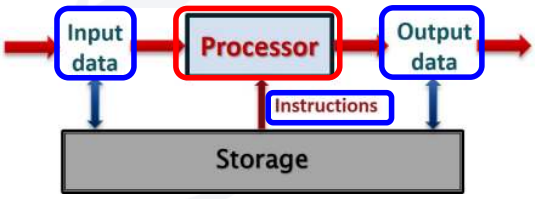
ENS1161 Computer Fundamentals

Module 4
Basic computation in computers

Moving forward..

- Last module:
  - how various types of data are formatted
  - the general format of instructions
  - methods by which operands are retrieved for the processor to use (addressing modes)
- Focus of this module: How data is processed
  - Mainly numerical processing



Module Objectives

On completion of this module, students should be able to:

- Carry out binary addition of unsigned and signed 2s complement integers
- Describe the function and structure of a basic ALU
- Describe how the ALU generates flags from binary addition, where they are stored and how they are used
- Work out the flags set from binary addition and accordingly represent results correctly when out of range conditions occur
- Describe how basic arithmetic and logic functions work and can be used
- Explain the how flags relate conditional branch instructions
- Describe factors that affect computing power and methods of measuring it

Introduction

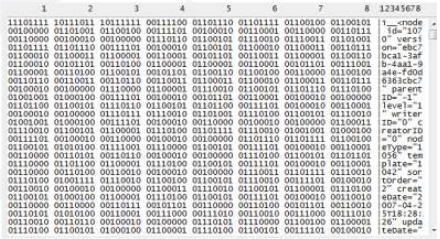
- **Module Scope**
  - Binary addition, including the generation of C, N, V flags
  - Arithmetic and logic operations
    - Addition, subtraction, multiplication, division, AND, OR, XOR, shifting, rotate
  - Status/flags register, interpretation of flags in in computation

Data & Instructions *(recap)*

- Bytes stored in the memory unit of a computer can represent a number of different things:
  - Binary numerical data
  - Coded data (text, images, sound, video, etc)
  - Instruction codes
  - Operand or program branching addresses

Primarily input and output data

Related to program instructions



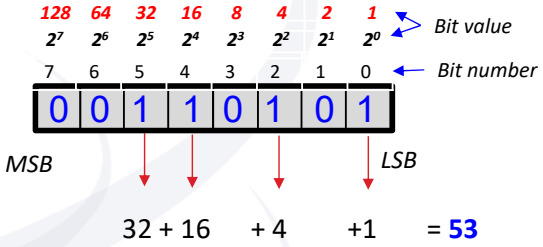
- Difference lies only in context
  - How it is used / interpreted
- Else they cannot be differentiated
  - Just a bunch of bits

Numerical data *(recap)*

- **Integer data**
  - Value of the number in binary
  - The larger the word size the greater the range of values that can be represented
    - More bit patterns possible
  - **Unsigned integers**
    - with an 8-bit word size the largest value is 255 *(2^8-1)*
    - with 16-bits it is 65,535 *(2^{16}-1),* and so on
  - **Signed integers**
    - 2's-complement system used to represent signed integers
    - The most significant bit represents the sign bit
    - an 8-bit signed word can represent numbers from -128 to +127
    - a 16-bit signed word numbers from -32,768 to +32,767

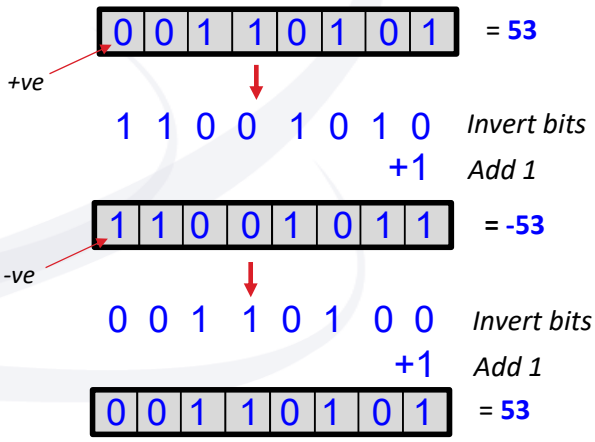
Unsigned integers in binary *(recap)*

- Each bit (place value) represents an increasing power of 2
- Each place value multiplied by 1 or 0
  - According to the bits in the number
- Terminology:
  - *LSB* : **L**east **S**ignificant **B**it
    - The rightmost bit (Bit 0), represents 2^0
  - *MSB* : **M**ost **S**ignificant **B**it
    - Leftmost bit
    - Value depends on number of bits in number
    - For *n* bits, MSB represents 2^{n-1}



