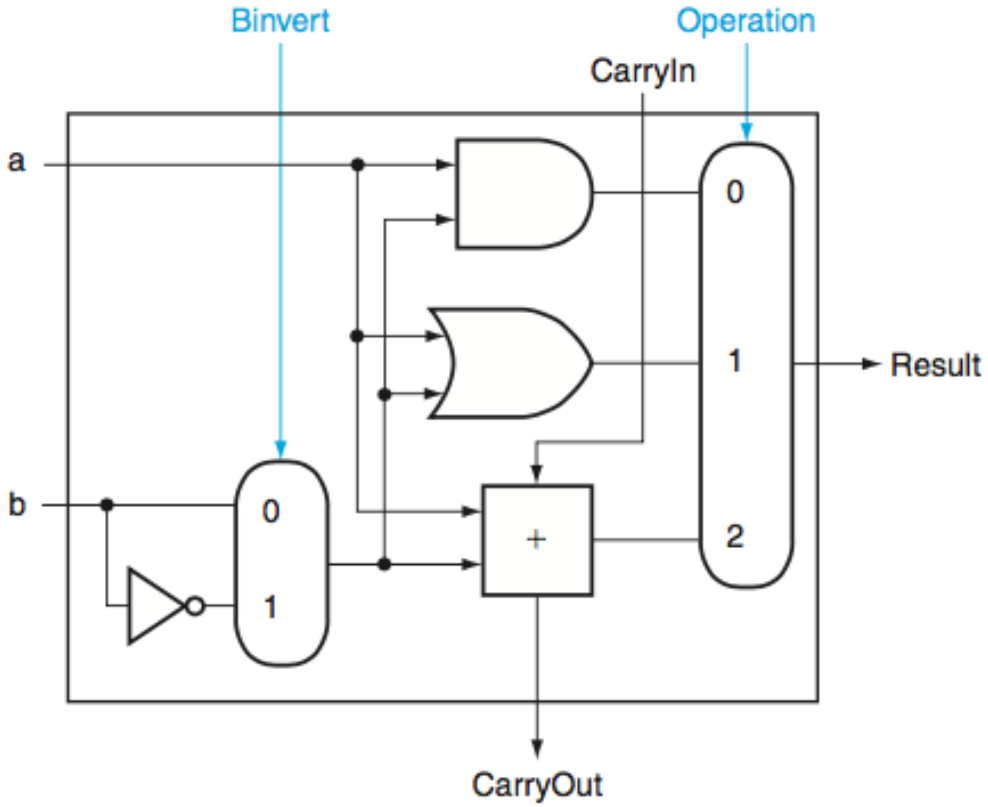


FIGURE B.5.2 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

Inputs			Outputs		
a	b	CarryIn	CarryOut	Sum	Comments
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

FIGURE B.5.3 Input and output specification for a 1-bit adder.


Operation
CarryIn
a
0
1
Result
+
2
b
CarryOut



Full Adder Equations

$$\begin{aligned}
S &= (\overline{C_{in}} \cdot \bar{A} \cdot B) + (\overline{C_{in}} \cdot A \cdot \bar{B}) + (C_{in} \cdot \bar{A} \cdot \bar{B}) + (C_{in} \cdot A \cdot B) \\
&= \overline{C_{in}} \cdot ((\bar{A} \cdot B) + (A \cdot \bar{B})) + C_{in} \cdot ((\bar{A} \cdot \bar{B}) + (A \cdot B)) \\
&= \overline{C_{in}} \cdot (A \oplus B) + C_{in} \cdot (\overline{A \oplus B}) \\
C_{out} &= (\overline{C_{in}} \cdot A \cdot B) + (C_{in} \cdot \bar{A} \cdot B) + (C_{in} \cdot A \cdot \bar{B}) + (C_{in} \cdot A \cdot B) \\
&= (\overline{C_{in}} \cdot A \cdot B) + (C_{in} \cdot A \cdot B) + (C_{in} \cdot \bar{A} \cdot B) + (C_{in} \cdot A \cdot \bar{B}) \\
&= (\overline{C_{in}} + C_{in}) \cdot (A \cdot B) + C_{in} \cdot ((\bar{A} \cdot B) + (A \cdot \bar{B})) \\
&= 1 \cdot (A \cdot B) + C_{in} \cdot (A \oplus B) \\
&= (A \cdot B) + C_{in} \cdot (A \oplus B) \\
&= (A \cdot B) + C_{in} \cdot (A + B)
\end{aligned}$$