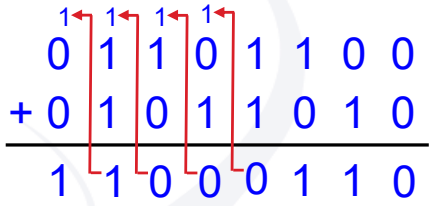
Signed 2s complement numbers *(recap)*

- Most common method of representing signed numbers in computers
- Split the binary combinations between positive and negative numbers
  - MSB = 0 for positive
  - MSB = 1 for negative
- 2s complement process
  1. *Invert all bits (complement)*
  2. *Add 1*
- The process converts numbers
  - +ve to –ve
  - -ve to +ve

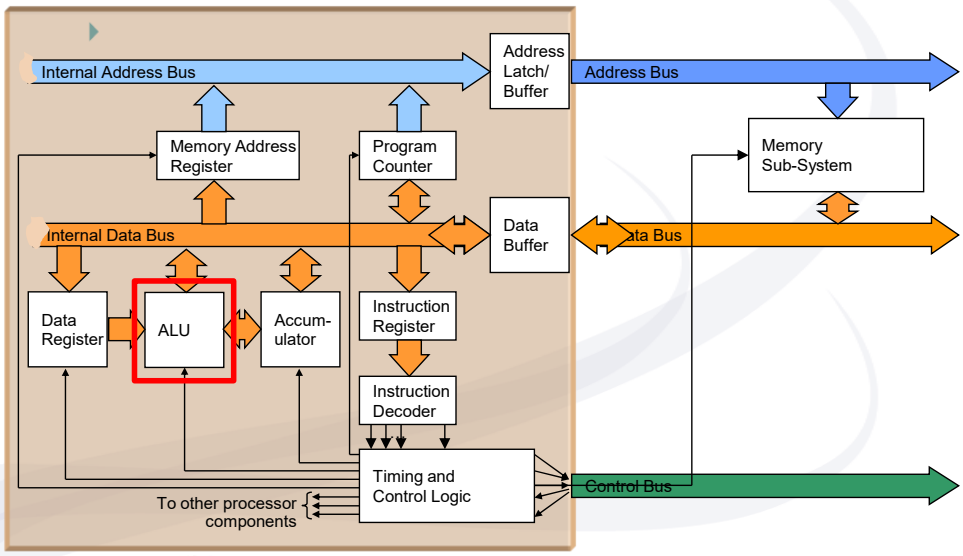


Binary Addition *(review)*

- 1st column: 0 + 0 = 0
- 2nd/3rd column: 1 + 0 = 0 or 0 + 1 = 1
- 4th column: 1 + 1 = 10
  - Result is 0
  - *Carry out* 1 to next bit addition
- 5th /6th column: add two 1s (including *carry in* bit) so result is 10
- 7th column: 1 + 1 + 1 = 11
  - Result is 1
  - *Carry out* 1 to next bit addition

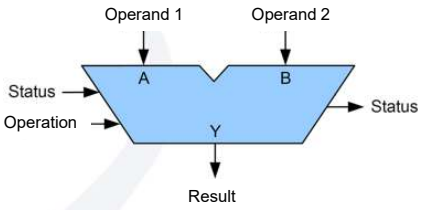


Simplified Diagram of a Processor (recap)



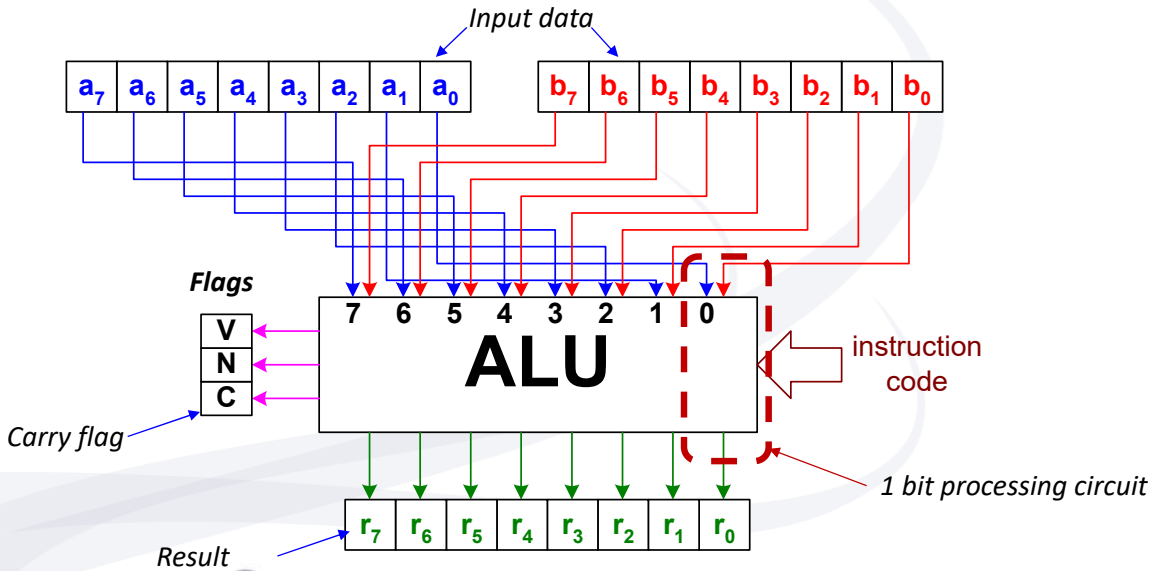
The Arithmetic Logic Unit (ALU) *(recap)*