Full Adder Circuit

- Full Adder Block Box: 2 inputs and carry in into a box while carry out and sum leaves; adds 2 1 bit numbers
- Ripple Carry Adder: multiple full adder block boxes put together so that multiple numbers can be added together

Propagation Delay

- Propagation decay, or gate decay: The length of time which starts when the input to a logic gate becomes stable and valid to change, to the time that the output of that logic gate is stable and valid to change
- Measured as length of time or number of gates
- In a ripple carry adder circuit, sdd up propagation delay gates for each output in the direction it is traveling in

Carry Propagation

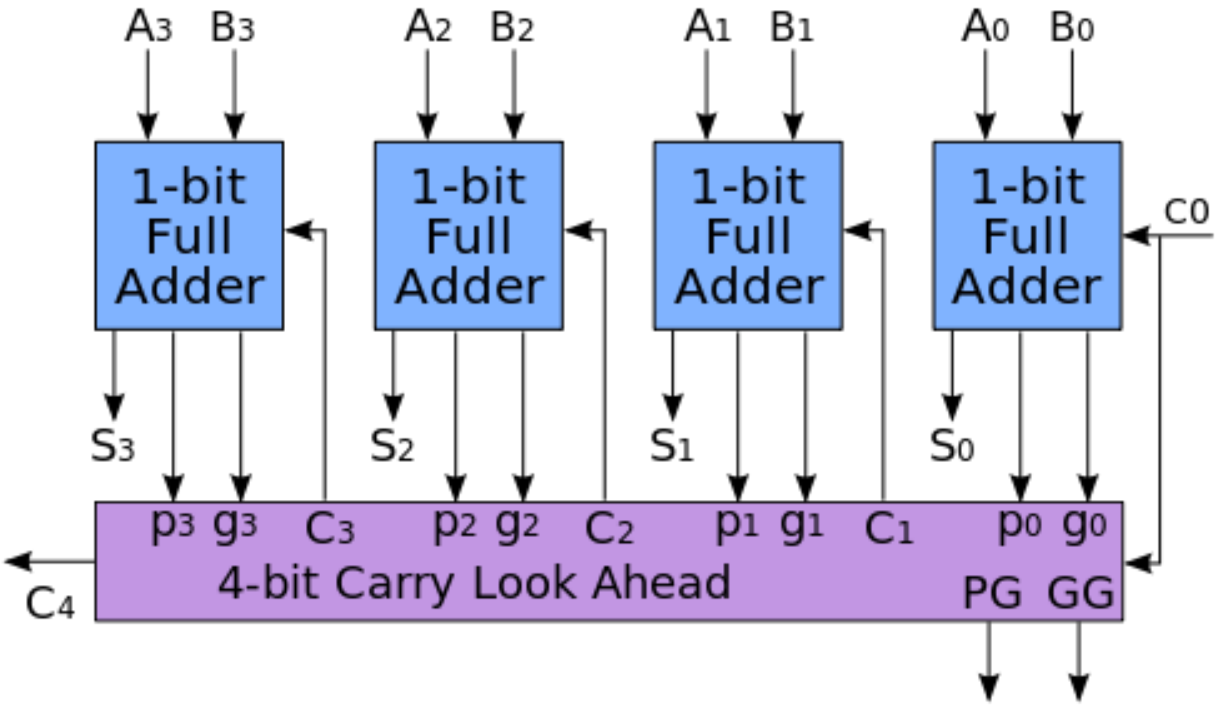
- $C_{out} = (A \cdot B) + C_{in} \cdot (A + B)$
- $G = A \cdot B$ is called the carry generate term
- If A and B are 1 then G is 1 and C_{out} is 1
- $P = A + B$ is called the carry propagate term
- If A or B are 1 but not both then P is 1 but only if C_{in} is 1 would C_{out} be 1
- $C_{i+1} = (A_i \cdot B_i) + C_i \cdot (A_i + B_i)$ for $0 \leq i < 32$
- $C_{i+1} = G_i + P_i \cdot C_i$ for $0 \leq i < 32$