- Electronic hardware designed to perform operations
  - Arithmetic operations : add, subtract, multiply, etc.
  - Logic operations : AND, OR, NOT, etc.
  - Binary data operations : Shift left/right, rotate, etc.



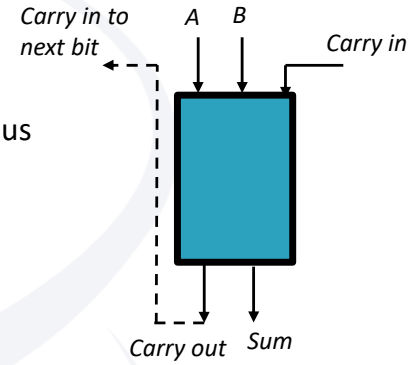
- Data to operate on (called operands) come from connected registers
  - Can have one or two operands, depending on operation
- What operation to perform is determined by signals from the Control unit
  - Based on current instruction
- The result is stored in a register (most commonly the *accumulator*)

Generic ALU structure

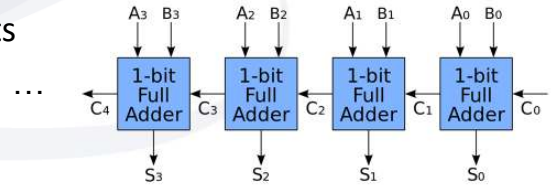


Adder circuits

- ***Full adder circuit***
  - Adds 1 bit from each data word and *carry in* from previous
  - Produces a *sum* (1 bit) and *carry out*
- Each full adder circuit adds 1 bit
- *Half adder circuit*
  - Same as full adder except no *carry in*



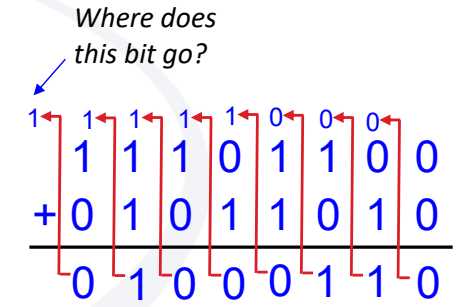
- The ALU has a number of these adder circuits cascaded together for **Add** function

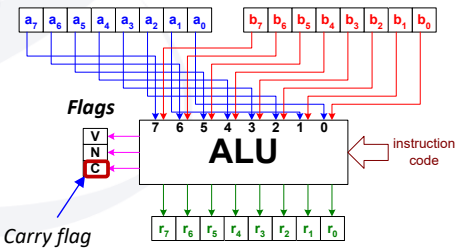


- *Note: Details of Full adder and half adder circuits are outside the scope of this unit*

Carry flag, C

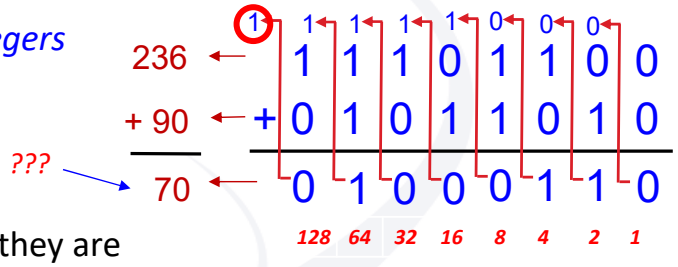
- There is a carry bit generated from the last bit (MSB)
- This 'extra' bit gets stored in the *Carry flag*
- The Carry flag is 1 bit in the *Flags register*
  - Can either be 1 (*True*) or 0 (*False*)
- *C = 1 (Carry is True)*
  - means the answer is too big to store in the result bits generated by the ALU if interpreted as an *unsigned integer*
  - Need more bits to store the answer





Extending unsigned integer results

- Interpreting sum as *unsigned integers*



- Result bits do not make sense as they are
- *Carry flag* = 1
  - Answer too big to fit
- Need to extend answer (double number of bits) and add carry in there

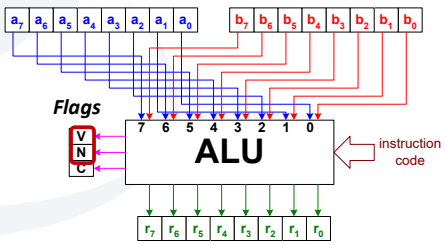
✓ 326 ← 0 0 0 0 0 0 0 1 0 1 0 0 0 1 1 0

256 128 64 32 16 8 4 2 1

- Answer is now correct

ALUs 'blind' addition

- The ALU (hardware) does the binary addition of the bits without consideration of what the bits represent
  - How to correctly use and interpret the result is in the hands of the programmer
  - Same for any other function / operation
- What if the bits being added are meant to represent *signed integers* (2s complement format) and not unsigned integers?
- How to handle *out of range* answers?
- ALU provides 2 other flags (automatically)
  - N flag – Negative
  - V flag - oVerflow



Range of integer values *(recap)*

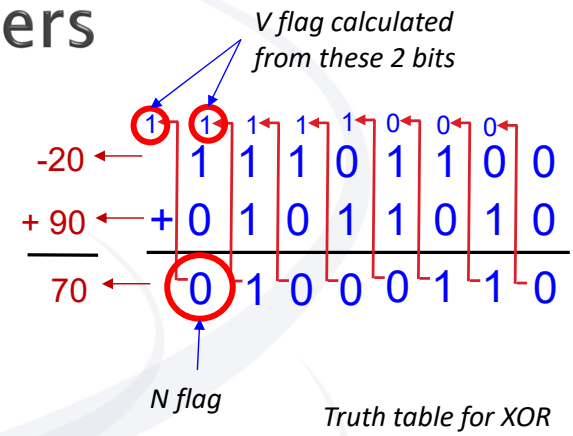
- An *n*-bit number can have 2^n different patterns
- For *unsigned integers*, this is from 0 to 2^n-1
- For *signed integers*, the patterns are split into two
  - Half for positive (including zero)
  - Half for negative

Signed	Unsigned	2's Complement
+5	5	0000 0101
+4	4	0000 0100
+3	3	0000 0011
+2	2	0000 0010
+1	1	0000 0001
0	0	0000 0000
-1	255	1111 1111
-2	254	1111 1110
-3	253	1111 1101
-4	252	1111 1100
-5	251	1111 1011

Range of integers	8-bit	16-bit
Unsigned	0 to 255	0 to 65535
Signed	-128 to +127	-32768 to +32767

Adding signed integers

- Interpreting sum as *signed integers*
- *N flag* = 0
  - Answer is positive
- *V flag* = *MSB Carry in* ⊕ *MSB Carry out*
  - ⊕ is Exclusive-OR (XOR) function
- So in example, *V* flag = 1 ⊕ 1 = **0**
  - i.e. number is in valid range
- ∴ Answer is correct

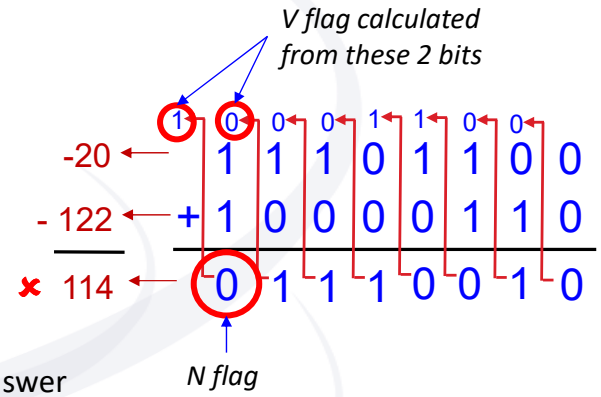


Truth table for XOR

p	q	p⊕q
0	0	0
0	1	1
1	0	1
1	1	0

Extending signed integer results

- Interpreting sum as *signed integers*
- *N* flag = 0
  - Flag means answer is positive
  - Answer does not make sense!
- *V* flag = 0 ⊕ 1 = 1
  - i.e. number is NOT in valid range
  - Needs to be extended to accommodate answer
- Double number of bits, extra bits are OPPOSITE of *N* flag



Sign bit for extended answer

✓ -142 ← 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 0

Other ALU generated flags

- Some flags are automatically generated by the ALU
  - Stored in the flags register
  - E.g. *C*, *N*, and *V* flags
- Zero *(Z)* flag
  - set to 1 (*true*) if ALU operation result = 0
  - A most common (and useful) ALU generated flag
- Half carry *(H)* flag
  - set to 1 (*true*) if there is a carry out from bit 3 into bit 4
- These flags are generated every time the ALU performs an operation
  - It is up to the program(mer) to decide how to use these flags
  - Based on the type of data and what needs to be done