Carry Out Expansions

$$
\begin{aligned}
C_0 &= \\
C_1 &= G_0 + P_0 \cdot C_0 \\
C_2 &= G_1 + P_1 \cdot C_1 \\
&= G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \\
C_3 &= G_2 + P_2 \cdot C_2 \\
&= G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \\
C_4 &= G_3 + P_3 \cdot C_3 \\
&= G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3
\end{aligned}
$$

4 Bit Full Adder with Carry Look Ahead: 4 1-bit full adder connected to a long 4-bit

carry ahead



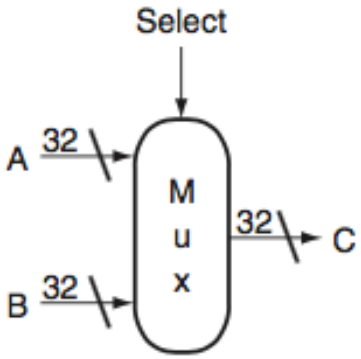
Group Propagate and Group Generate

$$C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$P_G = P_3 \cdot P_2 \cdot P_1 \cdot P_0$$
$$GG = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3$$

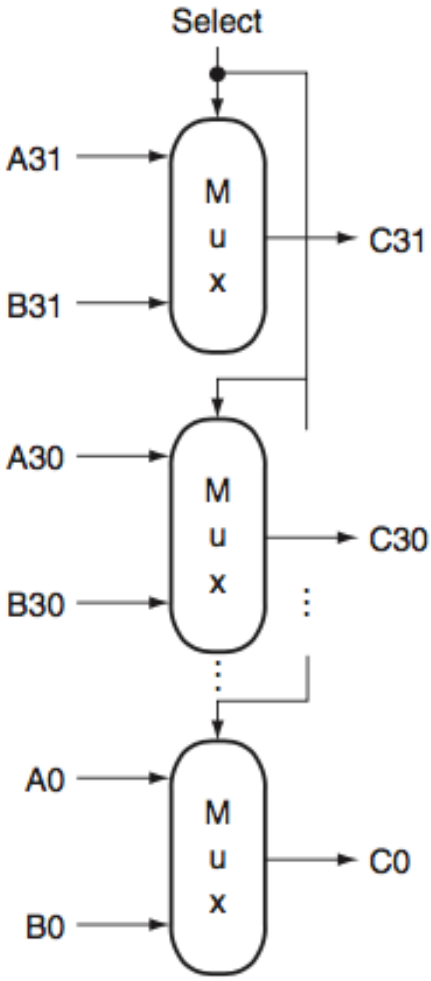
Array of Logic Elements

- If the same combinational operations are performed on each bit of a 32 bit word(s), build an array of logic elements

- 32 Bit wide 2 in 1 multiplexer and compact version of 32 bit 2 in 1 multiplexer