Flags register

- Flags register is a special register where each bit represents a particular flag
- A flag is a Boolean variable (1 bit)
  - Can either be 1 (*True*) or 0 (*False*)
- Flags normally indicate some kind of *condition* or *status*
  - Flags register also known as *condition code register (CCR)* or *status register*
- Some flags are related to ALU operations and some have other purposes
  - E.g. control purposes such as enabling or disabling interrupts *(I flag)*
- Each processor may have slightly different names for the common flags
  - The flags will also be stored in different bit positions in the flags register

Example Flags register

I	x	H	S	V	N	Z	C

8088 flags register (example)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	R	R	R	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

R = reserved
U = undefined
OF = overflow flag
DF = direction flag
IF = interrupt flag
TF = trap flag
SF = sign flag
ZF = zero flag
AF = auxiliary carry flag
PF = parity flag
CF = carry flag

- 6 *conditional flags*, indicate some condition resulting after an instruction executes - **CF**, **PF**, **AF**, **ZF**, **SF**, and **OF**
- Other 3 *control flags*, control the operation of instructions *before* they are executed.

8088 flags register (example)

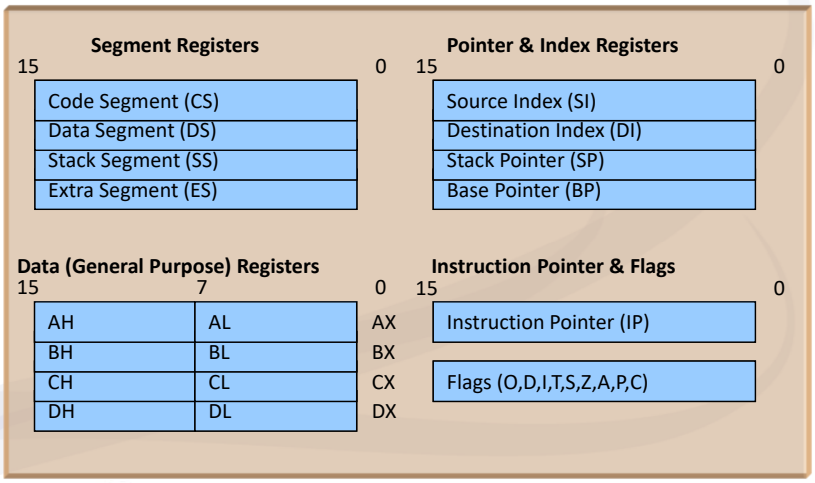
- **Conditional flags**
  - **O** = Overflow flag : set if signed result is out of range
  - **S** = Sign flag : high for negative 2's complement values
  - **Z** = Zero flag : set if any result = 0
  - **A** = Auxiliary carry flag : carry from b3 to b4, used for BCD calculations
  - **P** = Parity flag : set if any result has even parity
  - **C** = Carry flag : set if any calculation generates a carry
- **Control flags**
  - **D** = Direction flag : used for certain instructions (0 = Up)
  - **I** = Interrupt enable flag : to enable/disable external interrupt requests
  - **T** = Trap flag : used for debugging

Arithmetic Instructions

- All processors include some basic arithmetic operations
- Many early processors included only addition and subtraction instructions
  - These were only for fixed point arithmetic
  - Other more complex arithmetic operations must be carried out in software by performing appropriate sequences of these simple instructions
- Subsequent processors included multiplication and division instructions
- Floating point operations were later added
  - Initially carried out by 'maths co-processors' – separate special function processors
- Some examples of basic arithmetic and logic instructions given following
  - Based on the old **8086**, but still exist in current generation processors (e.g. *Core i7*)

The 8086/8088 Architecture

- **Programming Model of the 8086/8088 – Registers**



Addition and Subtraction

- **ADD** *destination*, *source*
  - **Adds** the two specified data values and places the result into the destination
  - *dest* ← *dest* + *src*

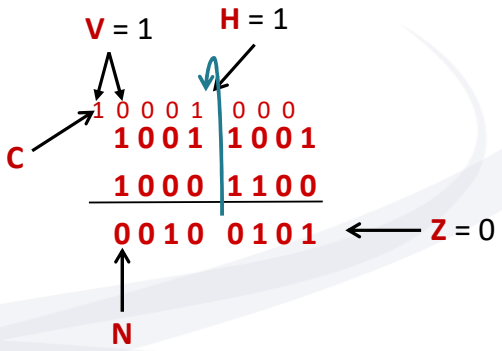
  E.g. ADD AL, 55h ; AL + 55h, result in AL, 55h is an immediate value
  ADD BH, AH ; BH + AH, result in BH, BH and AH both registers

- **ADC** *destination*, *source* (Add with Carry)
  - Adds value of Carry flag as Carry In to first bit
  - *dest* ← *dest* + *src* + *C*

- **SUB** *destination*, *source*
  - Used in the same way as the ADD instruction, but performs subtraction
  - *dest* ← *dest* - *src*

Arithmetic Instructions and flags

- **ADD Instruction**
  - Example of flags set
  - **ADD AL, BL**



  - **Result:**
  - What are the **C**, **N**, **V**, **H** and **Z** flag values after the operation above?