a. A 32-bit wide 2-to-1 multiplexor



b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

1 Bit ALU: AND and OR Operations

- Operation is 1 bit: insert ss
- If operation is 0, result = A AND B

- If operation is 1, result = A OR B

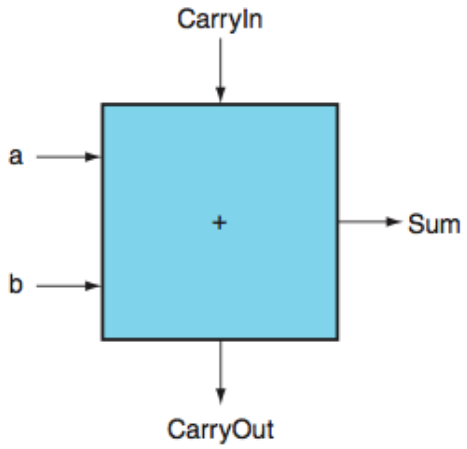
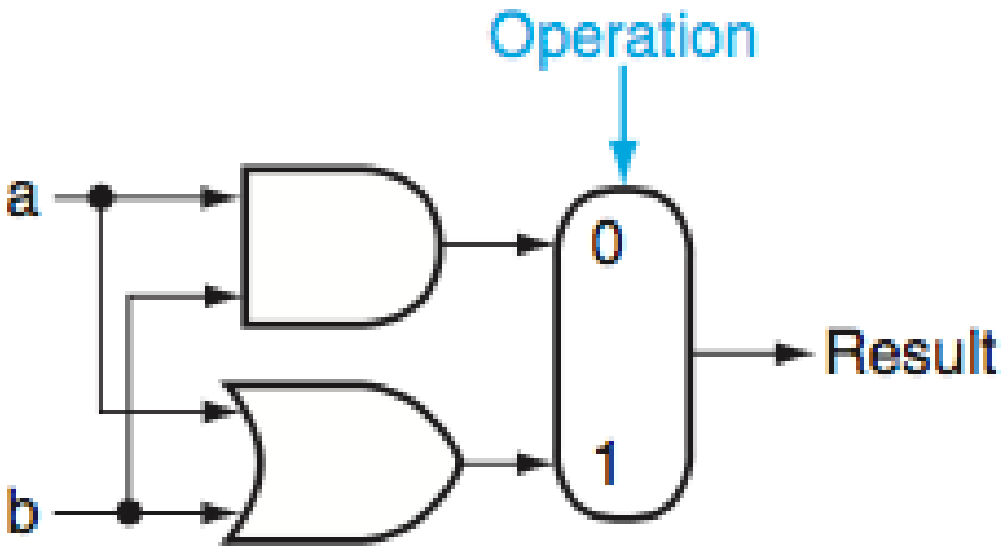


FIGURE B.5.2 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

1 Bit ALU: AND, OR, & Addition Operations

- Operation is now 2 bits
- If operation is 00, result = A AND B
- If operation is 01, result = A OR B
- If operation is 10, result = A + B