Increment and Decrement

- The most common arithmetic operations are incrementing (+1) or decrementing (-1) a register or variable
- These generally have their own (optimised) instruction
  - No need to specify / fetch a second operand
- **INC** *dest*
  - **Adds 1** to the destination
  - *dest* ← *dest + 1*

    E.g. INC AL ; AL = AL + 1
- **DEC** *dest*
  - **Subtracts 1** from the destination
  - *dest* ← *dest - 1*

Multiplication

- When multiplying 2 numbers, the result may need double the number of digits
  - E.g. Multiplying 2 single-digit decimal numbers may result in a 2 digit answer
    - 9 x 9 = 81
  - Same principle applies in binary – answer requires double the number of bits
  - **E.g. MUL** *operand*
    - Second number by default is in the AL (8-bit) or AX (16-bit) register
    - If operand is 8-bit :

8 bit *8 bit* ***16 bit result***
operand * AL → AX

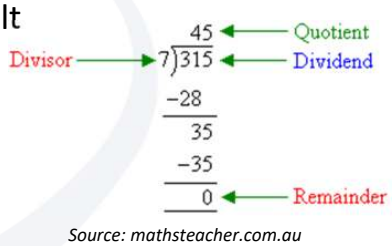
    - If operand is 16-bit:

16 bit *16 bit* ***32 bit result :*** *concatenation of 2 16-bit registers*
operand * AX → DX:AX

  - **N.B.** The IMUL instruction should be used for operations on signed data values

Division

- When dividing numbers, there are 2 parts to the result
  - The **quotient** (whole number answer)
  - The **remainder**
    - E.g. 53 / 5 = 10 remainder 3



Source: mathsteacher.com.au

  - **E.g. DIV** *operand* (*divisor*)
    - The dividend by default is in AX (16-bit) register or DX:AX (32-bit)
    - If operand (divisor) is 8-bit :

16 bit *8 bit* ***8 bit*** ***8 bit***
AX ÷ operand = quotient → AL, remainder → AH

    - If operand is 16-bit:

32 bit *16 bit* ***16 bit*** ***16 bit***
DX:AX ÷ operand = quotient → AX, remainder → DX

  - **N.B.** The IDIV instruction should be used for operations on signed data values

Logical Instructions

- All processors have a number of instructions that perform *bitwise* logical operations
- **AND / OR / XOR** *dest*, *source* *(8086 example)*
  - Performs the logical operation bit-by-bit between all bits of the specified operands
  - The result is stored in the first specified operand

E.g.
```
MOV AL, 01010101b      ; puts the specified bits in AL
AND AL,  10101011b     ; AND with bits in operand
```

This will result in AL= 00000001b

Logical Instructions

- **AND Masking**
  - Common use for the **AND** instructions
  - To selectively ***clear*** specific bits of a data word
  - Done by AND-ing the data word with another word called a ***mask***
    - contains 0s in the bit positions to be cleared
    - 1s everywhere else
  - *Common application: masking of network addresses*

p	q	p AND q
0	0	0
0	1	0
1	0	0
1	1	1

Mnemonic	Comments
MOV AL, 55h	; AL <= 0101 0101b
AND AL, 0Fh	; 0Fh = 0000 1111b
	; AL <= 0000 0101b (05h)

Logical Instructions

- **OR Masking**
  - **OR** instructions can be used to selectively ***set*** specific bits of a data word
  - The OR mask should have
    - 1s in the bit positions to be **set**
    - 0s everywhere else

p	q	p OR q
0	0	0
0	1	1
1	0	1
1	1	1

```
Mnemonic                Comments
MOV CH, 05h     ; CH <=   0000 0101b
OR CH, 30h      ; 30h  =  0011 0000b
                ; CH <=   0011 0101b
                ;   = 35h
```


Logical Instructions

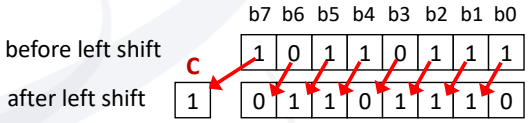
- **Selective Inversion**
  - **XOR** instructions can be used to selectively ***invert*** specific bits of a data word
  - The mask should contain
    - 1s in the bit positions to be **inverted**
    - 0s everywhere else

p	q	p XOR q
0	0	0
0	1	1
1	0	1
1	1	0

```
Mnemonic            Comments
IN AL, 0F7h     ; Load data from input device at F7h
XOR AL, 0Fh     ; Invert data in low nibble of AL
```

Shift Instructions

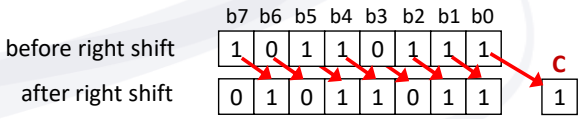
- **Logical Shift**
  - **SHL** *dest*, 1 (Logical Shift Left)
    - Shifts all bits in the destination 1 place to the left.
    - MSB gets shifted into C flag, Bit 0 is cleared
    - Effectively **multiplies the value by 2**



Mnemonic		Comments
MOV AL, $05	; AL <=	0000 0101b (05 hex)
SHL AL, 1	; AL <=	0000 1010b (0Ah, 10 dec)
SHL AL, 1	; AL <=	0001 0100b (14h, 20 dec)

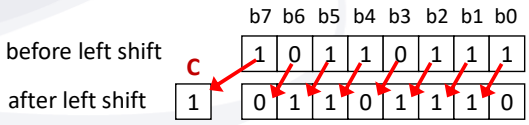
Shift Instructions

- **Logical Shift**
  - **SHR** *dest*, 1 (Logical Shift Right)
    - Shifts all bits in the destination 1 place to the right.
    - Bit 0 gets shifted into C flag, MSB is cleared
    - Effectively **divides the unsigned value by 2**



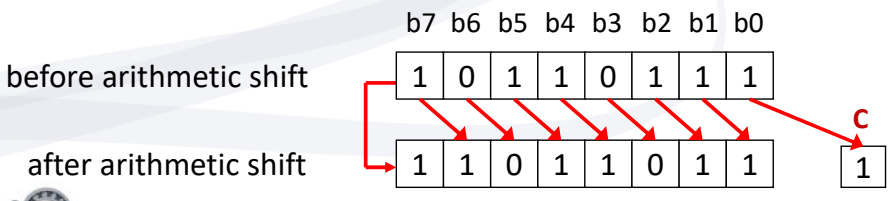
Shift Instructions

- **Arithmetic Shift**
  - **SAL** *dest*, 1 (Arithmetic Shift Left)
    - Works exactly the same as Logical Shift Left
    - Shifts all bits in the destination 1 place to the left.
    - MSB gets shifted into C flag, Bit 0 is cleared
    - Effectively **multiplies the value by 2**



Shift Instructions

- **Arithmetic Shift**
  - **SAR** *dest*, 1 (Arithmetic Shift Right)
    - Works similar (but not exactly the same) as SHR
    - Shifts all bits in the destination 1 place to the right, Bit 0 into C flag
    - Also effectively **divides the value by 2** but assumes a **signed number**
    - So sign bit is fed into MSB to ensure that sign stays the same



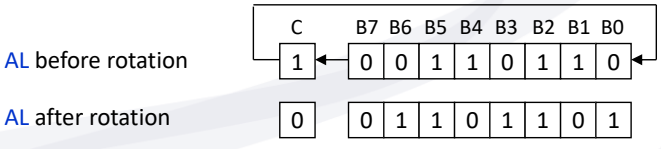
Rotate Instructions

- **ROL / ROR** *dest*, 1 (Rotate Left / Right)
  - Performs left or right rotations on the register
  - Similar to shift left / right, except that bit that 'falls out' moved to the opposite end

Rotate Instructions

- **RCL / RCR** *dest*, 1 (Rotate through Carry Left / Right)
  - Similar to ROL / ROR except that Carry flag is included in the rotation

- E.g. RCL AL, 1



Instructions that only set flags

- **Compare Instruction**
  - **CMP** *operand 1*, *operand 2*
    - Performs a pseudo-subtraction: *operand 1* – *operand 2*
    - The S, C, O, Z and P flags are set based on the result
    - But the result is not actually stored anywhere
    - The flags can then be checked / used via branch instructions
      - *Refer 2 slides down for more details on conditional branching*

E.g. CMP AL, BL

	C / S	Z
AL > BL	0	0
AL = BL	0	1
AL < BL	1	0

Instructions that only set flags

- **Test Instruction**
  - **TEST** *operand 1*, *operand 2*
    - Performs a bit-by-bit pseudo-AND : *operand 1* AND *operand 2*
    - The Z, S and P flags are set accordingly
    - The result is not actually stored anywhere
  - AND masking can be used to isolate and test one or more bits
    - Check whether they are set or cleared
    - Does not affect its contents

```
E.g.    TEST AL, 00001000b    ; Test bit 3 in AL
        JNZ Somewhere         ; Jump if it is set (Not Zero)
```

NB: The Z flag will be set if bit 3 in AL is 0, and cleared if it is 1

Conditional Branch Instructions

- Most conditional branch instructions are designed to be used after a CMP or arithmetic operation between two operands
  - e.g. A and B
  - Action whether to branch (jump) or not based on particular flags