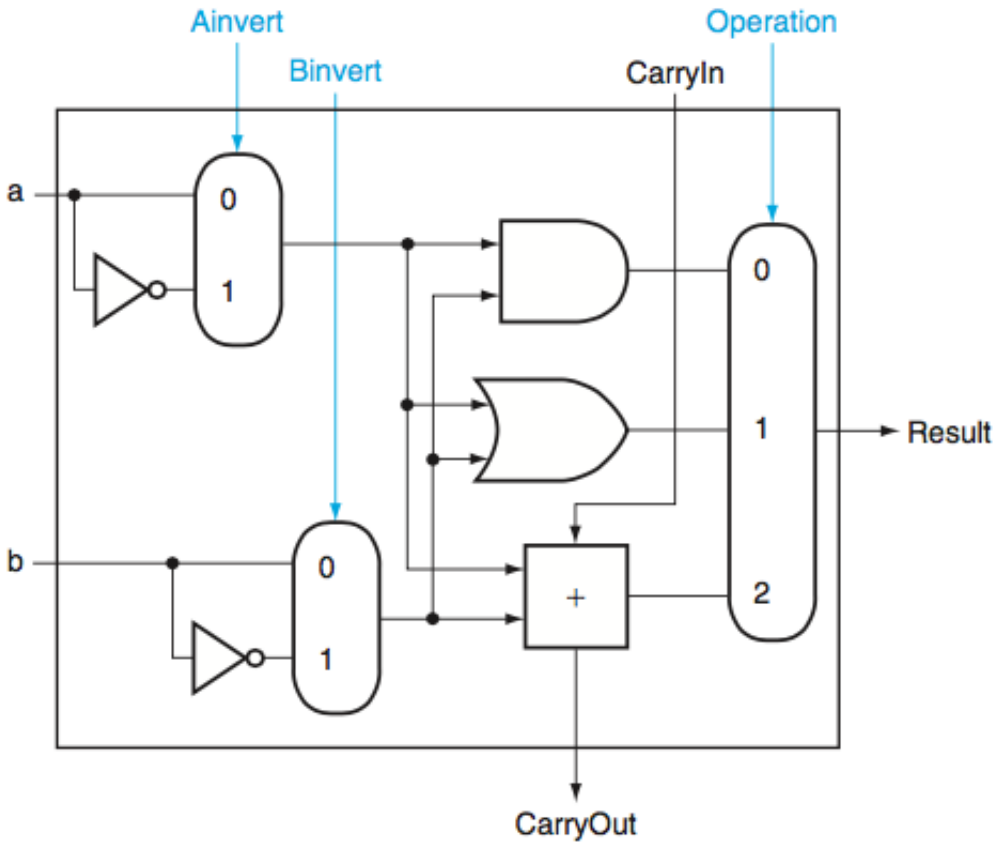


1 Bit ALU: AND, OR, Addition & Subtraction Operations

- Operation is still 2 bits
- If operation is 00, result = A AND B
- If operation is 01, result = A OR B
- If operation is 10, binvert is 0 and carry in (for bit 0) is 0, result = A + B
- If operation is 10, binvert is 1 and carry in (for bit 0) is 1, result = A - B

1 Bit ALU: AInvert Inclusion

- If operation is 00, ainvert is 1 and binvert is 1, result = A NOR B
- If operation is 01, ainvert is 1 and binvert is 1, result is A NAND B