Mnemonic	Condition	Mnemonic	Condition
JA	A > B (unsigned)	JC	C flag = 1
JAE	A ≥ B (unsigned)	JNC	C flag = 0
JB	A < B (unsigned)	JNO	O flag = 0
JBE	A ≤ B (unsigned)	JNP	P flag = 0
JE	A = B (Z = 1)	JNS	S flag = 0
JG	A > B (signed)	JNZ	Z flag = 0
JL	A < B (signed)	JO	O flag = 1
JNA	A ≯ B (unsigned)	JPE	P flag = 1
JNE	A ≠ B	JS	S flag = 1
JNL	A ≮ B	JZ	Z flag = 1

Use of Conditional Branching

- Conditional branching instructions are the basis of important programming concepts like:
  - Conditional statements (IF … THEN … ELSE..)
  - Loops
- This will be covered in the *next module*

Multipliers and their Symbols

'Normal' decimal

Multiplier	Name	Symbol
10^3	kilo	k
10^6	mega	M
10^9	giga	G
10^{12}	tera	T
10^{15}	peta	P
10^{18}	exa	E

Binary multipliers – used in computing

Multiplier	Value	Name	Symbol
2^{10}	$1024 \approx 10^3$	kilo	K *(or Ki)*
2^{20}	$1{,}048{,}576 \approx 10^6$	mega	M *(or Mi)*
2^{30}	$\approx 10^9$	giga	G *(or Gi)*
2^{40}	$\approx 10^{12}$	tera	T *(or Ti)*
2^{50}	$\approx 10^{15}$	peta	P *(or Pi)*
2^{60}	$\approx 10^{18}$	exa	E *(or Ei)*

While they use the same symbols, the values of each multiplier differ slightly

- Multiples of 1000 vs multiples 1024

Decimal vs binary multipliers

- Which multiplier to use depends on the context
  - Normal physical parameters vs parameters within the computer
  - E.g. Physical: 1 GHz clock = 10^9 Hz (cycles / second)
  - E.g. Computing: 4GB RAM = 4 x 2^{30} bytes = 4,294,967,296 bytes
- Number of bits in address bus defines the addressable memory
  - E.g. 16 bits address bus, addressable memory = $2^{16} = 2^6.2^{10}$ = 64 KB memory

Address bus size	Addressable memory	Equivalent
16-bit	$2^{16} = 2^6 \cdot 2^{10}$	64 KB *(or KiB)*
24-bit	$2^{24} = 2^4 \cdot 2^{20}$	16 MB *(or MiB)*
32-bit	$2^{32} = 2^2 \cdot 2^{30}$	4 GB *(or GiB)*
64-bit	$2^{64} = 2^4 \cdot 2^{60}$	16 EB *(or EiB)*

Other multipliers and their Symbols

'Normal' decimal

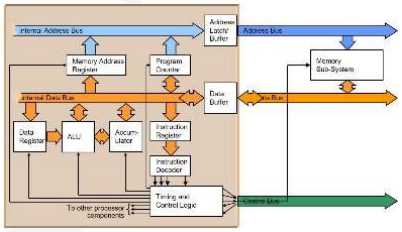
Multiplier	Name	Symbol
10^{-3}	milli	m
10^{-6}	micro	μ
10^{-9}	nano	n
10^{-12}	pico	p
10^{-15}	femto	f
10^{-18}	atto	a

These are used in measuring physical parameters

- E.g. memory access time in *ns* (nanoseconds)

Factors that affect computing power

- **ALU width**
  - Number of bits the ALU can process at one time
    - 8-bit, 16 bit, 32 bit, 64 bit
  - Normally matches the size of the internal data bus and registers



- **Number of computing operations that can be performed in a second**
  - Depends on a number of factors
    - Clock speed, number of microprogram steps per instruction
  - Also depends what each operation can do
    - E.g. integer addition vs. floating point multiplication

Some measures of computing power

- **Clock Speed**
  - Simplest measure of processor power
  - Least meaningful as each instruction may take different number of clock cycles
  - Also does not take into account processors may have multiple cores
- **Millions of Instructions per second (MIPS)**
  - Generally based on integer only computation
  - Only meaningful if processors being compared have similar architecture
    - ALU width, etc.
- **Floating-point Operations Per Second (FLOPS)**
  - Floating point operations form basis of 'real' computations – including graphics
- All these measures are only indicative - many other factors come into play

Module Objectives

On completion of this module, students should be able to:

- Carry out binary addition of unsigned and signed 2s complement integers
- Describe the function and structure of a basic ALU
- Describe how the ALU generates flags from binary addition, where they are stored and how they are used
- Work out the flags set from binary addition and accordingly represent results correctly when out of range conditions occur
- Describe how basic arithmetic and logic functions work and can be used
- Explain the how flags relate conditional branch instructions
- Describe factors that affect computing power and methods of measuring it