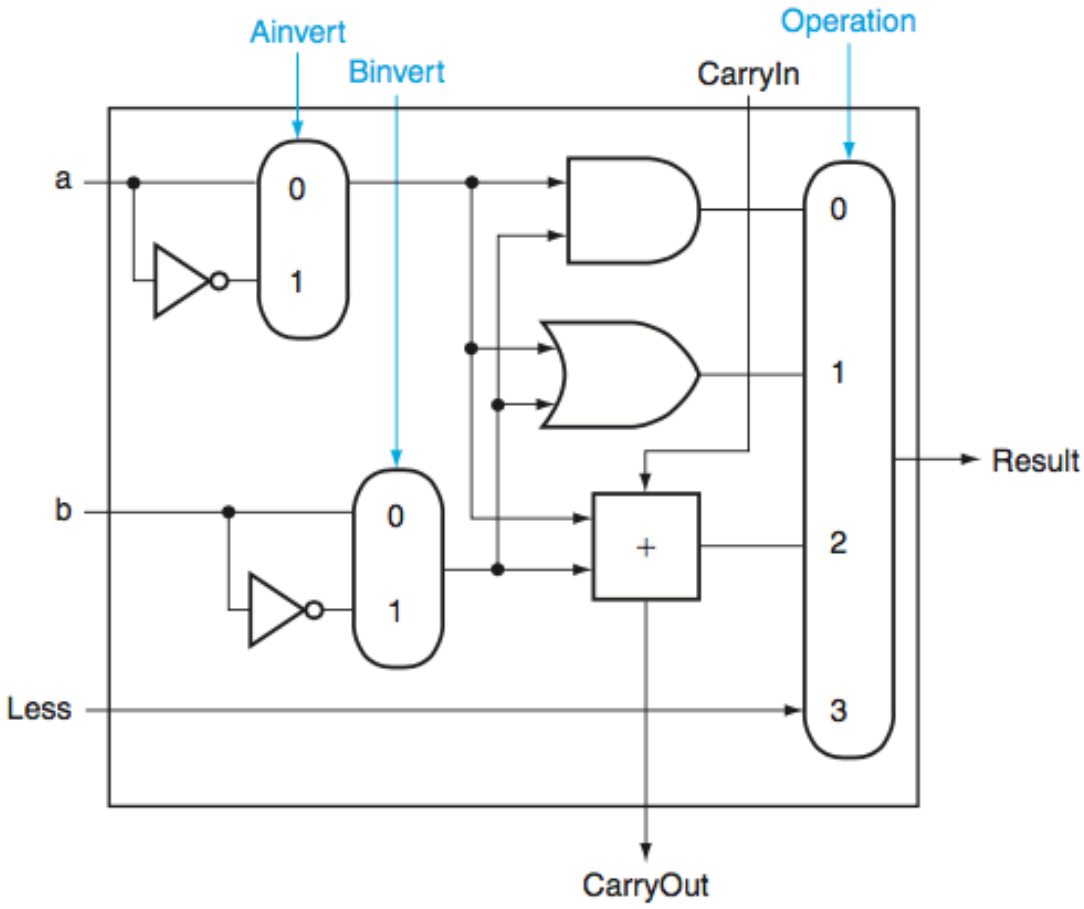
Set on Less than Instruction

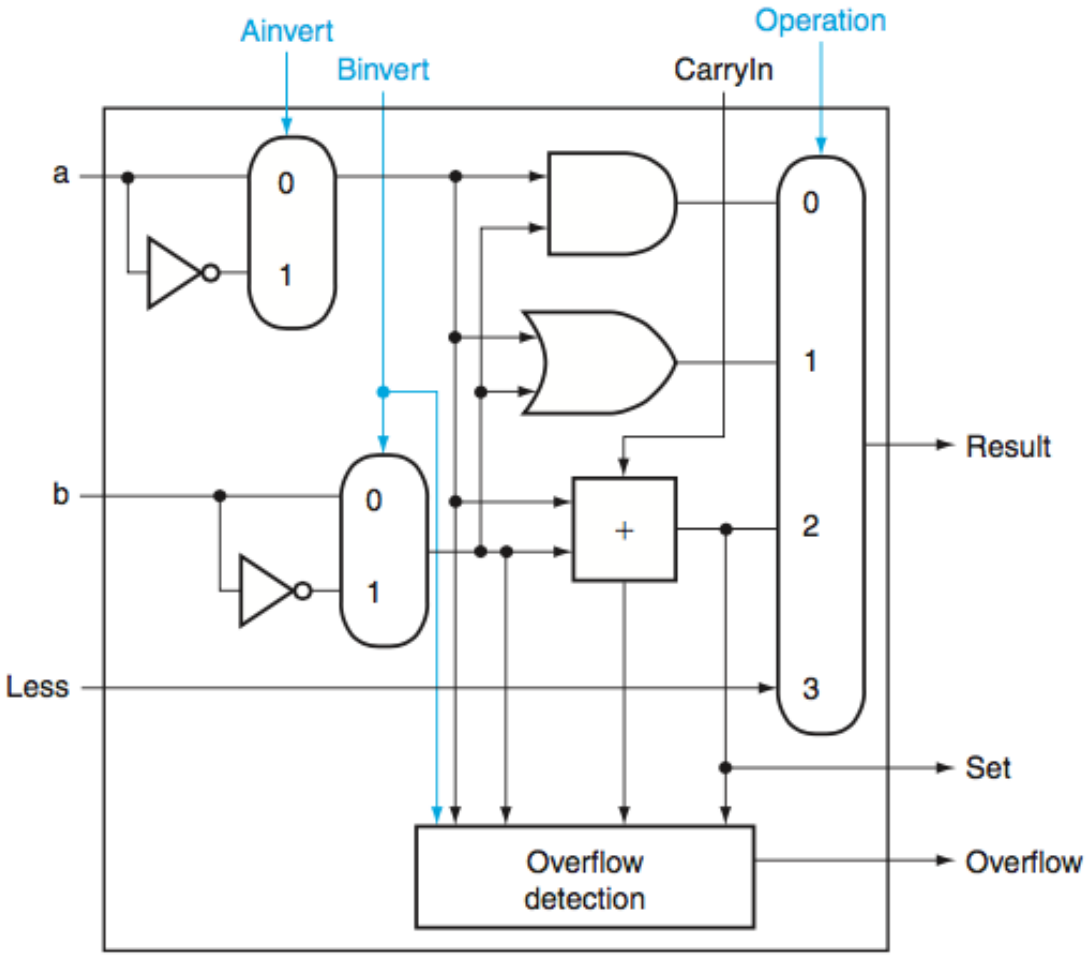
- If (A ¡ B) then result = 1; otherwise result = 0
- A, B and result refer to their 32-bit wide values
- Computing the value of result, the ALU's full adder has to compute A - B

- Hence, set ainvert to 0, binvert to 1 and carry in (for bit 0) to 1
- The answer to (A - B) will be negative if (A ¡ B) andopositive if (A $\geq$ B)
- The 5 output of the full adder of the most significant bit (bit 31) is 1 if (A - B) is negative and 0 if (A - B) is positive

1 Bit ALU: Set on Less Than

- Used for bits 0-30
- The Less input is the bit's value for the SLT instruction





1 Bit ALU for bit-31 (insert pic)

- Includes circuitry for overflow detection (overflow) and a value (set) for SLT
- The Less input is is this bit's value for the SLT instruction

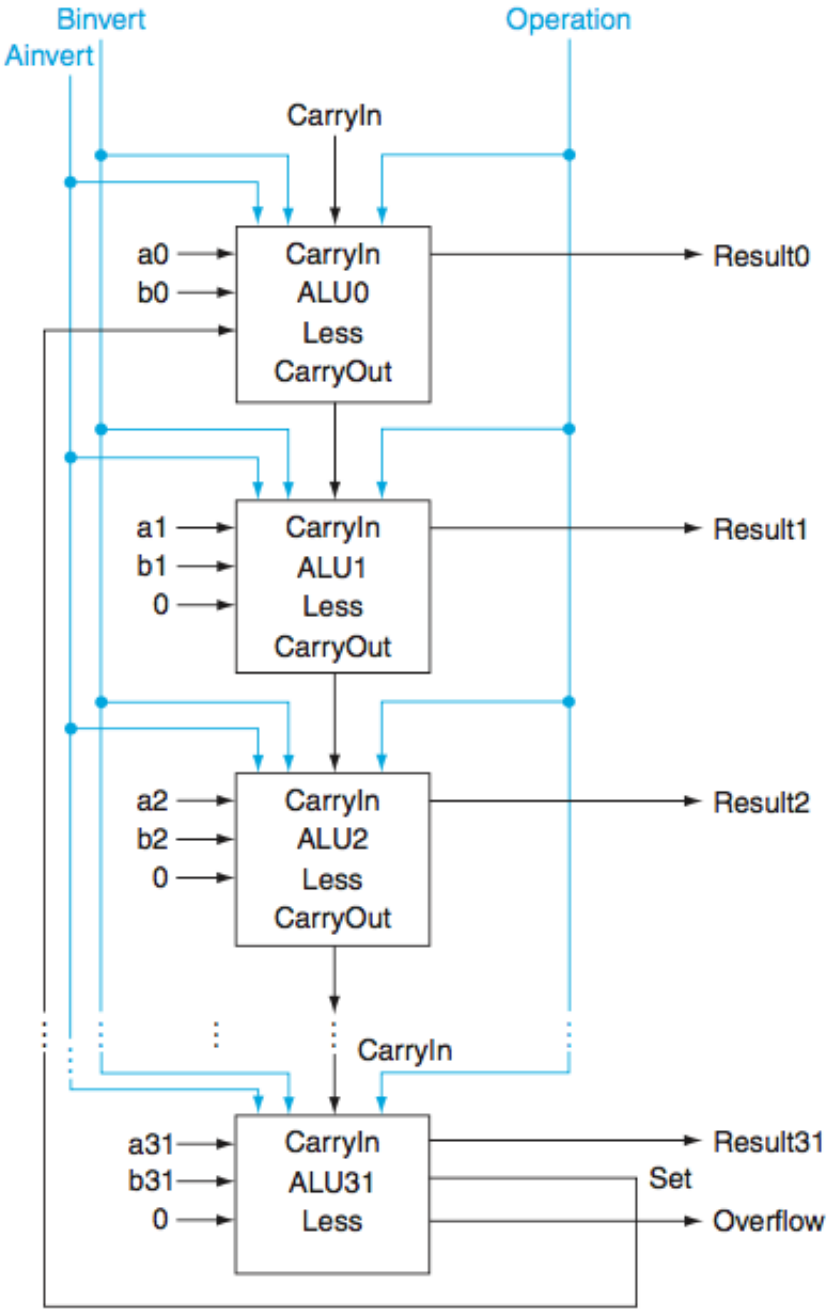
32 Bit ALU

- If operation is 11, ainvert is 0, binvert is 1, and carry in (for bit 0) is 1, then
- If A ¡ B. then result is 1; otherwise result = 0
- The Zero output of the ALU is 1 if all 32 bits of the result are 0; otherwise Zero is 0

32 Bit ALU Black Box

- A, B and result are 32 bit wide
- Zero, Overflow and carry out are 1 bit
- ALU operations is 4 bit wide

- The 4 bits of ALU operations are ainvert, binvert, and the 2 operation bits of the ALU design



ALU Operations

ALU Operations	Functions
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR